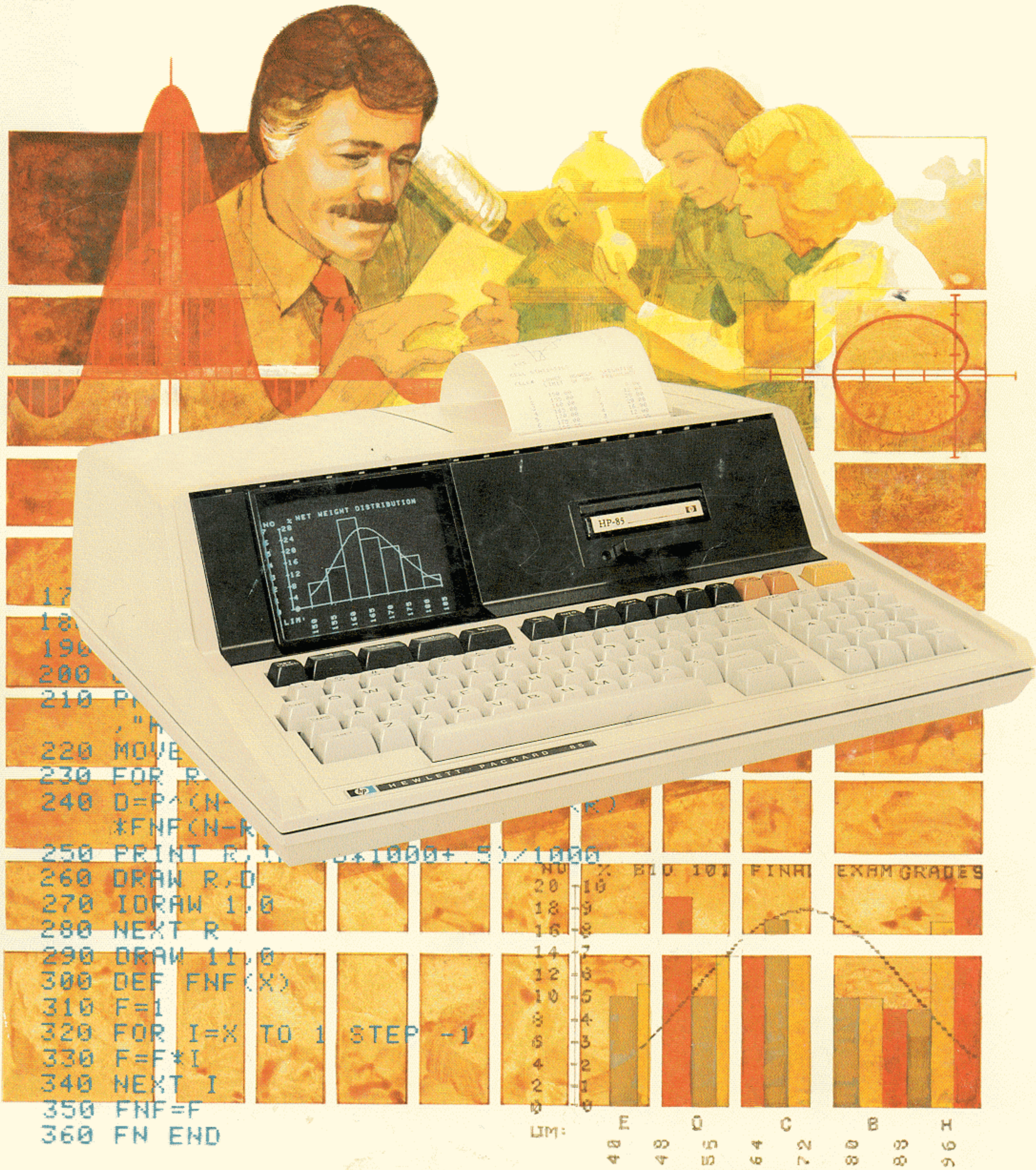
HEWLETT-PACKARD

HP-85

OWNER'S MANUAL AND PROGRAMMING GUIDE



HEWLETT PACKARD

HP-85

Owner's Manual

and

Programming Guide

January 1981

00085-90002 Rev. D 1/81

Printed in U.S.A.

Contents

Meet the HP-85 Personal Computing System

Your Hewlett-Packard 85 Personal Computer is a versatile, self-contained, personal computing device which enables you to perform a wide variety of useful and interesting functions. To mention just a few of the special features of your HP-85, you have the ability to:

- Perform calculations in a simple, straightforward manner—as if you had a calculator with dozens of mathematical and scientific functions.
- Compose programs in BASIC (Beginner's All-Purpose Symbolic Instruction Code) programming language. The HP-85 exceeds the latest American National Standard for Minimal BASIC. In many areas, the HP-85 includes *enhancements* to this standard.
- Execute BASIC programs. After you have written your programs, they may be executed, often at the touch of one key. The HP-85 offers several *typing aids* to your program execution and control.
- Load and store programs and data on a magnetic tape cartridge with the *built-in* tape drive. Thus you may permanently store your programs to be retrieved again, whenever you wish.
- List programs and data with the *built-in* thermal printer. Not only can you list programs, but you can copy anything that appears on the display onto the printer, to record and review your results.
- Perform graphics. The graphics capabilities of the HP-85 are sophisticated, yet easy to use. And again, anything that you can "draw" on the display can be transformed to hard copy with a single command: COPY.
- Edit, correct, and modify anything that appears on the display with *tremendous* ease. In fact, the HP-85 allows you to access and review 64 lines of characters on the display, and to edit them at your convenience.

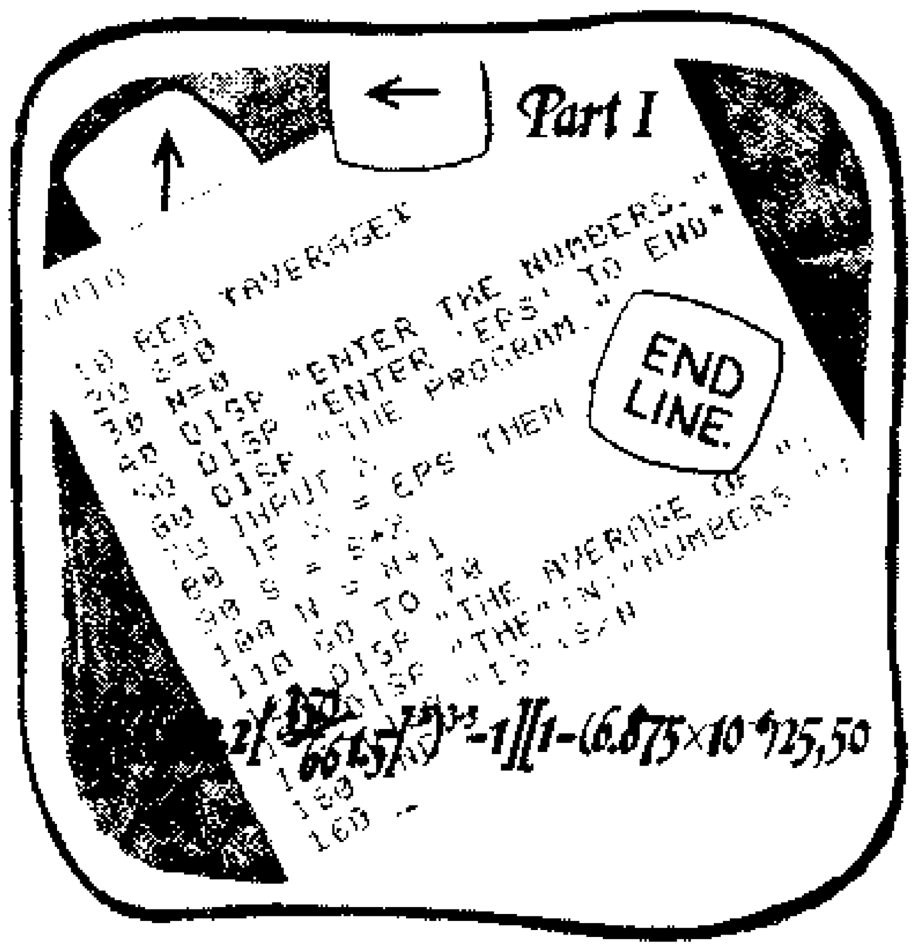
Now let's take a closer look at the HP-85 system to see how easy it is to use, whether we solve a problem manually, use one of the sophisticated prerecorded programs from the Standard Pac, or even write our own program.

How to Use This Manual

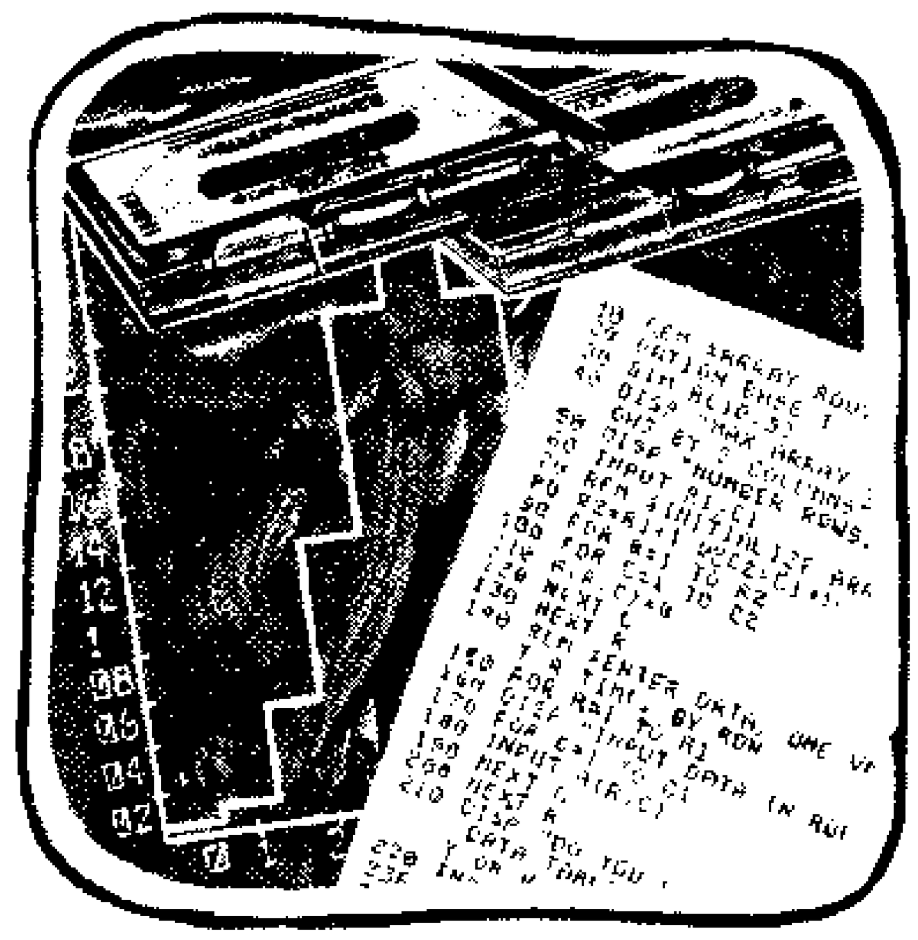
This handbook has been designed to enable you to use the utmost potential of your HP-85 Personal Computer and to answer your questions concerning BASIC programming with the HP-85.

If you have just received your new HP-85 Personal Computer, read appendix B before you attempt to operate the system. Appendix B contains initial set-up instructions and other pertinent owner's information.

Then familarize yourself with the HP-85 system by reading and following through the examples in part I of this handbook—with your computer. The best way to feel at ease with the system is to sit down with the owner's handbook and the HP-85 and actually key in the examples provided in each of the sections. It won't take long to become familiar with the system, and it's well worth the time you invest to obtain a more complete understanding of your HP-85. Even if you are an advanced programmer, you will benefit from the unique features and capabilities of your HP-85 that are introduced in part I.



Part II of the *HP-85 Owner's Manual and Programming Guide* discusses each of the BASIC statements used with the HP-85. It also covers tape cartridge operations, graphics on your HP-85 system, and debugging procedures. There are problems for you to work at the end of most of the sections in part II and, in case you get stuck, sample solutions are given in appendix F.



If you are a beginning programmer, and you have difficulties with part II, you may wish to refer to the HP-85 BASIC Training Pac. The pac is designed to help you get acquainted with the HP-85 and BASIC programming.

If you are an experienced programmer, you'll probably start programming with the HP-85 as soon as you've read part I. You can use part II as a reference guide to particular BASIC statements, but you'll probably find the *HP-85 Pocket Guide* and the *HP-85 BASIC Reference Card* most suited to your BASIC reference needs.

Where can you go next? After you've become familiar with the HP-85 itself, you may wish to enhance your programming capabilities with specific applications pacs, a memory module, extended capability ROMs, and peripherals. Be sure to check the accessories list in appendix A.

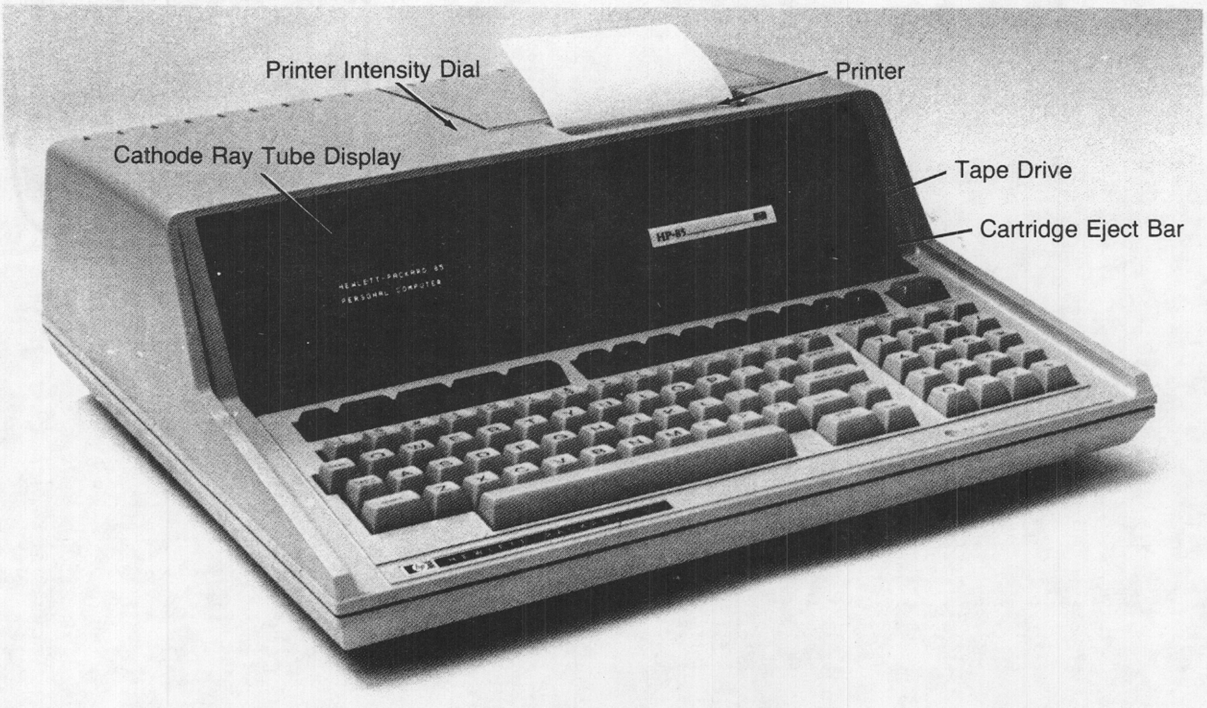
CAUTION

The inspection procedure and initial set-up instructions for the HP-85 are presented in appendix B of this manual. Please refer there:

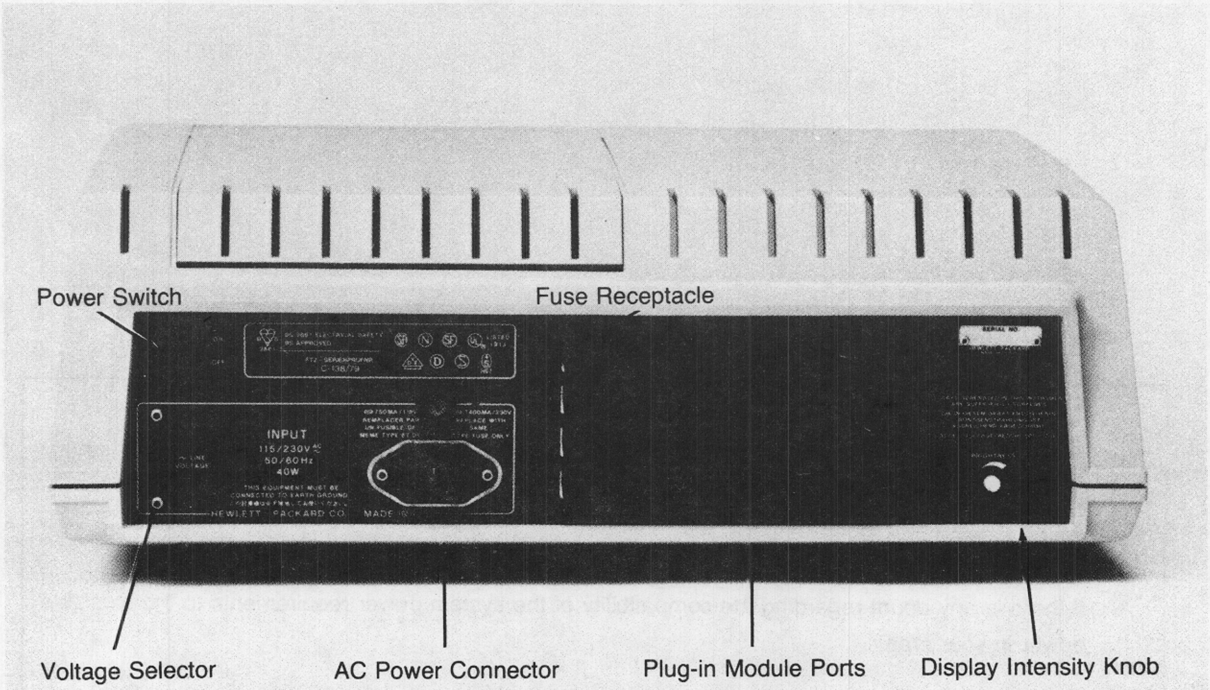
- If you have not inspected the HP-85.
- If there is any doubt regarding the compatibility of the system power requirements to the available power in your area.

Do not attempt to set up the HP-85 without first becoming thoroughly familiar with appendix B; it contains information that is important to avoid damaging your personal computer when it is initially set up.

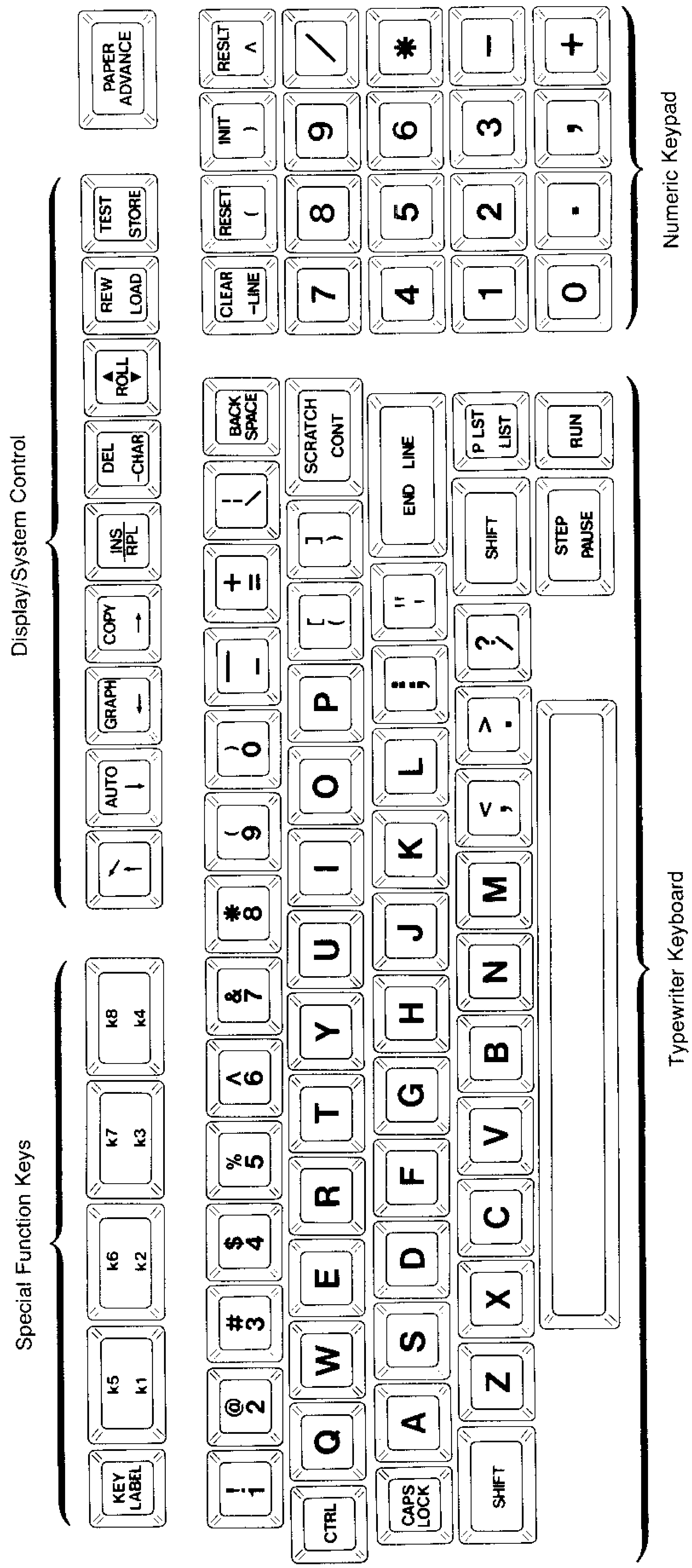
An Overview of the Hewlett-Packard 85



Front View



Back View



HP-85 Keyboard

HP-85 Key Index

Typewriter Keys

(A) through (Z). Alphabetic keys. In BASIC mode produce capital letters, and when used with (SHIFT) or (CAPS LOCK) produce small letters. In "typewriter" mode produce small letters, and when used with (SHIFT) or (CAPS LOCK) produce capital letters **(page 33).**

(SHIFT) Shift key. Used with the alphabetic keys to get reverse letter-case; with other keys to select alternate symbol, statement, or command on upper half of key **(page 33).**

(CAPS LOCK) Caps lock. Affects only the alphabetic keys. When pressed and locked, reverses the letter case of current typing mode **(page 33).**

(CTRL) Control. Used to select characters that are not normal typewriting characters and to output keycodes with decimal values below 32 **(page 34).**

Numerics, punctuation, symbols. The remainder of the typewriter keys operate like a standard typewriter. To select the symbol on the upper half of a key, hold (SHIFT) while you press the key **(page 33).**

(END LINE) Enters an expression, statement, or command into the computer to be interpreted and/or executed. Also performs a carriage return **(page 37).**

Numeric Keys

(0) through (9) Digits. (.) Decimal point. Used for keying in numbers **(page 18).**

(+) (−) (*) (/) (^) (\)
Arithmetic operators: addition, subtraction, multiplication, division, exponentiation, and integer division, respectively **(page 43).**

(( ) ( )) Parentheses. Used to key in numeric expressions and to enclose the arguments of functions **(page 45).**

(,) Comma. Separates input items and used as a separator in functions, statements, and commands **(page 47).**

(RESLT) Recalls to the display the most recently calculated result **(page 46).**

Special Function Keys

(k1) through (k4) (unshifted) and (k5) through (k8) (shifted). Special function keys for user-defined functions. Must be defined in a program **(page 154).**

(KEY LABEL) Recalls the current labels for the special function keys and displays them on the CRT. Also moves the cursor to the upper left corner of the display **(page 154).**

Display Control

(↖) (↑) (↓) (←) (→) Positions the cursor on the CRT display in the direction of the arrow, without erasing characters **(page 19).**

(INS RPL) Insert/Replace. Toggles between insert mode and replace mode. When the cursor is under a character in *replace* mode, typing any character will replace the character at the cursor position.

In *insert* mode, two cursors appear and the next character typed will be inserted between the characters marked by the cursor locations **(page 39).**

(-CHAR) Deletes the character above the cursor **(page 38).**

(-LINE) Deletes a line from the cursor to the end of the line **(page 19).**

(BACK SPACE) Erases characters as it backspaces **(page 38).**

(SHIFT) (BACK SPACE) Backspaces rapidly **(page 38).**

(CLEAR) Clears 16 lines of the display from the cursor position, then rolls the information above the cursor out of view and homes the cursor **(page 38).**

(ROLL) Recalls information that has "rolled" out of view. Pressing (ROLL) rolls down information that has most recently left the display, while (SHIFT) (ROLL) rolls up the oldest information saved on the display **(page 36).**

(GRAPH) Sets the system display to graphics mode, showing the current graphics display. Press any alphanumeric key to return to the normal alphanumeric display **(page 197).**

Cartridge Control

(REW) Rewinds the tape cartridge **(page 280).**

Program Control

(AUTO) Typing aid to display AUTO on the CRT display. The AUTO command instructs the computer to number program statements automatically. You may specify, at your option, the beginning line number and numbering interval; otherwise the system will number program lines beginning with 10 and incrementing by 10. The AUTO command is then executed by pressing (END LINE) **(page 80).**

(DEL) Typing aid to display DELETE on the CRT display. The DELETE command is used to delete a line or a section of a program. DELETE must be followed by the line number, or the first and last line number of a section of a program to be deleted. The DELETE command is then executed by pressing (END LINE) **(page 95).**

(PAUSE) Immediate execute key which halts a running program without otherwise affecting the program. Produces an audible beep when interrupting program execution **(page 97).**

(CONT) Immediate execute key used to continue execution of a program that has been halted by a PAUSE statement **(page 98).**

(INIT) Initializes (allocates memory to) a program without executing it **(page 99).**

(STEP) Executes a single program statement. The program must first be initialized by either RUN or INIT before you can single step through it **(page 259).**

(RUN) Immediate execute key which first initializes the current program, then executes it **(page 99).**

(LOAD) Typing aid to display LOAD on the CRT display. The LOAD command loads a specified file from a tape cartridge. LOAD must be followed by a file name within quotes or a string expression that specifies the file name. The command is then executed by pressing (END LINE) **(page 179).**

(STORE) Typing aid to display STORE on the CRT display. The STORE command stores a specified file onto a tape cartridge. STORE must be followed by a file name within quotes or a string expression that specifies the file name. The command is then executed by pressing (END LINE) **(page 176).**

(LIST) Immediate execute key which displays one full screen of the current program in memory starting at the beginning of a program. Each successive time (LIST) is pressed, another screen full of program lines is displayed until the end of the program is reached. Following the list of the last program line, LIST displays the remaining number of memory locations **(page 75).**

(P LST) Immediate execute key which will list the current program in its entirety on the system printer. Press any key to halt the printer listing **(page 75).**

Printer Control

(PAPER ADV) Moves the paper one line. If the key is held down, the paper advance will repeat until the key is released **(page 35).**

(COPY) Immediate execute key which copies the exact contents of the display onto the system printer **(page 35).**

System Commands

(RESET) Returns the computer to its condition at power on, except that programs are not erased **(page 40).**

(TEST) Performs a functional test of the processor and built-in peripherals **(page 39).**

(SCRATCH) Typing aid to display SCRATCH on the CRT display. The SCRATCH command clears computer memory. The command is executed by pressing (END LINE) **(page 78).**

Part I
Using Your HP-85

Section 1

Getting Started

In this section, we will discuss many topics in relatively few pages so that you can:

- Do a wide variety of calculations in just a few minutes.
- Begin using the editing capabilities of the computer.
- Use the tape cartridges.
- Begin programming.
- Have some fun!

It is our intent to "get on board" fast! For this reason, some of the more sophisticated concepts are greatly simplified or reserved for later sections.

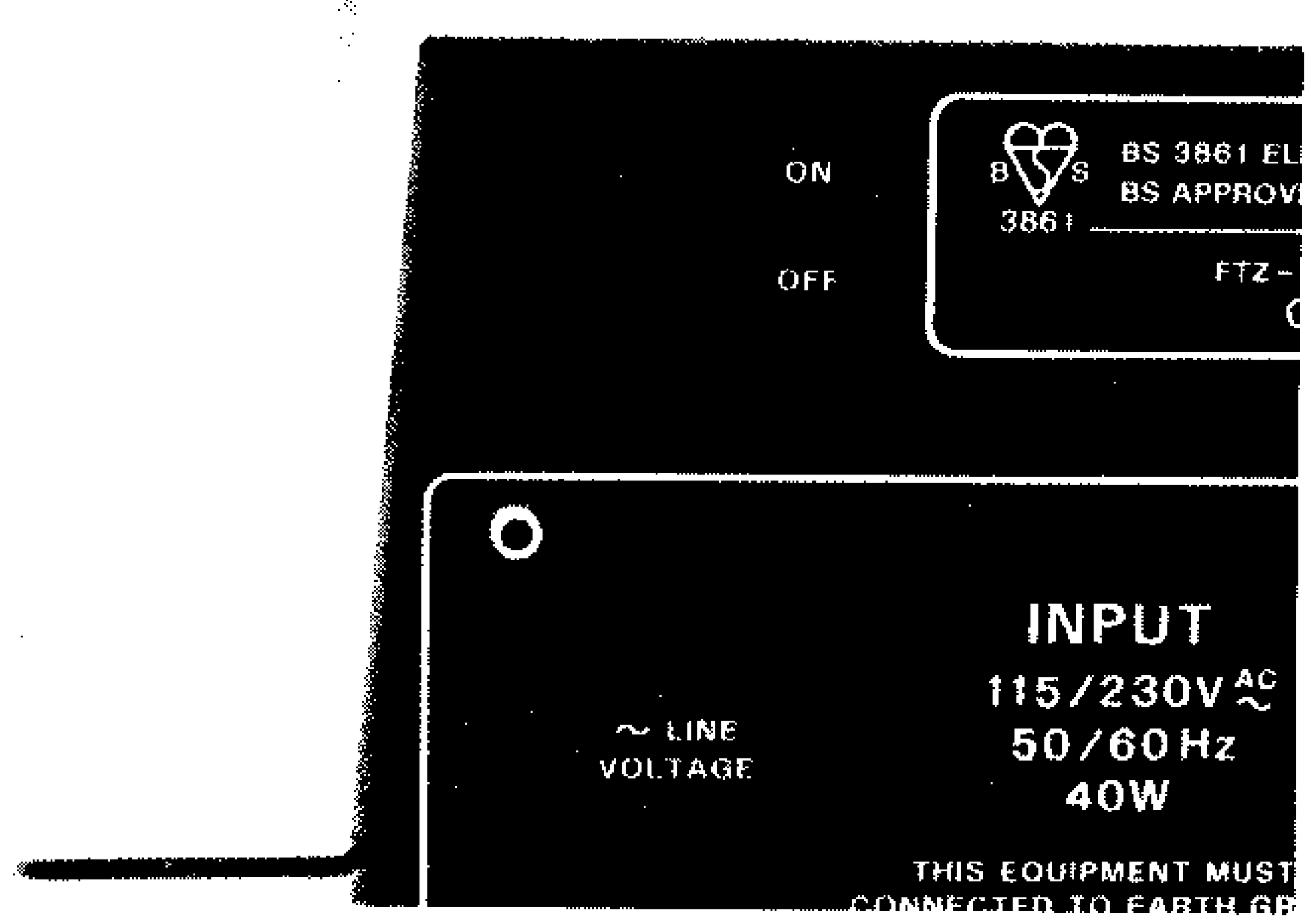
After working through this section, you'll have enough background to try things on your own, which is actually the best way to attain a good working knowledge of your personal computer. *And don't worry, you can't damage the HP-85 with any keyboard operation!*

Power On

There are a few things to notice each time you switch on your HP-85.

If the system is turned off:

- Set the power switch, located on the rear panel of the computer, to the ON position.



- When the cursor (underscore) appears after approximately 3 seconds in the upper left hand corner of the display (the "home" position), the HP-85 is ready to use.

- If a tape cartridge is present in the tape drive, the system will search for a program tape named "Autost" (for automatic start). The autostart routine permits the computer to load and run a program without operator instructions. More about this later.
- The system automatically runs through a self-test routine when the power is switched on. If it finds a problem in the system circuitry, it will beep and display:

```
Error 23 : SELF TEST
```

This message means that your system is not operating properly; contact your local authorized dealer or your nearest HP sales and service office (addresses supplied in the back of this handbook).

If the system is switched on and the tape drive is not being accessed, but the display remains blank, hold down the (SHIFT) key, then press (RESET). This operation resets the system to a ready state (see page 40). Also adjust the display intensity knob on the rear panel of the system. If the display still remains blank, first check the power connection and the fuse, as described in appendix B. For further assistance, call your nearest HP dealer or HP sales and service office.

If the system is on and the cursor is in the home position, you are ready to go!

Before we begin, make sure that the (CAPS LOCK) key is released to the same level as the other keys.

Manual Problem Solving ("Calculator" Mode)

Let's try a few simple calculations to get the feel of your HP-85.

Type in the problems as you see them under the column marked **Press.** You may use either the numbers and arithmetic operators conveniently located on the right side of the keyboard or the numbers and symbols on the typewriter part of the keyboard. When you press (END LINE), the answer will appear on the line below your input.

> **Note:** Any spacing that you use between characters, in manual calculations or in program statements, is totally arbitrary. When you list a program, the HP-85 adjusts the spacing of statements so that they can be output in their most legible form.

If you should make a mistake while typing the following problems, simply press the (BACK SPACE) key to erase the incorrect character, correct your mistake, and then continue keying in the problem.

To Solve	Press	Display	
5 + 6	5 (+) 6 (END LINE)	`5 + 6`	
		`11`	The result appears below the problem, indented one space.
9 × 8	9 (*) 8 (END LINE)	`9 * 8`	
		`72`	
2^9	2 (^) 9 (END LINE)	`2 ^ 9`	
		`512`	
$\sqrt{7921}$	SQR(7921) (END LINE)	`SQR(7921)`	
		`89`	
Sine of 3.3 radians	SIN(3.3) (END LINE)	`SIN(3.3)`	The system "wakes up" in radians mode. You can change it to degrees mode by typing DEG (END LINE).
		`-.157745694143`	

Arithmetic expressions are typed algebraically—just as you would write them on paper. Functions, like SQR and SIN, must be followed by the "argument" (or number) enclosed within parentheses. A complete list of functions may be found in appendix D. And, as you have seen, you must press (END LINE) to tell the system to solve the problem.

Simple Display Editing

Next, let's make some intentional errors and learn how to correct them. Suppose you wish to type the expression 3+INT(SQR(45*ABS(3-PI/.2))), but by mistake, type the following (don't press (END LINE)):

3+INT(SQR845*ABS

At this point, you realize that the 8 should have been a left parenthesis. The line could be corrected by backspacing and retyping. Let's save some typing time by pressing the (←) *(cursor left)* key. The (←) key enables you to backspace without erasing characters that are already on the display. Press the (←) key until the cursor rests under the 8. Then type (.

Now finish the problem by holding down the (→) *(cursor right)* key, until the cursor is past the S in ABS. Then type:

(3-PI/.2)))

The whole line should appear as:

3+INT(SQR(45*ABS(3-PI/.2)))

When you press (END LINE), the answer will appear (26). Parentheses specify which operations are performed first—more about this in section 3.

As you may have guessed, just as the (←) and (→) move the cursor back and forth on the display, the (↑) *(cursor up)* and (↓) *(cursor down)* keys move the cursor up and down on the CRT display. Thus, you can edit any line on the display (and more, as we shall see later). Finally, the (↖) *(home)* key returns the cursor to the home position on the display.

For example, using the (↑) key, move the cursor up the display so that it rests under the S in SQR(7921) in the problem that you solved above. Now, using the (→) key, move the cursor so that it rests under the 7. Then type 980.

Now the line should read:

SQR(9801)

When you press (END LINE), you will be finding the square root of the new number, 9801. The answer, 99, will appear below the line you edited; the cursor will appear below the answer, ready for another problem.

Note that you did not have to move the cursor past the right parentheses in the problem above before you pressed (END LINE). The cursor may rest anywhere directly under the problem that you wish to enter into the computer. The HP-85 will read the full line, regardless of the cursor placement under the line.

Remember, *what you see* on the display, *is what you get*. If you have extra characters on the same line as you are editing, be sure to clear them before you press (END LINE). You can erase characters to the left of the cursor by pressing (BACK SPACE) and erase characters to the right of the cursor by pressing the space bar or by pressing (-LINE).

Clearing the Display

The (-LINE) *(clear to end of line)* key clears a line from the cursor to the end of the line. The (SHIFT)(CLEAR) keys clear the display and return the cursor to the home position. Typing CLEAR (END LINE) also clears the display.

If you have been following along with the examples, the display should look like this:

```
5 + 6
 11
9 * 8
 72
2 ^ 9
 512
SQR(9801)
 99
SIN(3.3)
-.157745694143
3+INT(SQR(45*ABS(3-PI/.2)))
 26
```

After editing and executing this line ...

... the cursor rests here.

If you press (END LINE) now, with the cursor resting under the S in SIN(3.3), you will be executing the function again. Instead, do the following:

Type	**Display**
5/4	5/4(3.3)

Here, you replaced the letters SIN, with 5/4. Before you can execute the expression you must clear the characters (3.3). You can do this by pressing the space bar until the cursor moves past the right parenthesis. But a faster and easier way to erase the characters is to press (-LINE).

Press	**Display**	
(-LINE)	5/4_	Characters from cursor to end of line are now deleted.
(END LINE)	1.25	Result.

Why don't you press (SHIFT) (CLEAR) now, to clear the display before we continue?

Error Messages and Warnings

If you attempt an improper operation, the HP-85 beeps and displays the word Error or Warning, followed by a number and short description. The error number corresponds to a particular error condition that will help you pinpoint the error. A complete list and description of error messages is provided in appendix E.

There is no need to worry if the HP-85 returns an error or warning message—no keyboard operation is capable of damaging the system. Furthermore, most errors can be simply and easily corrected by editing the line in which the error occurred.

For example, executing the following expression will display an error message:

```
3*(5/7
Error 88 : BAD STMT
```

This expression is not complete because the right parenthesis has been left out. The system cannot interpret the expression so an error message is displayed and the cursor returns to the position in the expression where the system first detected an error—in this example, the asterisk. The cursor returns to the line you have executed when the system interprets an attempt to enter a program statement. (Actually, the system tries to interpret a line first as a program statement and then as an expression. If both attempts fail, the system reports the first error it finds.)

Now you may either edit and correct the line, or clear it by pressing (-LINE), or clear the whole display by pressing (SHIFT)(CLEAR).

Or you can forget about the error and use the arrow keys to position the cursor elsewhere on the display.

With most errors that occur during math calculations, the system displays a warning message and a default value. Then the cursor moves to the beginning of a new line.

For example,

```
5/0
Warning 8 : /ZERO
 9.99999999999E499
_
```

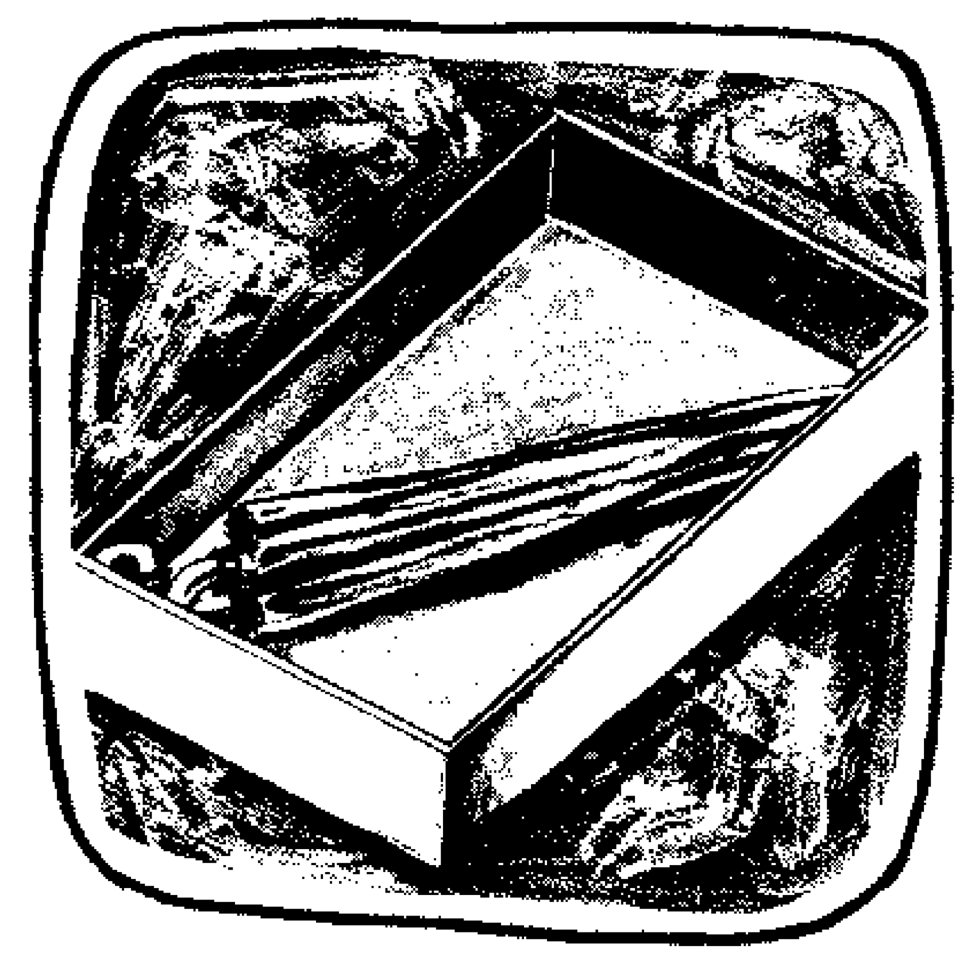
Division by zero causes a warning message and the default value to be displayed.
The cursor moves to the beginning of a new line.

Here, the system alerts you to the error, and then waits in a ready state for you to enter another expression.

Variables

Often it is convenient to assign values to letters and then use these letters in expressions. In programs, a letter can have its value continually updated or changed—hence the term "variable." But you can also perform variable arithmetic straight from the keyboard.

Example: Suppose you receive a telegram from your archaeologist friend, Arthur I. Factus, in South America. He's soon joining an expedition through the rain forest and writes, "PLEASE SEND UMBRELLA IMMEDIATELY." You find a shallow rectangular box, 24 inches wide by 32 inches long. What is the maximum length of an umbrella that will fit inside the box?



You can easily determine the diagonal length of the box, using the Pythagorean theorem, $d = \sqrt{l^2 + w^2}$, where d is the diagonal, l is the length of the box, and w is the width of the box.

One way to solve the problem is to type the following:

```
SQR(24^2+32^2)
```

Here, you substituted the dimensions of the box for the variables in the formula. When you press (END LINE), the answer, `40`, appears on the next line.

Another way to solve the problem is to assign the dimensions of the box to variable names, type in the formula, and let the HP-85 do the substituting. A variable name can be either a letter of the alphabet or a letter followed by a number 0 through 9.

First, ensure that the paper roll has been properly installed in the system, (refer to appendix B) and then type:

`PRINT ALL` (END LINE)

Note: To conserve power, the display turns off when the printer prints.

Then:

Press	Display		Printer
W = 24 (END LINE)	W = 24	Assigns width of box to W.	W = 24
L = 32 (END LINE)	L = 32	Assigns length of box to L.	L = 32
D=SQR(W∧2+L∧2) (END LINE)	D=SQR(W^2+L^2)	Evaluates expression and assigns a value to D.	D=SQR(W^2+L^2)
D (END LINE)	D 40	Fetches the value of D and displays it.	D 40

You can assign a numeric value or the result of an expression to a variable name, as shown above. Whenever you wish to recall the value of an assigned variable, type the variable name and press (END LINE). (Although it may be extra work for this problem, variables are extremely useful in programs in which the values of variables are always changing.)

And you can see the printer has preserved a record of your calculation. Press the paper advance, located in the upper right-hand corner of the keyboard, and save this printout. You are going to use it to write a BASIC program for the HP-85. But first let's look at a prerecorded program—one of the 15 that are included with the Standard Pac shipped with your computer.

With the system in print all mode, you will have a printed copy of everything that you type and that the computer displays. If you wish to cancel print all mode, type:

NORMAL (END LINE)

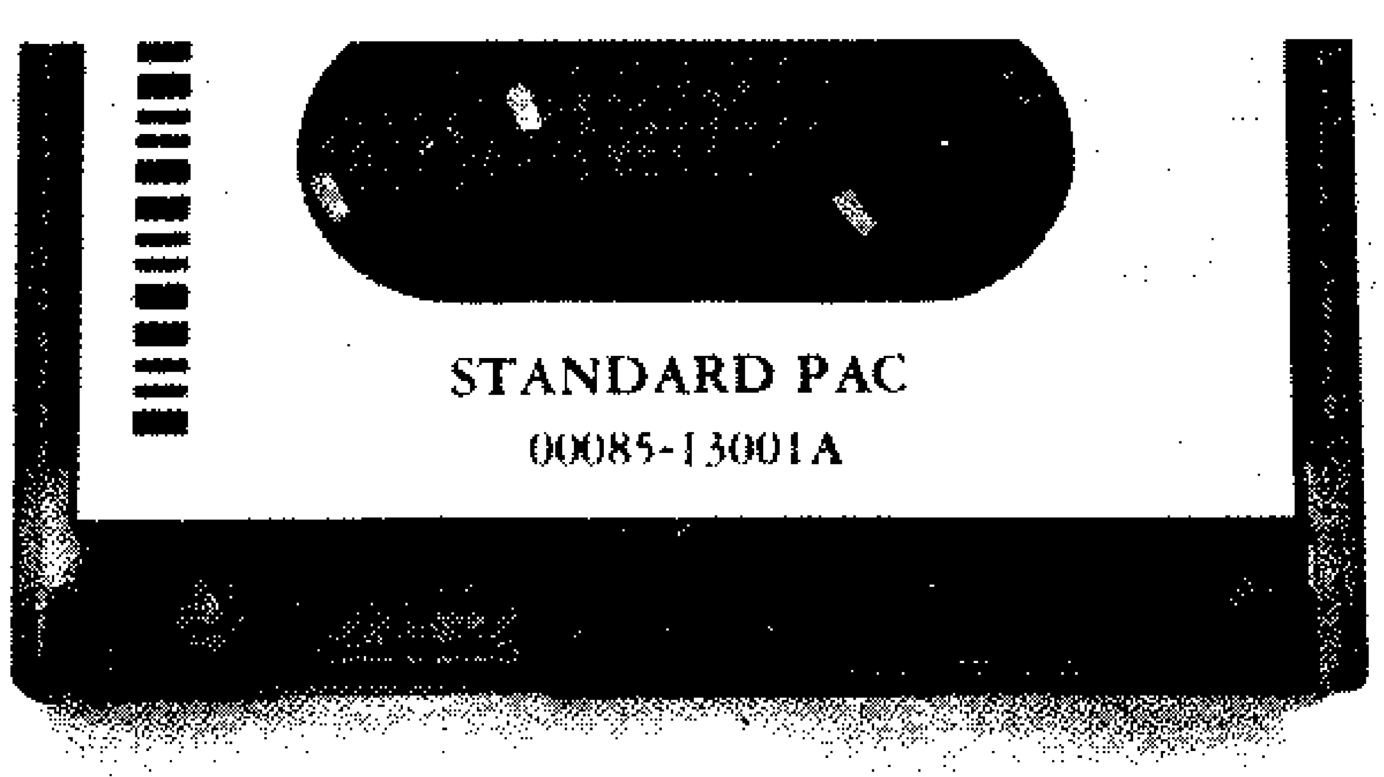
The NORMAL command returns the system from print all mode to normal display mode.

Running a Prerecorded Program

The Standard Pac magnetic tape cartridge shipped with your HP-85 contains 15 prerecorded programs. By using programs from the Standard Pac (or from any of the optional application pacs available in areas like finance, statistics, mathematics, engineering, linear programming, beginning BASIC programming, ...) you can use your HP-85 to perform extremely complex computations just by following the directions in each pac. Let's try running one of these programs now.

Loading a Program From the Standard Pac

1. Before you insert the Standard Pac tape cartridge, make sure that the RECORD slide tab is in the left-most position (as shown). This will protect your tape, so that no other programs can be accidently recorded on the tape.



When the RECORD slide tab is in the left-most position (the opposite direction of the arrow), nothing can be recorded on the tape; your tape is protected.

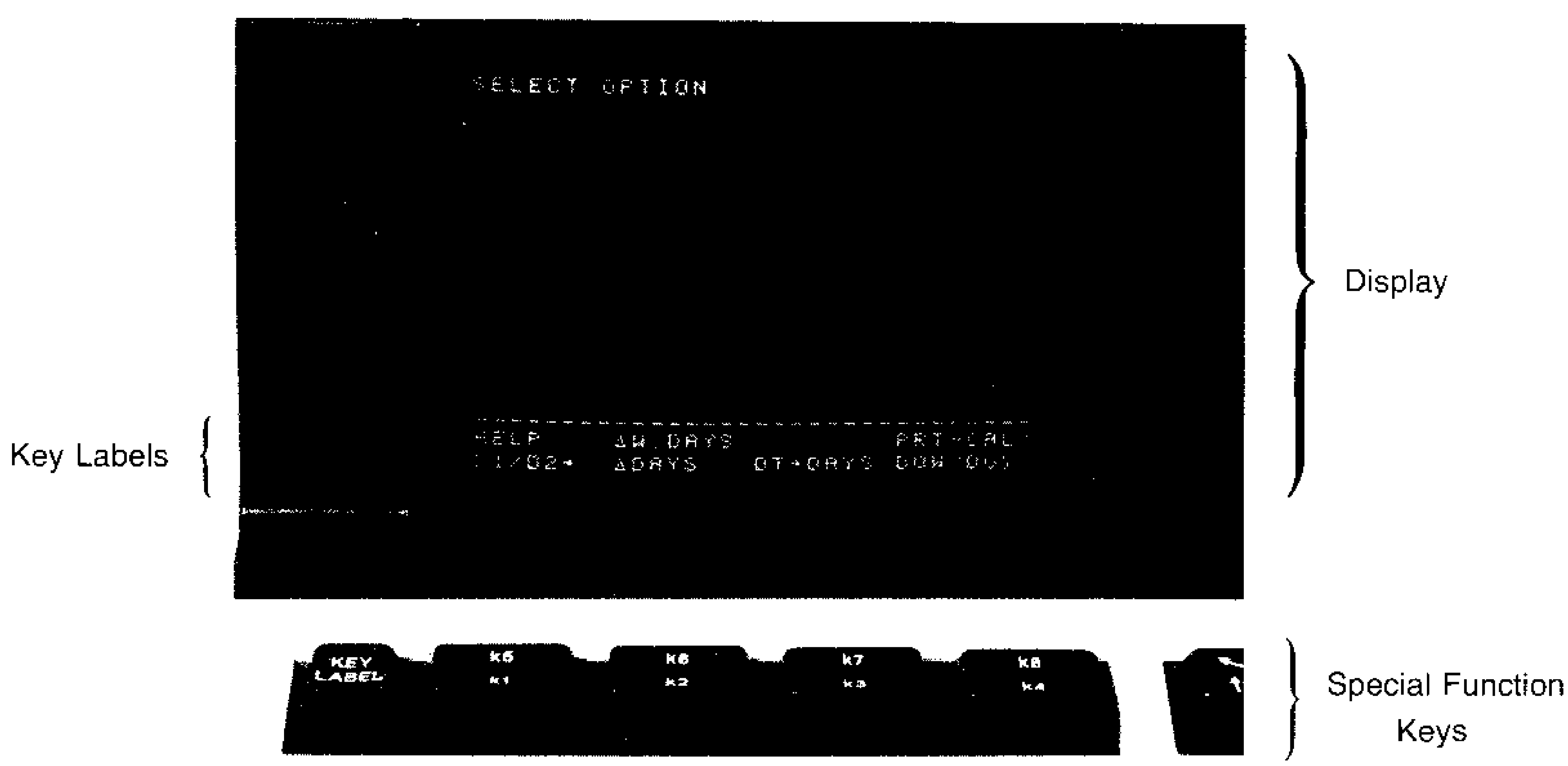
2. Insert the tape cartridge so that its label is up and the open edge is toward the computer. The tape drive door will open when the cartridge is pressed against it; the cartridge can then be inserted. (To remove the tape cartridge, you must press the eject bar. If it is pulled out without pressing the eject bar, another cartridge cannot be inserted until the eject bar is pressed.)

3. To load the Calendar Functions program, type:

   LOAD "CALEND" END LINE

   The LOAD command instructs the computer to find the specified program on the tape cartridge and then load it into computer memory. The CRT screen will blank out while the HP-85 is searching for and loading the program. And an amber light, located to the left of the eject bar, will glow while the cartridge is being used to let you know that the system's attention has been transferred to the tape drive.

4. When the cursor returns to the display and the amber tape drive light goes out, press the RUN key to start the program. After you press RUN, the following should appear on the display:



Many of the programs in the Standard Pac use the special function keys. This is how they work. The bottom two lines of the display correspond directly with the special function keys on the keyboard. The bottom line of the screen displays the labels for the unshifted keys, k1 through k4; the line above it refers to the shifted keys, k5 through k8.

The key labels will remain on the display until they are over-written by characters that you type or that the HP-85 displays. In any case, you can always display the current labels on the screen by pressing KEY LABEL.

With SELECT OPTION and the key labels displayed, you are ready to use the program.

Example: How many days are there between November 25, 1945, and July 25, 1954?

Solution: Since this may be the first time you've used the Calendar Functions program, let's ask for help.

For a more detailed explanation of the key functions, press SHIFT k5.

```
        CALENDAR FUNCTIONS
K1:TWO DATE ENTRY FOR #DAYS/WEEK
   DAYS BETWEEN DATES(K2 AND K6)
K2:NUMBER OF DAYS BETWEEN D1,D2
K3:COMPUTE DATE N-DAYS BEFORE OR
   AFTER ENTERED DATE.
K4:COMPUTE DAY-OF-WEEK AND DAY-
   OF-YEAR OF ENTERED DATE.
K5:HELP
K6:NUMBER OF WEEKDAYS BETWEEN
   D1 AND D2.
K8:GENERATE CALENDAR FOR MO.&YR.
_
--------------------------------
HELP     ΔW.DAYS          PRT-CAL
D1/D2→   ΔDAYS    DT→DAYS DOW/DOY
```

Use k1 (D1/D2→) to enter the two dates.
Then use k2 (Δ DAYS) to find the number of days between the dates.

k7 is not used in the Calendar program.

If you wish to have a printed copy of the display as you see it, simply press SHIFT COPY.

Now let's continue—enter the dates:

Press	**Display**
k1	ENTER FIRST DATE:MM.DDYYYY?

This means you key in the date in the form: month (01 to 12), decimal point, day (01 to 31), year (four digits). Press END LINE after you type the date to enter the data into the program. Thus, to enter November 25, 1945, and July 25, 1954:

Press	**Display**
11.251945 END LINE	11.251945
	ENTER SECOND DATE:MM.DDYYYY?
07.251954 END LINE	07.251954
	DATES ENTERED

Now that the dates have been entered, press [k2] (ΔDAYS) to find the number of days between the dates:

Press	Display
[k2]	NUMBER OF DAYS BETWEEN 11.251945 AND 7.251954 IS 3164 DAYS.

With the HP-85 finding days between dates is that easy! Should you run into difficulties using the Calendar Functions program, refer to the user instructions in the Standard Pac.

Before we leave the calendar program for you to explore on your own, let's use the program to generate a calendar for January 1980 and demonstrate just some of the graphics capabilities of your HP-85. First clear the display.

Press	Display	
[SHIFT] [k8] 1,1980 [END LINE]	MONTH, YEAR=? 1,1980 ENTER HEADING? _	Function key [k8] (PRT-CAL) requests a numeric entry for the month (1-12) and year. Again, enter data into the computer using [END LINE].

Here's your chance to be creative! You can type anything that will fit on one line. If you want your heading to look like ours, type:

```
HAPPY NEW YEAR!!!@%$***@!
```
[END LINE]

Now, watch the HP-85 go to work as it first "draws" the calendar on the video display and then copies it onto the printer. The display will blank out while the printer is in operation. When the printer has finished, you will have a printed copy of the calendar that appears on the display:

HAPPY NEW YEAR!!!@%$***@!
JANUARY 1980

SUN	MON	TUE	WED	THU	FRI	SAT
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

You may want to know the names of the other programs on the Standard Pac tape cartridge. But before you can view the tape directory (catalogue of programs on the tape), you must stop the calendar program from running.

Halting Program Execution

Press the PAUSE key to halt a running program at any time and return system control from the program to the user. (The system beeps when PAUSE is pressed and the program is running.) You may resume the execution of a paused program by pressing CONT *(continue)*.

Now lets take a quick look at the catalogue of programs on the Standard Pac tape cartridge. First press PAUSE to stop the calendar program, then type:

```
PRINT ALL [END LINE]
CAT [END LINE]
```

```
NAME     TYPE   BYTES  RECS  FILE
MOVING   PROG    256    40     1
AMORT    PROG    256    18     2
POLY     PROG    256    29     3
SIMUL    PROG    256    47     4
ROOTS    PROG    256    19     5
CURVE    PROG    256    55     6
FPLOT    PROG    256    22     7
DPLOT    PROG    256    43     8
HISTO    PROG    256    36     9
TEACH    PROG    256    27    10
CALEND   PROG    256    22    11
BIORHY   PROG    256    21    12
TIMER    PROG    256    30    13
COMPZR   PROG    256    56    14
SKI      PROG    256    20    15
MUSIC    DATA    256    44    16
```

Return the system to normal display mode again by typing NORMAL END LINE.

The CAT *(catalog)* command returns the names of the programs on the tape along with some other information about the files. We will discuss the tape filing system in more detail in section 11. For now, just note the names of the programs in the left-most column.

You have seen from the calendar functions example how simple and how much fun it is to use your HP-85. You can run the program again as often as you like. And you can begin using your Standard Pac, or any of the optional application pacs, right *now*. Load any program stored on tape using the LOAD command, followed by the program name in quotes. All you have to do to begin taking advantage of the computing power and programmability of the HP-85 is follow simple instructions like these.

Writing Your Own Programs

If you have never written a program, you may possibly feel uneasy about programming. No need to worry! BASIC is easy to use, yet enables you to perform many complex operations.

BASIC makes use of statements that resemble English. Once a statement is explained, its function is easy to remember. A BASIC program is made up of numbered statements which direct the system to perform certain tasks.

Earlier, you calculated the diagonal of one side of a rectangular box, and you may have saved the printed copy with the values and formula for the problem. Now, if you want to calculate the diagonal of several rectangles (or the hypotenuse of any right triangle), you could repeat the procedure, using different values for the dimensions of the sides. Or you could change the values of the variables using the editing capabilities of the HP-85.

The easiest and fastest method, however, is to create a BASIC program that will compute the diagonal of any rectangle.

Creating the Program

Essentially, you have already created it. When you write a program, you must ask yourself the following questions:

1. What answer(s) do I want?
2. What information do I know?
3. What method will I use to find the solution from what I know?
4. How can the HP-85 help me solve the problem?

We want to find the diagonal of any rectangle. We know that we can use the Pythagorean theorem to compute the diagonal given the lengths of the sides of the rectangle. Thus, we know that we must assign values to two variables and then compute the result using the given formula. We'll answer the other two questions below (and discuss the details of BASIC programming in part II of this handbook).

For now, notice that each statement begins with a number and the last statement of a program is END. (You may wish to clear the display before you key in the program; press (SHIFT) (CLEAR).)

Entering the Program

To enter the program into the system:

1. Press (SHIFT) (SCRATCH) and then press (END LINE) to clear the computer and erase previous programs from computer memory.
2. Type the following program exactly as shown (including the statement numbers), pressing (END LINE) after each statement.

`10 DISP "ENTER SIDE LENGTHS"` (END LINE)	Statements 10 through 40 display
`20 DISP "OF A RIGHT TRIANGLE,"` (END LINE)	the quoted text on the CRT screen.
`30 DISP "SEPARATED BY A COMMA."` (END LINE)	
`40 DISP "THEN PRESS END LINE."` (END LINE)	
`50 BEEP` (END LINE)	Audio, as well as visual, prompt!
`60 INPUT L,W` (END LINE)	Enables you to assign values to L and W from the keyboard.
`70 D = SQR(L ^ 2 + W ^ 2)` (END LINE)	Computes the hypotenuse.
`80 PRINT "HYPOTENUSE =";D` (END LINE)	Prints quoted message and value of D.
`90 END` (END LINE)	Marks end of program.

Running the Program

To run the program, simply press the (RUN) key. Find the length of the hypotenuse of a right triangle with sides 7.5 inches and 10 inches.

Press	**Display**	
(RUN)	`ENTER SIDE LENGTHS`	
	`OF A RIGHT TRIANGLE`	
	`SEPARATED BY A COMMA`	
	`THEN PRESS END LINE.`	
	`?`	Beep!
7.5,10 (END LINE)	`7.5, 10`	

Now the HP-85 will print the result:

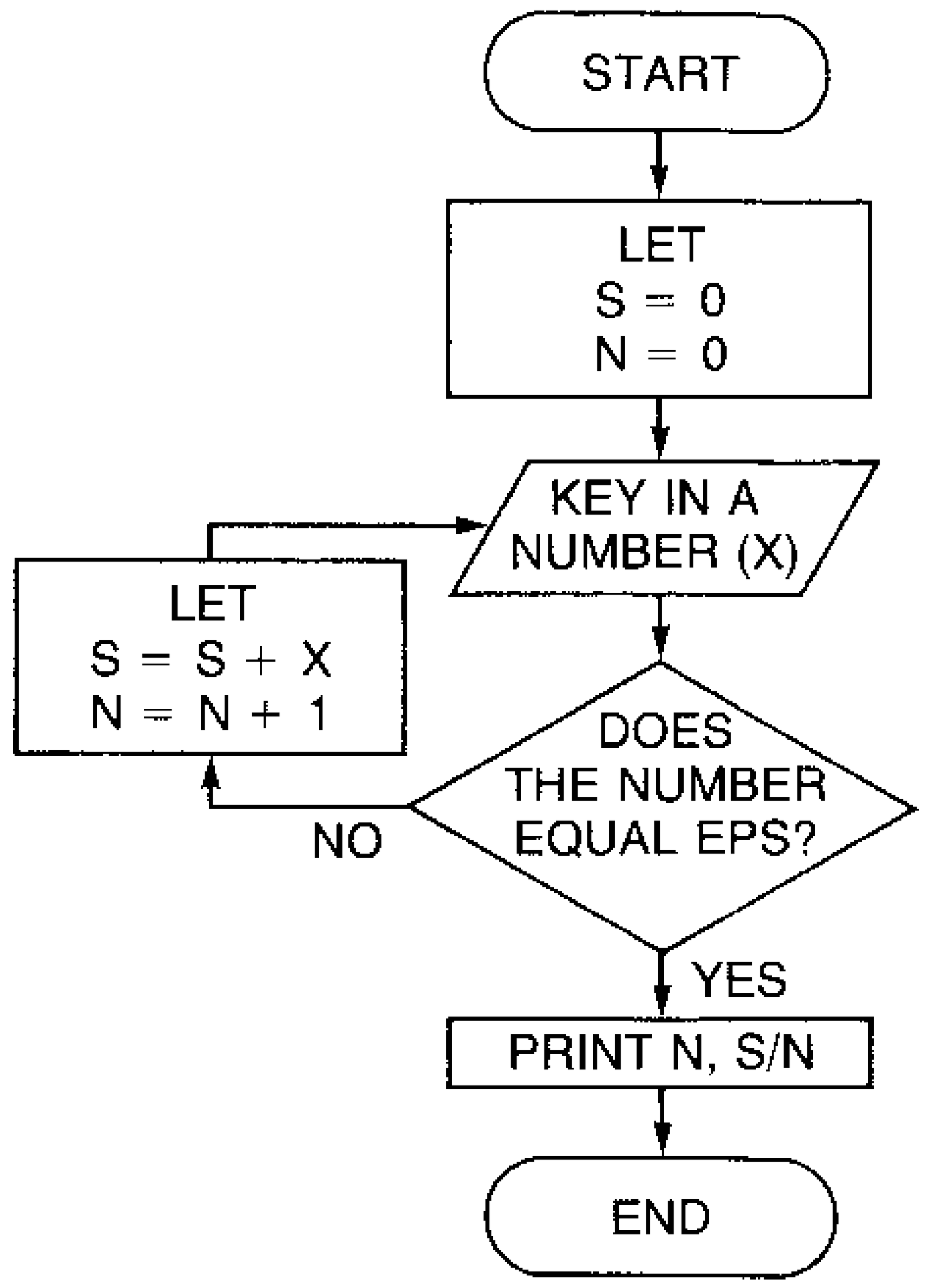
```
HYPOTENUSE = 12.5
```

You can run the program as many times as you like, simply by pressing the RUN key.

An Averaging Program

Since you may not be sending umbrellas to South America in the near future, or calculating the hypotenuses of right triangles or the diagonals of rectangles, let's write a program that may be of more use to you, and then record it on a tape cartridge.

This flowchart outlines the steps in a program that enable you to enter a set of numbers and then find their average.



First, we initialize the variables we will use. S will be the sum of the numbers, N determines how many numbers are being averaged, and X represents each new number.

When you key in EPS (which stands for epsilon, the smallest number you can obtain on the HP-85), the program stops asking for new numbers and prints the average of the numbers you have keyed in.

Before you key in the following program, be sure to press SHIFT SCRATCH END LINE to erase the previous program.

And let's use the AUTO key to provide us with statement numbers automatically, so that we don't have to type them ourselves. Simply press SHIFT AUTO and the system will display:

```
AUTO_
```

Then press END LINE and the system will display 10 and wait for you to enter a program statement. After you enter the statement by pressing END LINE, the system will display 20 and wait for another statement to be entered. The AUTO command numbers statements beginning with 10 and in increments of 10. (You can also change the starting number and increment value with the AUTO command, as we shall see later.) Stop auto line numbering by backspacing over the unwanted statement numbers and typing NORMAL END LINE.

Now enter the averaging program below. From now on, we won't be showing the (END LINE) key with the program listing, but, **it must be pressed after each statement.** When the system displays the statement number, enter the rest of the statement and press (END LINE).

`AUTO`	Press (SHIFT) (AUTO) (END LINE) for automatic line numbering.
`10 REM *AVERAGE*`	Remark.
`20 S=0`	Initialize variable S.
`30 N=0`	Initialize variable N.
`40 DISP "ENTER THE NUMBERS."`	Display quoted message.
`50 DISP "ENTER 'EPS' TO END"`	
`60 DISP "THE PROGRAM."`	
`70 INPUT X`	Assign a value to X from the keyboard.
`80 IF X = EPS THEN 120`	Check X to see if it is EPS.
`90 S = S+X`	Add X to sum.
`100 N = N+1`	Add 1 to counter.
`110 GOTO 70`	Go back to line 70 to enter a new number.
`120 DISP "THE AVERAGE OF ";`	Display result.
`130 DISP "THE";N;"NUMBERS ";`	
`140 DISP "IS";S/N`	
`150 END`	Marks end of program.
`160 _`	Now backspace over `160` and type `NORMAL` (END LINE) to stop auto line numbering.

Let's take an example to test the program.

Example: What is the average of the distances in light-years of the five brightest stars (aside from the sun) seen from the earth?

Star	Distance (light-years)
Sirius	8.7
Canopus	100
Alpha Centauri (Rigil Kentaurus)	4.4
Arcturus	36
Vega	26.5

Press	**Display**

(RUN)	`ENTER THE NUMBERS.`
	`ENTER 'EPS' TO END`
	`THE PROGRAM.`
	`?`

8.7 (END LINE)	`8.7` `?`
100 (END LINE)	`100` `?`
4.4 (END LINE)	`4.4` `?`
36 (END LINE)	`36` `?`
26.5 (END LINE)	`26.5` `?`

EPS (END LINE)	`EPS`
	`THE AVERAGE OF THE 5 NUMBERS IS`
	`35.12`

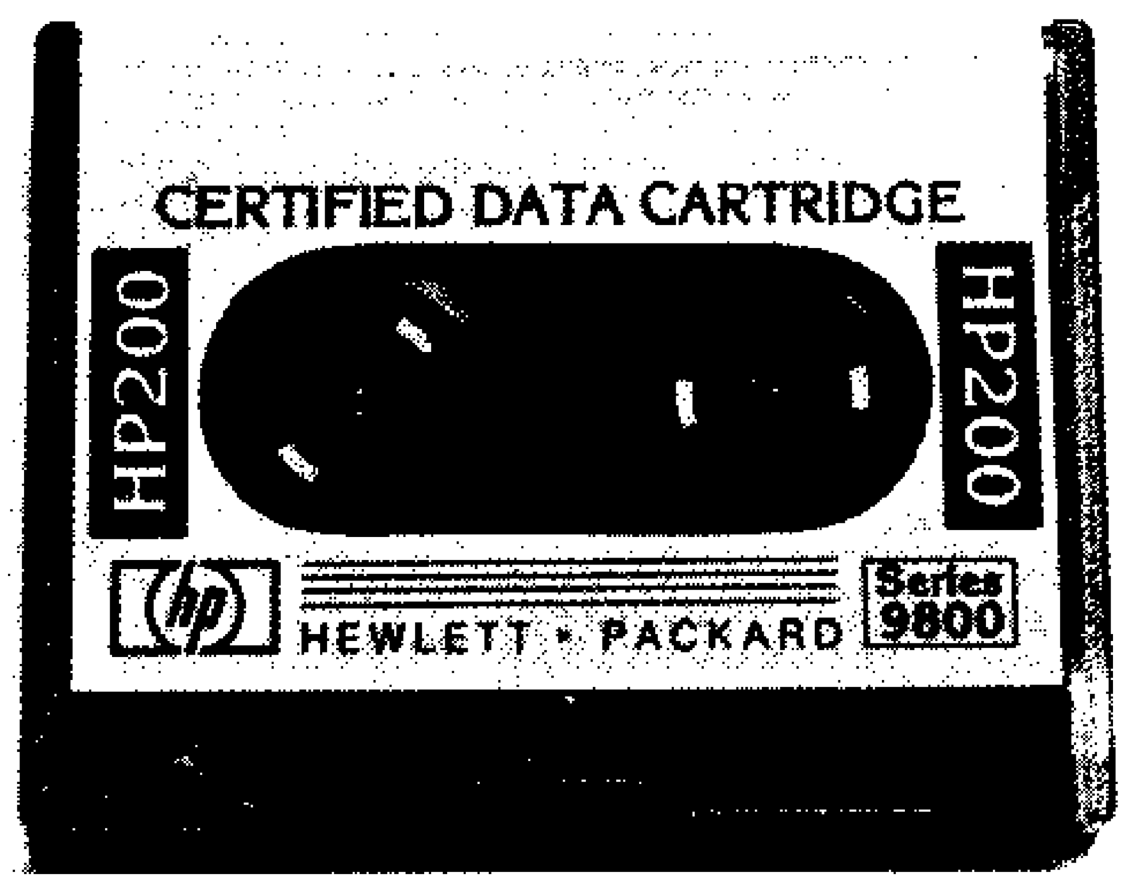
Recording the Program

Just as the programs in the Standard Pac have been recorded on a magnetic tape cartridge, you also can record your own programs on a cartridge.

To Record Your Program:

1. Select a blank magnetic tape cartridge. Use only HP Data Cartridges with your HP-85.

2. Check to see that the RECORD slide tab is in the right-most position, in the direction of the arrow (as shown). When the RECORD slide tab is in the left-most position, your cartridge is protected; i.e., you cannot record anything on it or delete from it.



   When the RECORD slide tab is in the right-most position, you can record programs on the tape or delete existing programs from the tape.

3. Insert the cartridge so that its label is up and the open edge is toward the computer. (If the Standard Pac is still in the tape drive, take it out by pressing the eject bar.)

Note: The ERASETAPE command should only be performed the first time you use a new tape or when you wish to erase all of the existing programs on a tape.

4. Type ERASETAPE (END LINE). This command erases the tape and in the process, sets up a "directory" so that your programs can be filed.

5. Now, decide what you want to name your program. Pick something that will remind you of the program—a name no longer than six characters. However, any combination of characters may be used, except quotation marks. Then press the (STORE) key and type the name of your program enclosed within quotation marks. If you want your program to be stored like ours, type:

   (STORE) "AVERAG" (END LINE)

When you press (END LINE), the HP-85 records your averaging program on the magnetic tape cartridge in its own "file." Again, notice that the CRT display is turned off to conserve power while the cartridge is accessed.

That's all there is to it! To record future programs on the same tape cartridge, you simply follow step 5 again. Future programs will automatically be stored in separate files.

You can verify that your program has been stored by executing the CAT command as we did earlier. The information displayed will be discussed later.

Erasing a Program From the Tape Cartridge

An averaging program may be of no use to you, so before we go on to the next section we'll tell you how to erase a specific program file using the PURGE command. First, make sure that the RECORD slide tab is in the right-most position. Then:

1. Type PURGE.
2. Type the name of the program or file you wish to delete from the cartridge, enclosed within quotation marks.
3. Press (END LINE) to purge the specified file.

If "AVERAG" was the name you used for the averaging program, simply type:

PURGE "AVERAG" (END LINE)

Your program or file will be erased, ready for something else to be stored there. You can store many long programs on one tape, so you don't have to purge little-used programs constantly.

Keyboard, Printer, and Display Control

Now that you've had a chance to familiarize yourself with the HP-85, let's look at some of its features in greater detail.

The Keyboard

As you've noticed, the keyboard is divided into the following areas:

- Typewriter Keyboard
- Numeric Keypad
- Special Function Keys
- Display Control and System Command Keys

Some of the features in each area were discussed in section 1. The rest of the display editing features will be discussed in this section. The remaining keys are helpful in a variety of ways—as typing aids, in running programs, using the printer, and recording programs on tape. The keys are described, in appropriate places, throughout this manual. Refer to the HP-85 key index on pages 12 and 13.

Typewriter Keys

The alphanumeric keys operate much like those on a standard typewriter keyboard. If, for instance, you want to display the dollar sign, $, you must hold down the (SHIFT) key while you press ($ 4). You must also use the (SHIFT) key to select any command or symbol on the upper half of a key. But we won't be showing the (SHIFT) key in the keystroke sequences in this handbook.

If the command is a shifted operation, it will appear in the upper half of the key. For instance, when you see (CLEAR), it means you must hold the (SHIFT) key down while you press (CLEAR -LINE).

The HP-85 keyboard differs from a standard typewriter in two major ways:

- Unshifted letters appear as capital letters on the display (unless you use the FLIP command, as we'll see in a moment).
- All of the keys repeat automatically if you continue to hold them down.

BASIC Typewriter Mode

Unshifted letters initially appear as capitals on the display because the standard BASIC language requires its "keywords" (like PRINT, GOTO, IF... THEN, etc.) to be in capital letters.

In BASIC mode you can select small letters by using the (SHIFT) or (CAPS LOCK) keys with the alphabetic keys.

Thus, when you press (A), a capital "A" appears on the display; when you press (A) while holding down the (SHIFT) key, a small "a" appears on the display.

The CAPS LOCK key operates like the CAPS LOCK on a standard typewriter except that if the key is pressed and locked in BASIC mode, alphabetics appear as small letters. Once the CAPS LOCK key is pressed, it remains locked until you press it again. Note that *only* the 26 letters of the alphabet are affected by the CAPS LOCK key.

Normal Typewriting Mode

If you wish to type in "normal typewriting mode" where unshifted letters produce small letters and shifted letters produce capital letters, use the system command FLIP. Whenever you type FLIP and press END LINE, the unshifted case switches from small letters to capital letters or vice versa.

Programming Note: Even though standard BASIC requires "keywords" to be in capital letters, the HP-85 will interpret keywords and variables that are typed in either uppercase or lowercase letters. Thus, the following program, typed in normal typewriting mode, is legal:

```
10 print "You can use small"
20 print "letters or capital"
30 print "letters in BASIC"
40 print "programs."
50 end
```

As soon as you list the program, the small letters in keywords and variable names are converted to capital letters, but strings (quoted text) and remarks will remain as typed.

HP-85 Character Set

The HP-85 character set consists of 256 characters, 128 of which are directly accessible from the keyboard.

You can see the uppercase letters, punctuation and other typewriter symbols on the face of the keys; and you've seen how lowercase letters can be accessed using SHIFT, CAPS LOCK, or the FLIP command.

Five more characters can be accessed with the SHIFT key. Thirty-two more characters are accessed with the CTRL key. The remaining 128 characters can be accessed with the CHR$ function; they are merely the first 128 characters underscored.

The extra shifted characters are:

`	SHIFT KEY LABEL	
π	SHIFT /	To access these characters, use operators from the numeric keypad only.
÷	SHIFT −	
Σ	SHIFT *	
⊢	SHIFT +	

The CTRL characters are those in the first column of the table of characters in appendix C, page 292. They are generated by holding down the CTRL key and pressing the key that is superscripted by a "c" next to the character in the table.

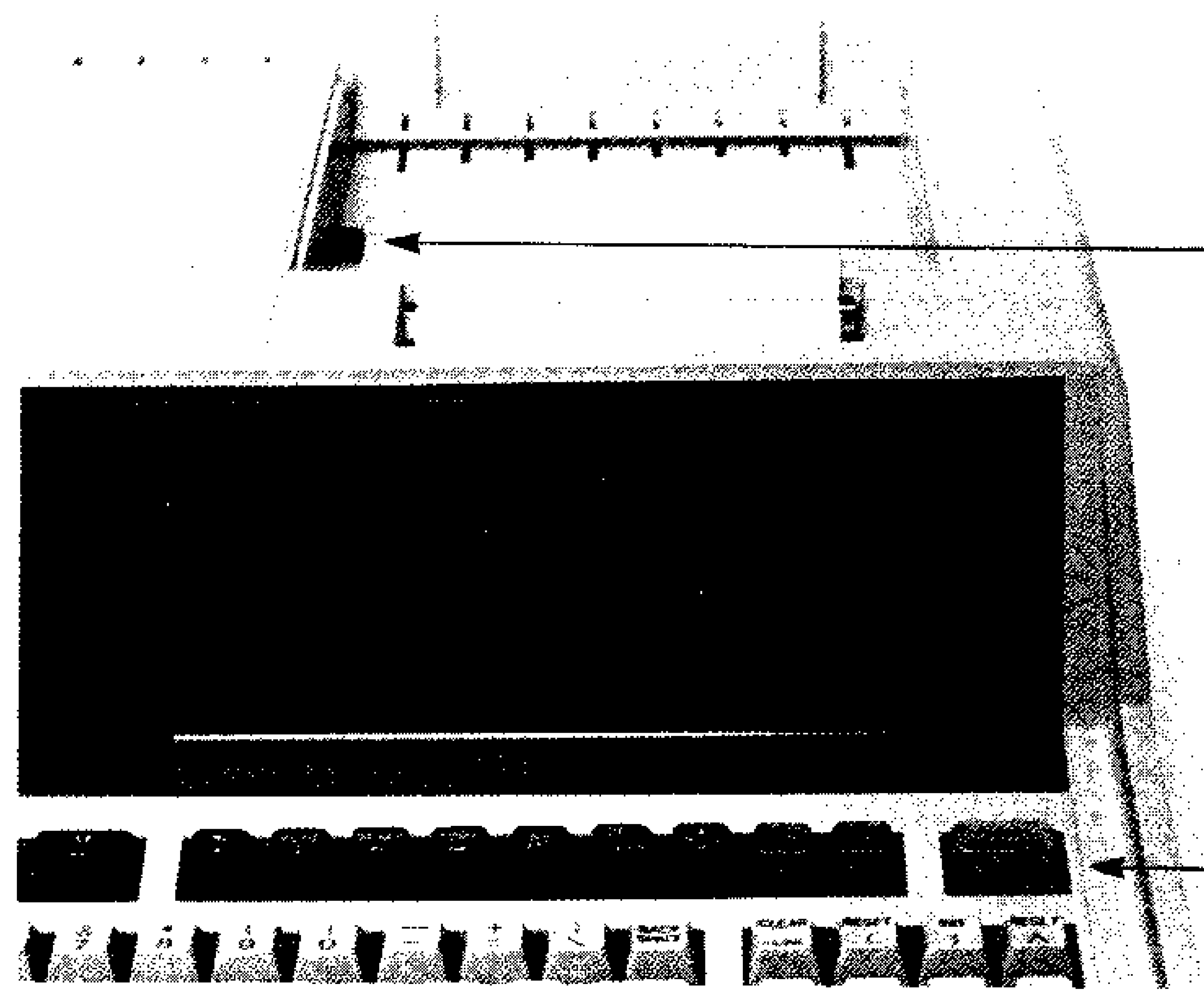
For instance, to generate the Greek letter Δ, hold down the CTRL key and press H (H^c).

And, since @ is a shifted symbol, generate the character ä by holding down the CTRL key and SHIFT key and pressing @ 2 ($@^c$).

Each of the characters is assigned a decimal code, from 0 through 255. These codes are useful in advanced programming. We'll discuss character codes in section 8, Using Variables.

Printer Control

The HP-85's built-in thermal printer prints 32 characters per line.



Adjust the intensity of printed characters by rotating the printer intensity dial, located to the left of the paper roll. The lightest setting is when the dial shows 0, the darkest setting is when the dial shows 7. You can extend the long-term life of the printer by setting the printer intensity dial to 4 or less.

There are several ways to access the printer:

- Pressing the (COPY) key produces a printed copy of whatever is currently displayed on the CRT screen. The (COPY) key can be pressed to copy either the alphabetics or graphics on the display. You can also copy the alpha screen by *typing:*

  `COPY` (END LINE)

- Executing the `PRINT ALL` command sets the HP-85 to print all mode; everything that you enter into the system and every message or result that the system displays will be recorded by the printer.

  `PRINT ALL` (END LINE)

  Return to normal display mode by typing:

  `NORMAL` (END LINE)

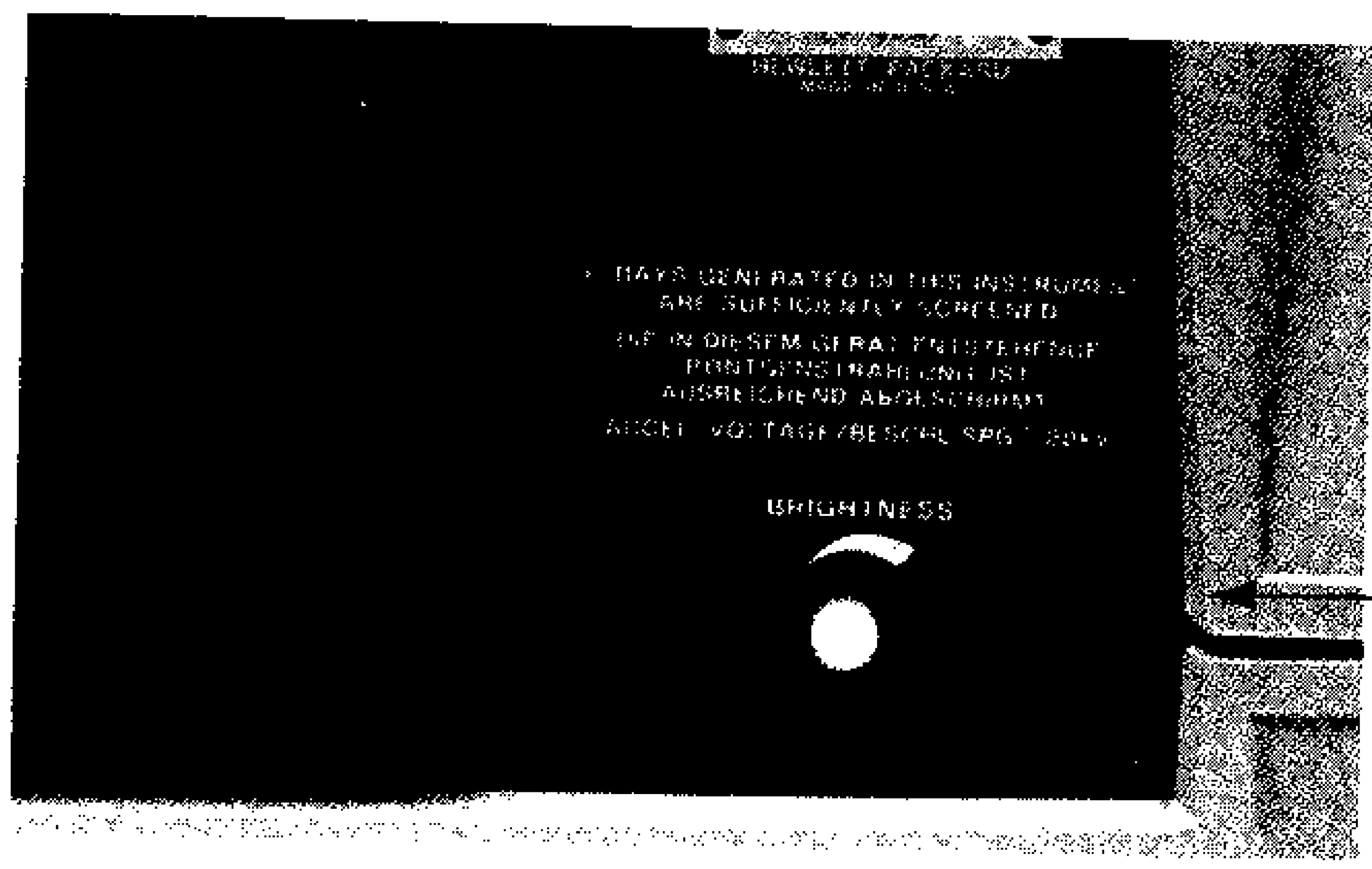
- And, of course, whenever you execute the `PRINT` statement, either manually from the keyboard or in a program, the `PRINT` message will be output to the printer.

To advance the printer paper, press the (PAPER ADV) key, located in the upper right corner of the keyboard. To advance the paper more than one line, simply hold the (PAPER ADV) key down until the paper has advanced the desired amount. To replace the paper roll, refer to appendix B.

The Display

The CRT (cathode ray tube) display consists of a 32-character by 16-line display screen and is the primary means of editing programs, and of viewing data, keyboard entries, program listings, error messages, system comments, and results.

You can increase the intensity of characters on the display by rotating the display intensity knob in the direction of increasing width of the brightness symbol.



Display Intensity Knob

You can display a maximum of 16 lines at any one time, but you actually have immediate access to four full screens worth (64 lines) of information.

The (ROLL) key is used to recall information that has "rolled" out of view. There are three full screens of past history, plus the current screen, available for rolling up or down. You'll appreciate the (ROLL) key when you are writing, reviewing, or listing lengthy programs.

When you hold down the (ROLL) key, information in the display will "roll down" to reveal the lines most recently lost.

When you press (SHIFT) (ROLL), information in the display will "roll up" to reveal either the oldest lines (if no previous rolling has been done) or lines that have been rolled down (if some previous rolling has been done).

Entering Long Expressions

Suppose you wish to solve a lengthy numeric expression like:

$$\sqrt{5\left[\left(\left\{\left[\left(1+0.2\left[\frac{350}{661.5}\right]^{2}\right)^{3.5}-1\right]\left[1-(6.875\times 10^{-6})\,25{,}500\right]^{-5.2656}\right\}+1\right)^{0.286}-1\right]}$$

Do you have to break the expression into parts and solve one line's worth of the problem at a time?

No! An expression can contain as many as 95 characters (including spaces) or three full lines of the display minus one character position for (END LINE).

Before we attempt to evaluate the long expression, press one of the character keys, such as the [*] key, and continue to hold it down until it repeats across the display.

```
********************************
********************************
********************************
********************************
```

As long as you hold down the [*] key, row after row of asterisks will be repeated across the display. There is no need to press [END LINE] at the end of the line on the display; when the cursor reaches the end of a line, typing another character automatically sends it to the beginning of the next line.

Now press [CLEAR] to clear the display. As you type in the following expression, notice that when the cursor is at the end of a line, typing automatically sends the cursor to the next line. Don't press [END LINE] until you have keyed in the entire expression.

`SQR(5*(((((1+.2*(350/661.5)^2)^3` `.5-1)*(1-6.875E-6*25500)^-5.2656`	Typing merely continues on the next line.
`)+1)^.286-1))` [END LINE]	Now press [END LINE] to execute this expression.
`.835724535179`	The answer.

The 95-character maximum length of an expression also applies to program statements (including line numbers). For instance, in the Pythagorean theorem program in section 1 we typed:

```
10 DISP "ENTER SIDE LENGTHS" [END LINE]
20 DISP "OF A RIGHT TRIANGLE," [END LINE]
30 DISP "SEPARATED BY A COMMA." [END LINE]
40 DISP "THEN PRESS ENDLINE." [END LINE]
```

But we could have entered the display message in one statement, like this:

```
10 DISP "ENTER SIDE LENGTHS OF A
 RIGHT TRIANGLE, SEPARATED BY A
COMMA. THEN PRESS ENDLINE." [END LINE]
```

Again, at the end of a line on the screen, the cursor automatically moves to the beginning of the next line. But you must press [END LINE] to enter the program statement into computer memory. [END LINE] marks the end of an expression or statement *and* positions the cursor at the beginning of a new line.

What happens when you fill the display with characters, or type more than 95 characters in an expression or statement? The HP-85 will allow you to key in four full screens worth of characters as long as you don't press [END LINE].

But, if you try to enter an expression consisting of more than 95 characters by pressing [END LINE], you will probably get odd results. The system will try to interpret the most recently typed three lines of the display, yielding either an error message or interpreting only part of what you keyed in. If you are confused, execute the `PRINT ALL` command and the system will echo exactly what it understood your line to be.

Display Editing

In section 1, we introduced the following display editing features of your HP-85:

←	Cursor Left	These keys merely position the cursor in the display without erasing characters. The vertical and horizontal arrow keys repeat automatically if you continue to hold them down.
→	Cursor Right	
↑	Cursor Up	
↓	Cursor Down	
↖	Home	
CLEAR		Clears the display.
-LINE		Deletes a line from the cursor to the end of a line.
SPACE BAR		The space bar moves the cursor forward one space, or, if held down, repeats automatically. If characters are already present on a line when you press the space bar, they will be replaced with spaces.
BACK SPACE		Erases characters as you backspace. The key repeats when held down continuously.

There are three more important display editing features: SHIFT BACK SPACE *(fast backspace)*, -CHAR *(delete character)*, INS RPL *(insert/replace)*.

Fast Backspace

If you press both the SHIFT key and the BACK SPACE key at the same time, the cursor will *rapidly* backspace, erasing characters at the same time. To protect the user from accidently erasing the whole screeen, SHIFT BACK SPACE moves the cursor back to the beginning of a line, not to the home position of the display. But if you continue to hold down SHIFT BACK SPACE, it will repeat rapidly, erasing the next line above.

Deleting Characters

The -CHAR key enables you to delete a character from the display, without leaving a space in its place. If you hold down the -CHAR key, it repeats automatically.

Example: Type, without pressing END LINE:

```
This line will be depleted._
```

But what we meant, of course, is that soon we will *delete* `This line`. Move the cursor with the ← key, so that it resets under the `p` and press -CHAR once.

Now move the cursor back to the beginning of the sentence, again with the ← key. Then hold down the -CHAR key to delete `This line`. The remainder of the sentence should look like this:

```
will be deleted.
```

And this can be deleted with the stroke of one key. Press -LINE to delete the rest of the sentence.

Example: Change:

```
78 + 36 + 92 + 100 + 91 + 89 + 8
```

to:

```
78 + 36 + 92 + 100 + 8
```

Position the cursor under the plus sign between `100` and `91`, in the first expression, then press -CHAR until `+ 91 + 89` has been deleted. Now press END LINE to get a result of `314`.

Inserting Characters

When there is only one cursor on the display, the computer is in *replace* mode. In other words, when you type characters "on top of" characters that are already in the display, those characters are replaced by the ones you type in.

The (INS RPL) *(insert/replace)* key alternates between *insert* and *replace* mode, allowing you to insert characters in a line that has already been typed. For instance, type the following without pressing (END LINE):

```
COS(3)*4
```

Suppose you really wanted the cosine of 2.3 radians not 3 radians. Move the cursor back under the 3 using the (←) key, press (INS RPL), and type `2.`. Now the display should show:

```
COS(2.3)*4
```

When you pressed (INS RPL), another cursor appeared to the left of the original cursor. The double cursor informs you that the computer is in insert mode, and tells you that the next character typed will be inserted between the two cursors.

Like the single cursor, the double cursor can be positioned anywhere in the display with the arrow keys. But when you press (END LINE) the second cursor will disappear.

After you have inserted the desired characters, press the (INS RPL) key once again to remove the second cursor from the display and return to replace mode.

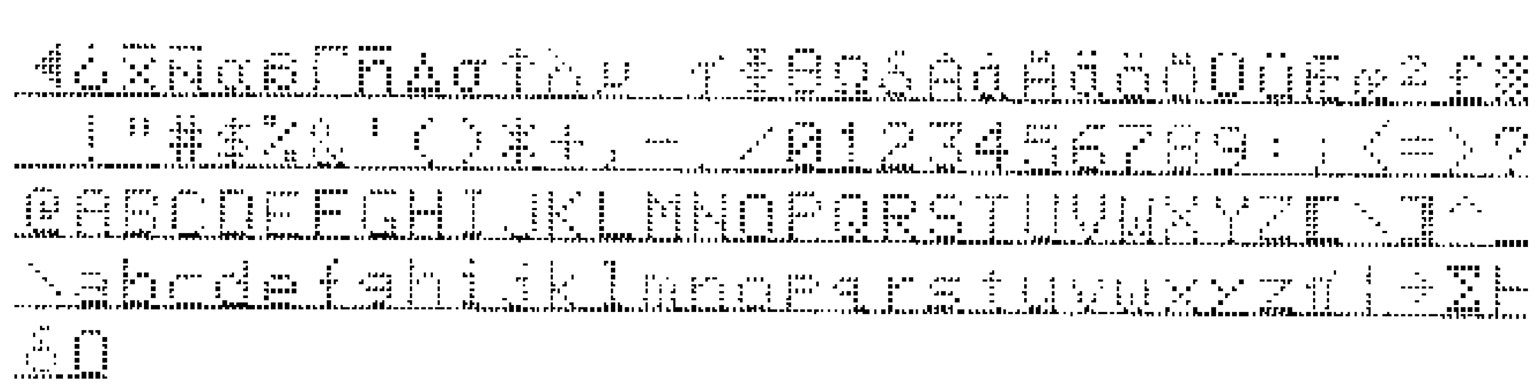
You can insert as many characters as you wish into an expression or a program statement. But make sure that the final expression does not extend beyond three display lines or 95 characters.

If you press (-CHAR) in insert mode, the character above the right cursor will be deleted and the system will return to replace mode automatically.

System Self-Test

Should you feel that the HP-85 is malfunctioning, press the (TEST) key. This causes the system to run through an electronic check of all internal components.

If everything is working properly, the HP-85 system displays and prints the following characters at the end of the test:



These two characters will vary depending on the contents of computer memory.

The graphics display will be cleared, but programs and/or variables in computer memory will remain intact.

If the system is not operating properly, it will display:

```
Error 23 : SELF TEST
```

This tells you that a problem exists in the computer's circuitry; contact the nearest HP dealer or HP sales and service office immediately for system repair.

Resetting the Computer

If the computer becomes inoperative due to a system or input/output malfunction, it may need to be reset. The computer is reset and returned to a ready state by pressing (RESET) while holding down (SHIFT).

Resetting the computer immediately aborts all system activity. The reset operation returns the computer, as well as some peripherals and interfaces, to a ready state. The reset operation is useful when you want to return the system's components to a known configuration before loading or running a program. In other words, (RESET) sets the trigonometric mode, data pointers, graphics scale and pen, timers, output devices, print all mode, etc., to the same default state as when the system was switched on. If a program is running, any pending or executing input/output operation is terminated and information may be lost.

Refer to the Reset table in appendix C for a list of conditions affected by (RESET).

Expressions and Keyboard Operations

In this section, we will discuss ''expressions'' and some of the components of expressions, as well as related keyboard operations. An **expression** is any logical combination of numbers, characters, variables, operators, or functions.

The section's topics include:

- Arithmetic operators.
- Number ranges and number formats.
- Simple numeric and string variables.
- Relational and logical operators.
- Time functions.

The math functions will be discussed in section 4.

So that you'll be familiar with operators, variables, and functions when we use them later in program statements, we'll discuss them in ''calculator'' mode (from the keyboard, not in programs) now.

Keyboard Arithmetic

You have already become familiar with the numeric keypad. Numeric entry is easy on the HP-85. The HP-85 requires only that you press (END LINE) after the expression is typed, in order to obtain the result.

The arithmetic operations that can be performed on the system are:

- Addition (+)
- Subtraction (−)
- Multiplication (*)
- Division (/)
- Exponentiation (^)
- Integer division (\ or DIV)
- Modulo (MOD)

To perform an arithmetic operation:

1. First key in the expression. (Either the numeric keypad or the typewriter keyboard may be used to type numbers.)
2. Then press (END LINE) to execute the expression.

The result will appear under the line you executed.

For example, multiply 8 by 3:

Press	Display	
8 [*] 3	8*3	
[END LINE]	24	

To raise a number to power, such as 8^3:

Press	Display	
8 [^] 3	8^3	
[END LINE]	512	

You do not need to use parentheses to raise a number to a negative power. For instance, compute 8^{-3}:

Press	Display	
8 [^] -3	8^-3	
[END LINE]	.001953125	Result.

MOD and DIV

In addition to the usual arithmetic operators, +, -, *, /, and ^, there are two more arithmetic operators that may prove useful to you. These operators are DIV *(integer division)* and MOD *(modulo)*. They are used just like the other five operators.

Integer division (DIV or \) returns the integer portion of the quotient. In other words, normal division takes place, but all digits to the right of the decimal point are truncated (not rounded) so that you only have the whole number result. Integer division can be specified either by keying in DIV or by using the symbol \ for the operator. For example:

Press	Display	
16 DIV 5	16 DIV 5	Key in the expression.
[END LINE]	3	Then press [END LINE].
5 DIV 16	5 DIV 16	
[END LINE]	0	
5 [\] 16	5 \ 16	
[END LINE]	0	

Given two values A and B, A DIV B= IP(A/B); in other words, the "integer part" of A divided by B.

The MOD (modulo) operator returns the remainder resulting from a division. Like DIV, a normal division occurs, but instead of taking the whole number result as DIV does, MOD takes the remainder and returns it as the result. For instance, when you divide 7 by 3, the division result is 2 with a remainder of 1. MOD would return the 1 as the result of its operation, while DIV would return the 2. For example:

Press	Display	
16 MOD 5 [END LINE]	16 MOD 5	$16 = 3*5 + 1$
	1	
-8 MOD 3 [END LINE]	-8 MOD 3	$-8 = -[(2 * 3) + 2]$
	-2	
(-8)MOD 3 [END LINE]	(-8)MOD 3	$-8 = (-3)*3+1$
	1	

Given two values A and B, `A MOD B = A-B*INT(A/B)`; in other words, A minus B times the greatest integer less than or equal to the quotient of A divided by B. AMOD0 is A, by definition. From the definition, it turns out that $0 \leq AMODB < B$ if $B > 0$ and $B < AMODB \leq 0$ if $B < 0$.

Despite the fact that `DIV` *can* be spelled out, and `MOD` *must* be spelled out since it has no special symbol, they are still *operators* and are used just like the other five operators are used.

Arithmetic Hierarchy

When an expression has more than one arithmetic operation, the order in which the operations take place depends on the following hierarchy:

`^`	Exponentiation.	Performed first.
`MOD`, `DIV` or `\`, `*`, `/`	Modulo, integer division, multiplication, and division.	↓
`+`, `-`	Addition and subtraction.	Performed last.

When an arithmetic expression contains two or more symbols at the same level in the hierarchy, the order of execution is from left to right.

So an arithmetic expression such as 1+3*2 is equal to 7. The computer performs the multiplication before the addition because of its hierarchy. What if, instead of computing 1+3*2, you really wanted 1+3 and the result times 2? Use parentheses.

Parentheses

The prescribed order of execution can be altered if you use parentheses. Using the example of 1+3 and then multiplying the result times 2, you would type:

`(1+3)*2` [END LINE] The answer, 8, is returned.

Note that only rounded, `( )`, parentheses may be used in numerical operations. The square brackets, `[ ]`, cannot be used in mathematical calculations.

You may have more than one set of parentheses in an expression, but they must always be "paired up." If you leave out a parenthesis (so that the expression can be said to be "unbalanced"), the HP-85 will return an error message when you press [END LINE]—it won't even try to compute the answer.

When parentheses are used, they take highest priority in the mathematical hierarchy. When parentheses are nested (i.e., when one pair of parentheses is contained inside another pair), like (5*(4−2)), the innermost quantity (4−2) is evaluated first.

Suppose you wish to evaluate the following expression:

$$2 + \frac{3 \times 6}{(7-4)^2}$$

Key it into the computer in one line as follows:

`2 + 3 * 6 / (7 - 4) ^ 2`

The computer scans an expression from left to right performing the operations of highest priority first. Thus, the above expression would be executed in the following manner:

2 + 3 * 6 / (7 - 4) ^ 2	Multiplication.
2 + 18 / (7 - 4) ^ 2	Evaluate parentheses.
2 + 18 / 3 ^ 2	Exponentiation.
2 + 18 / 9	Division.
2 + 2	Addition.
4	Result.

Whenever you are in doubt as to the order of execution for any expression, use parentheses to indicate the order.

Using parentheses for "implied" multiplication is not allowed. So 3(9−5) must appear as 3*(9-5). The operator, *, must be used explicitly to specify multiplication.

The RESULT Key

The value that is displayed after you press the END LINE key to execute a numeric expression is stored in a location called "RESULT." It is obtained for use in other calculations by pressing SHIFT RESLT.

For instance, what if you decided to multiply the result of our last calculation by 3.7:

```
(2+3*6/(7-4)^2)*3.7
```

Press	Display	
RESLT *3.7	4*3.7	The RESLT key immediately displays last result.
END LINE	14.8	Now 14.8 is the result.

Now suppose you wish to square this result:

Press	Display	
RESLT * RESLT	14.8*14.8	
END LINE	219.04	Now 219.04 is the result.

PRINT and DISP

The PRINT statement and the DISP statement are two important program statements. But they can also be used in calculator mode, to have the results of calculations printed, or to output results concurrently. Both of these statements will be discussed further in section 5.

If you wish to display the results of two or more equations simultaneously, use the `DISP` statement and separate your expressions with commas or semicolons. If you use commas, the results will be "spread apart," whereas semicolons will cause the results to be packed together.

Examples:

```
DISP PI*12^2/4,PI*12
 113.097335529
 37.6991118431
```

Execute the statement by pressing (END LINE). Results displayed.

```
DISP 80*43;83*44;86*45;89*46
 3440  3652  3870  4094
```

Press (END LINE) to display results.

Or, if you wish to output only the results of your calculations to the printer, use the `PRINT` statement. Press (END LINE) to print results.

Examples:

```
PRINT 222*11,528*8
PRINT 80*43;83*44;86*45;89*46

 2442                 4224
 3440  3652  3870  4094
```

Standard Number Format

Your HP-85 has been designed so that for most computations, your results appear in an easy-to-read form, as specified by ANSI*.

All results are calculated with the full precision of the computer. Results are displayed or printed in the following manner unless you specify otherwise in a program statement. (Refer to section 10.)

In standard format:

- All significant digits of a number (maximum of 12 digits) are printed or displayed. For example, if you typed `9876543210.12345` it would be output as `9876543210.12`.
- Excess zeros to the right of the decimal point are suppressed. For example, `32.100000` would be output as `32.1`.
- Leading zeros are truncated. For example, `00223.` is output as `223`.
- Numbers whose absolute values are greater than or equal to 1, but less than 10^{12} are output showing all significant digits and no exponent.
- Numbers between -1 and 1 are also output showing all significant digits and no exponent if they can be represented precisely in 12 or fewer digits to the right of the decimal point.
- All other numbers are expressed in scientific notation.

Let's look at a few examples of standard format. In the following table, if you key in the number in the left column and press (END LINE), that number will be displayed in the format shown in the right column.

* American National Standards Institute.

Number	Standard Format
15.000	15
00.23500	.235
-.0547∧9	-4.38415537301E-12
000987.5	987.5
10000∧6	1.E24
.01E4	100
120E-4	.012

Scientific Notation

In the right-hand column above, you see two numbers expressed in scientific notation. When you execute an expression in which the result is too large or too small to be displayed fully in 12 digits, the number is displayed with a single digit to the left of the decimal point, followed by up to 11 digits to the right of the decimal point, followed by the letter E and an exponent of 10.

12-Digit Mantissa Sign of Exponent

-4.38415537301E-12

Sign of Mantissa Denotes Exponent of 10

For example:

Press	Display	
60000*90000000	60000*90000000	
END LINE	5.4E12	Result, 5.4×10^{12}.
.00006*.00000009	00006*.00000009	
END LINE	5.4E-12	Result, 5.4×10^{-12}.

Keying In Exponents of Ten

You can key in numbers multiplied by powers of 10 (like the last two examples in the table above), by typing the number, then E, followed by an exponent of 10. For example, to key in 15.6 trillion (15.6×10^{12}) and multiply it by 25:

Press	Display	
15.6E12*25	15.6E12*25	
END LINE	3.9E14	Result.

To key in negative exponents of 10, type the number, type E, and then type the negative exponent. For example, type Planck's constant (h)—roughly, 6.625×10^{-27} erg seconds—and multiply by -50.

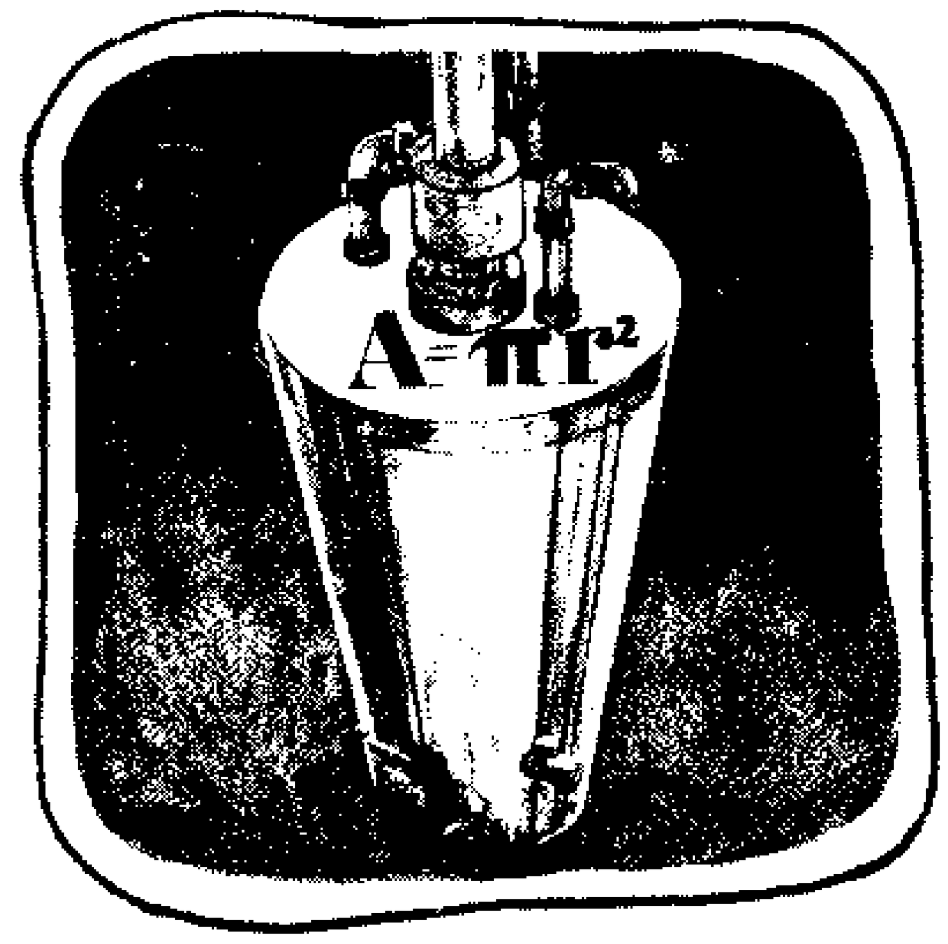
Press	Display	
6.625E-27*-50	6.625E-27*-50	
END LINE	-3.3125E-25	Erg seconds.

Range of Numbers

The range of values which can be entered or stored is $-9.99999999999 \times 10^{499}$ through -1×10^{-499}, 0, and 1×10^{-499} through $9.99999999999 \times 10^{499}$.

Variables

Algebraic formulas usually contain names that represent assigned values. These names are known as variables and, with the HP-85, specify a location in memory where a value is stored. For instance, the formula for the area of a circle, $a = \pi r^2$, contains two variables, a and r. To use the formula, you assign a value to r (radius) to solve for a.



Types

With the HP-85 you can specify either numeric variables or "character string" variables. Characters strings, or "strings" for short, can be composed of any valid characters and can be of any length—from zero characters to a maximum limited only by available memory. But since numeric data is more often used, we will discuss numeric variables first, then touch briefly on string variables. We'll continue our discussion on variables in section 8.

There are three types of numeric variables allowed by the HP-85.

- REAL numbers are stored with the full precision of the computer. REAL numbers are represented internally with 12 digits and a three-digit exponent in the range of −499 through 499; in other words, a 12-digit number in the range $-9.99999999999 \times 10^{499}$ through $-1.00000000000 \times 10^{-499}$, 0, and $1.00000000000 \times 10^{-499}$ through $9.99999999999 \times 10^{499}$.
- SHORT numbers are represented internally with five digits and a two-digit exponent in the range −99 through 99; in other words, a five-digit number in the range of -9.9999×10^{99} through -1.0000×10^{-99}, 0, and 1.0000×10^{-99} through 9.9999×10^{99}.
- INTEGER numbers are stored with five digits, with no digits following the decimal point. The range of integers is −99999 through 99999.

All numbers are full precision (real) unless you specify otherwise. But you can conserve computer memory if you designate SHORT or INTEGER numbers; refer to page 141.

Forms

There are two forms that a variable may have:

- Simple.
- Array (subscripted).

With simple variables, you assign a numeric value or expression to a name. Arrays are convenient for handling large groups of data within a program.

Simple variables can be assigned values in either calculator mode or within a program.

Calculator mode variables are temporary—they are cleared from memory whenever you run a program or press SCRATCH or RESET. Use them when you want to calculate immediate results from the keyboard. Otherwise, use variables in programs, where you can use them over and over again. The following statements about variable names and assignments apply to both calculator mode variables and program variables.

Simple Variables

On the HP-85 you can use the following for simple variable names:

- Any letter from A through Z. (Small letters can be used, but they are interpreted as if they were capital letters.)
- Any letter immediately followed by a digit from 0 through 9.

For instance, acceptable simple variables names are: A1, c, F0, j, J5, r2, x, Y.

Note: Lower case variable names are turned into upper case letters by the system (e.g., a1 is the same as A1).

In all, 286 simple variables can be named.

Variables are assigned values using an equal sign to create an assignment statement. For example, to assign 15 to A and 2*25 to X3:

Press	Display
A = 15 END LINE	A = 15
X3 = 2*25 END LINE	X3 = 2*25

In the assignment statement, the variable name appears first, followed by the equal sign. The value or numeric expression assigned to the variable appears to the right of the equals sign.

Now that some variables have assigned values, they can be used in place of numbers in math calculations:

Press	Display	
A/X3 END LINE	A/X3	
	.3	Result of 15/50.
A^2 END LINE	A^2	
	225	Result of 15^2.
X3 * 3 END LINE	X3*3	
	150	Result of 50 * 3.

Variables can be reassigned values. For instance to change the value of A to 16 you could execute either A = A+1 or A = 16.

To recall the value of any assigned variable, simply type the variable name and press (END LINE).

Press	Display	
A (END LINE)	A	
	15	Value of A.
X3 (END LINE)	X3	
	50	Value of X3.

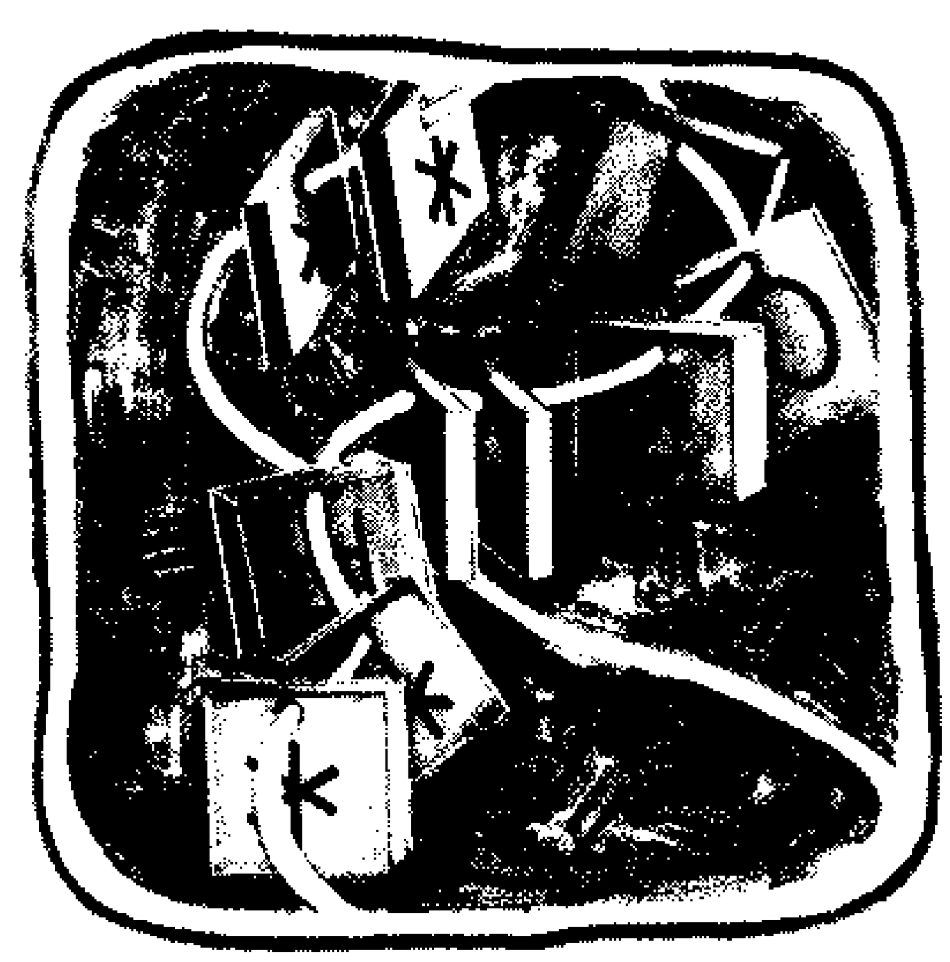
You can assign the same value to more than one variable in the same line by using commas to separate the variables. For example, assign the variables A, B, C, and D the value of 100.

Press	Display	
A, B, C, D = 100 (END LINE)	A,B,C,D=100	
A (END LINE)	A	Verify that all variables have been assigned the value 100.
	100	
B (END LINE)	B	
	100	
C (END LINE)	C	
	100	
D (END LINE)	D	
	100	

You can use one more type of numeric variable on the HP-85—an array variable. We'll discuss arrays in section 8.

String Variables

A character string is a series of characters like **HELLO!** that can be given a string variable name. The length of the string refers to the number of characters assigned to the string. A string variable can be any length (limited only by available memory).



You can use string variables without dimensioning them (allocating memory to them) if they contain less than 18 characters. If they are longer than 18 characters, you must use a DIM or COM statement to declare the length (page 121).

String variables are assigned names in the same way that numeric variables are assigned names, but the string variable name must be followed by a dollar sign ($).

For example, acceptable string variable names are: A1$, C$, F0$, j$, J5$, r2$, X$, Y$.

In all, 286 string variables can be named. (Remember, the system interprets small letters in variable names as if they were capital letters; thus, you could reference the same string variable A1$ with a1$.)

To assign HELLO to A$, and GOODBYE to B$:

Press	Display	
A$="HELLO" (END LINE)	A$="HELLO"	The string must
B$="GOODBYE" (END LINE)	B$="GOODBYE"	be enclosed with
DISP A$,B$ (END LINE)	DISP A$,B$	quotation marks.
	HELLO GOODBYE	
DISP A$;"-";B$ (END LINE)	DISP A$;"-";B$	
	HELLO-GOODBYE	

String Concatenation

"Concatenation" is the one operation allowed in string expressions. This operation causes one string to be tacked onto the end of another. The symbol used for string concatenation is the ampersand (&). To join two strings together, it is necessary only to interpose the ampersand.

For example, assign the following string variables the characters shown:

```
A$="BUTTER"
B$="DRAGON"
C$="HOUSE"
D$="FLY"
E$=" "
```

Press (END LINE) after each assignment statement.

The string variable E$ contains one space.

Now execute these statements:

```
DISP A$ & D$ (END LINE)
BUTTERFLY
```

Displays the two strings joined together.

```
F$=B$ & D$ (END LINE)
F$ (END LINE)
DRAGONFLY
```

Joins strings B$ and D$ to make F$.

```
DISP F$ & E$ & C$ & D$ (END LINE)
DRAGONFLY HOUSEFLY
```

Since concatenation makes a string longer than its parts, be sure that the final string in a string variable assignment is less than or equal to 18 characters in length. (Or, if the string has been dimensioned using DIM or COM statement in a program, the final string must not exceed the length that you have designated.)

The Null String

The null string is a string that contains no characters or blanks, like:

```
N$=""
```

We define the null string here because it is referred to often.

Logical Evaluation

In logical evaluation, expressions can be compared by using relational operators and/or logical operators. An expression can be a constant (like 7.2), a variable (like B), or an arithmetic expression (like 7.2*SQR(6)). If the comparison is 'true', the value '1' is returned; if the comparison is 'false', the value '0' is returned.

Relational Operators

Relational operators are used to determine the value relationship between two expressions.

Symbol	Meaning
=	Equal to.
<	Less than.
>	Greater than.
<=	Less than or equal to.
>=	Greater than or equal to.
<> or #	Not equal to (either form is acceptable).

The < symbol corresponds to the shifted (<;) key, > corresponds to the shifted (>.) key, and # corresponds to the shifted (#3) key.

Be careful to note that the equal sign is used in both variable assignment statements and in relational operations. This distinction only becomes important at the beginning of an expression that could be interpreted either way; in which case, the system will always assume that the expression is a variable assignment. To specify a relational operation, place parentheses around the equality relational operation or place the value to the left of the equal sign and the variable it is being compared with to the right of the equal sign.

Examples:

A=3	Assigns A the value of 3.
(A=3) 3=A	Both expressions perform the equality relational operation and compare the value of A with 3. They return values of 0 or 1 depending on whether A has a value of 3.
A,B=3	Assigns A and B the value of 3.
A=B=3 A=(B=3)	Both statements assign the value 0 or 1 to A, depending on whether B does or does not equal 3. You don't need to use parentheses since the variable name is to the left and its value (the result of the expression B=3) is to the right of the equal sign.

Let's look at some examples using relational operations. First let's assign values to the variables A,B,C, and D.

```
A = 1
B = 2
C,D=3
```

Press (END LINE) after each line to assign the variable(s) the specified value.

Now execute the following operations:

`A < B`	1 < 2
`1`	True.
`B < A`	2 < 1
`0`	False.
`B # C`	2 # 3
`1`	True.
`C # D`	3 # 3
`0`	False.
`3 = C`	3 = 3
`1`	True.
`4 = A`	4 = 1
`0`	False.
`A = 4`	Assigned 4 to A.

As you can see, the last statement did not assign a value of `1` (true) or `0` (false) to the expression because `A = 4` is an assignment statement; so A is assigned the value of 4. To determine the value relationship between the value of A and 4, type `4 = A`, as shown above, or use the parentheses around this expression: (`A=4`).

Strings and string variables can also be compared using the relational operators. Each character in the string is represented by a standard decimal code, as shown in appendix C. When two string characters are compared, the lesser of the two characters is the one whose decimal code is smaller. For example, 3 (decimal code 51) is smaller than B (decimal code 66).

Strings are compared, character by character, from left to right until a difference is found. If one string ends before a difference is found, the shorter string is considered the lesser.

Relational operators are valuable when they are used in `IF` ... `THEN` statements as described in section 7.

Advanced Programming Note: Relational comparisons can be quite complex. Suppose the following statements are used in a program:

`X=69*(A=3)+287*(B=83)`	Assigns 69 to X when A=3 and adds 287 when B=83, otherwise adds nothing.
`G1=4*(A$="A")+3*(A$="B")+2*(A$="C")+(A$=D$)`	Assigns 4 for an "A", 3 for a "B", 2 for a "C" and 1 when A$=D$.
`L=L+(LEN(A$)>9)`	Adds 1 to L when A$ is longer than nine characters.

Logical Operators

The logical operators, often called 'Boolean' operators, are `AND`, `OR` (inclusive or), `EXOR` (exclusive or), and `NOT`. A value of zero is considered false. Any other value is considered true. The result of a logical operation is either 0 or 1.

- AND checks two expressions. If both expressions are true, (that is, both non-zero), the result is true (1). If one or both of the expressions is false (0), the result is false (0).

A	B	A AND B
T	T	T
T	F	F
F	T	F
F	F	F

- OR checks two expressions. If one or both of the expressions is true, the result is true (1). If neither expression is true, the result is false (0).

A	B	A OR B
T	T	T
T	F	T
F	T	T
F	F	F

- EXOR (exclusive or) compares two expressions. If only one of the expressions is true, the result is true (1). If both are true or both are false, the result is false (0).

A	B	A EXOR B
T	T	F
T	F	T
F	T	T
F	F	F

- NOT returns the opposite of the logical value of an expression. If the expression is true (non-zero), the result is false (0). If the expression is false (0), the result is true (1).

A	NOT A
T	F
F	T

We have used A and B in the truth tables to denote numeric expressions. The expressions used with logical operators can be either relational or non-relational. In the order of execution of expressions, NOT has higher priority than the relational operators and the relational operators have higher priority than AND, EXOR, and OR. If you are in doubt, use parentheses.

Here are some examples; first let's assign values to the variables A,B,C, and D.

```
A = 0
B = 2
C,D = 4
```

Press (END LINE) after each line to assign the variables the specified values.

Now execute these logical expressions:

```
A<B AND C=D
 1
```

Since both relational expressions A<B and C=D, are true, the result is true.

```
A AND C=D
 0
```

The expression, A, is false since its arithmetic value equals zero. The expression, C=D, is true. But since AND requires that both expressions be true to return a true result, the result is false.

```
A OR B
 1
```

The arithmetic value of A is zero (false) while the arithmetic value of B is two (true). Since at least one of the expressions is true, the whole expression is true.

```
A OR C-D
 0
```

Both arithmetic expressions have a value of 0 (false).

```
NOT A
 1
```

Since A is false. NOT A is true.

```
A EXOR B=2
 1
```

Since A is false and B=2 is true, the result is true.

Advanced Programming Note: The results returned from logical or relational operations, either 0 or 1, can be used in calculations. Using the variables, A, B, C, and D again, let's evaluate S in the equation shown below:

```
S = ((B<C) + (NOT D=A))*12 [END LINE]
S [END LINE]
  24
```

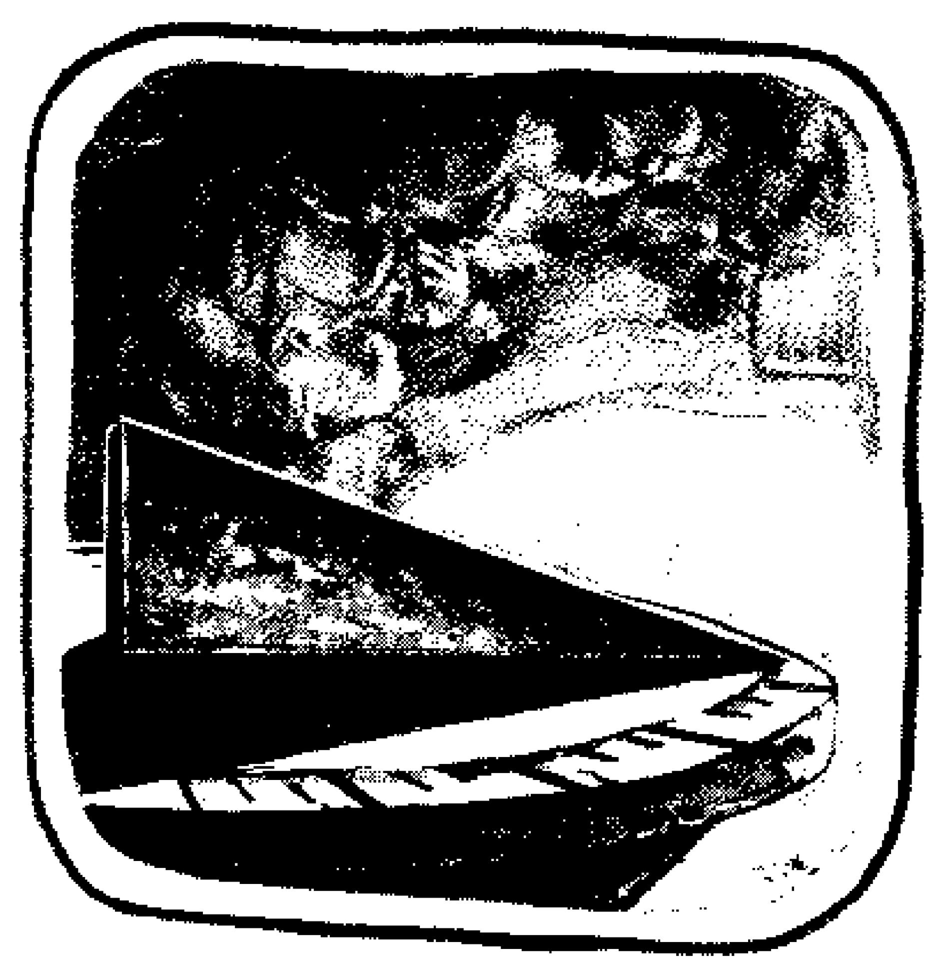
The result of the true relation (`B<C`) is first added to the result of the true relation (`NOT D=A`). In other words 1 + 1 = 2. This result is then multiplied by 12 for a product of 24.

Here's a truth table summarizing logical operations:

A	B	A AND B	A OR B	A EXOR B	NOT A	NOT B
T	T	1	1	0	0	0
T	F	0	1	1	0	1
F	T	0	1	1	1	0
F	F	0	0	0	1	1

The Time Functions

Often it is desirable to document programs, computations, and test runs with the current time and date of execution. With the HP-85 you can set the time and date and then recall the current time whenever you wish. You can even use the time functions in calculations.



As soon as you turn the power on, the system timer is set to 0 and begins to count the time in milliseconds. After it counts 86,400 seconds (24 hours), the system timer increments the date by 1, and then starts to count milliseconds from 0 again.

You can specify the starting time and date for the system counter with the `SETTIME` statement as follows:

`SETTIME` seconds since midnight `,` day of the year

Although the time must be set in seconds to count properly, the date can be specified any way you want—as long as you remember that the date is an integer number that is incremented by 1 at midnight (assuming the system timer has been set properly).

For example, you can set the timer at 8 a.m., March 16, 1980 as follows:

`SETTIME 28800,80076` 28800 seconds since midnight, 76th day of 1980; date in form yyddd.

Eight o'clock in the morning is 8 hours × 60 minutes/hour × 60 seconds/minute = 28800 seconds since midnight. And March 16, 1980 is the 76th day of 1980.

Now the timer will increment 28800.000 by 1 every millisecond until the time is 86400 (midnight). Then it will add 1 to 80076 and start counting seconds from 0 again.

Since the date is just an integer number that is incremented by 1 every 24 hours, you can enter the date in any form you wish, as long as the number is between 1 and 99999. The time parameter can be a numeric expression with a value of 0 through 86400. If the timer is set to 86400, midnight, the system immediately increments the date and begins counting milliseconds from 0 again.

For instance, you could have set the date and time for 8 a.m., March 16th as follows:

`SETTIME 8*60*60,316`	Date in form mdd.

Notice that you can use a numeric expression to set the time. We used the number 316 to specify the 3rd month and the 16th day, but remember, the system interprets 316 as just a number to be incremented by 1 at midnight.

The `TIME` function recalls the current time in seconds since midnight, assuming the time has been set properly, or the number of seconds since power on if the time was not set with `SETTIME`. To recall the time, type:

`TIME` [END LINE]

The `DATE` function recalls the current date in the same format that you specified, or it recalls 0 if you had not set the date with `SETTIME`. To recall the date, type:

`DATE` [END LINE]

All values for `SETTIME`, `TIME`, and `DATE`, are lost when you turn the power off. `TIME` begins counting from 0 each time the power is turned on.

The `TIME` and `DATE` functions are programmable and can be used in numerical expressions.

For instance, to recall the time in hours, execute:

`TIME/3600` [END LINE]	Time in seconds divided by (60 minutes/hour × 60 seconds/minute). Result gives time in decimal hours.

Mathematics Functions and Statements

Many predefined functions are available to you through BASIC programming language on the HP-85. But you don't need to write a program in order to use them. Each function operates the same way, regardless of whether you execute the function straight from the keyboard or use it as part of a program statement.

In this section:

- Each built-in math function is explained as it is used manually, in its simplest form.
- The math functions are placed in the total math hierarchy.
- Math errors are discussed in conjunction with the DEFAULT ON statement.

A **function** is a prescription for doing something with a given value, or set of values, that yields a single output. The values that are acted upon by a function are called the "arguments" or, sometimes, the "parameters." An argument is often just a single number, but it may be a mathematical expression containing variables or other functions.

Most of the functions on the HP-85 require only one argument, but there are several that require two, and a few that require none.

To use any of the functions in "calculator" mode:

1. Type the function name.
2. Then type the argument, if the function requires one, enclosed within parentheses. If the function requires two arguments, separate them with a comma.
3. Press (END LINE) to compute the result.

Appendix D lists all of the functions available to you with BASIC on the HP-85.

Number Alteration

There are several functions that allow you to alter numbers on the HP-85. These functions are: ABS, INT or FLOOR, CEIL, FP, and IP. The table below lists the function name and argument along with the meaning of the function. The argument X may be a number (like 2.75), a variable (like A), or a numeric expression (like 3*SQR(A)).

Function and Argument	Meaning
ABS(*X*)	Absolute value of *X*.
IP(*X*)	Integer part of *X*.
FP(*X*)	Fractional part of *X*.
INT(*X*)	Greatest integer less than or equal to *X*.
FLOOR(*X*)	Greatest integer less than or equal to *X*. (Same as INT(*X*); relates to CEIL)
CEIL(*X*)	Smallest integer greater than or equal to *X*.

Note: IP and INT differ only with negative numbers.

Absolute Values

Some calculations require the absolute value, or magnitude, of a number. To obtain the absolute value of any expression, simply type ABS(expression), where the expression may be a constant, a variable, or an arithmetic expression. Then press (END LINE). The result will be displayed below the line you type.

Examples:

```
ABS(-235)
 235
```
Press (END LINE) to display the result $|-235|$.

```
ABS(2.7)
 2.7
```
$|+2.7|$

```
ABS(4-7/1.5)
 .666666666667
```
$|4-7/1.5|$

Integer Part of a Number

To extract and display the integer part of a number, type IP, followed by the argument enclosed within parentheses. Then press (END LINE).

Examples:

```
IP(123.456)
 123
```
Press (END LINE) and the integer part of the number is displayed.

```
IP(-4.56)
-4
```
Integer part of -4.56.

```
IP(1.748)
 1
```
Integer part of 1.748.

When the IP function is executed, the fractional part of the number is lost.

Fractional Part of a Number

To extract and display only the fractional part of a number, type FP, followed by the argument enclosed within parentheses. Then press (END LINE).

Examples:

```
FP(123.456)
 .456
```
When you press (END LINE), only the fractional part of the number is displayed.

```
FP(-4.56)
-.56
```
Fractional part of -4.56.

```
FP(1.748)
 .748
```
Fractional part of 1.748.

When the FP function is executed, the integer part of the number is lost.

Greatest Integer Function

To display the greatest integer less than or equal to a number, type INT or FLOOR, followed by the number or expression enclosed within parentheses. The greatest integer function returns the largest integer that is less than or equal to the evaluated expression.

Examples:

`INT(123.456)` `123`	123 is the greatest integer <= 123.456.
`FLOOR(123.456)` `123`	`FLOOR` performs the same function as `INT`.
`INT(-6.257)` `-7`	−7 is the greatest integer <= −6.257.
`INT(-1.748)` `-2`	−2 is the greatest integer <= −1.748.

Note the difference between the `IP` *(integer part)* function and the `INT` or `FLOOR` *(greatest integer)* function. In the above examples, `IP(-6.257)` yields `-6`, while `INT(-6.257)` yields `-7`.

Smallest Integer Function

To display the smallest integer greater than or equal to a number, type `CEIL` *(ceiling)*. Then type the number or expression, enclosed within parentheses, and press [END LINE].

Examples:

`CEIL(123.456)` `124`	124 is the smallest integer >= 123.456.
`CEIL(-6.257)` `-6`	−6 is the smallest integer >= 6.257.
`CEIL(-1.748)` `-1`	−1 is the smallest integer >= −1.748.

General Math Functions

Several of the following functions do not require an argument. For instance, `PI` always returns the 12-digit approximation of π. A few of the functions below require two arguments; for example, given two values, `MAX` returns the larger of the two values. The arguments, denoted by X and Y, may be numbers, numeric variables, functions, or numeric expressions.

Function and Argument	Meaning
`SQR(`*X*`)`	Positive square root of *X*.
`SGN(`*X*`)`	Sign of *X*; yields −1 if $X < 0$, 0 if $X = 0$, and +1 if $X > 0$.
`MAX(`*X*`,`*Y*`)`	Maximum; if $X > Y$ returns *X*, otherwise returns *Y*.
`MIN(`*X*`,`*Y*`)`	Minimum; if $X < Y$ returns *X*, otherwise returns *Y*.
`RMD(`*X*`,`*Y*`)`	Remainder of *X* divided by *Y*: $X-Y$*IP(X/Y).
`PI` *no argument*	12-digit approximation of π; 3.14159265359.
`INF` *no argument*	Machine infinity (+9.99999999999E499).
`EPS` *no argument*	Epsilon; smallest positive machine number (1E-499).
`RND` *no argument*	Random number; generates next number in a sequence of numbers greater than or equal to zero and less than one.

Square Root Function

To calculate the square root of a number, use the SQR function. The square root function returns the square root of a nonnegative expression.

Examples:

```
SQR(88)
 9.38083151965

SQR(16.1)
 4.01248052955
```

When you press (END LINE), the square root of the number is displayed.

Sign of a Number

The sign function returns a 1 if the expression is positive, 0 if it is 0, and −1 if it is negative. To use the sign function, type SGN, followed by the argument enclosed within parentheses. Then press (END LINE).

Examples:

```
SGN(-5)
-1
```

Sign of a negative number is −1.

```
SGN(0)
 0
```

Sign of zero is 0.

```
SGN(4.3)
 1
```

Sign of a positive number is 1.

Maximum and Minimum

You'll find the MAX and MIN functions very useful in BASIC programs. Given two values, MAX returns the larger of the values and MIN returns the smaller. Both MAX and MIN require two arguments enclosed within parentheses, following the function name. And these two arguments must be separated by a comma.

Examples:

```
MAX(4.5,4.76)
 4.76

MIN(4.5,4.76)
 4.5

MAX(-1,-5)
-1

MIN(-1,-IP(SQR(5)))
-2
```

Note that the arguments must be separated by a comma. The arguments may be numbers, simple variables—if the variables are defined—or arithmetic expressions (including functions).

The Remainder Function

Given two values, the remainder function, RMD, divides the first value by the second and displays the remainder. RMD(X,Y)= X-Y*IP(X/Y).

Examples:

```
RMD(5,2)
 1
```

2 goes into 5 twice, with a remainder of 1.

```
RMD(17.35,4.26)
 .31
```

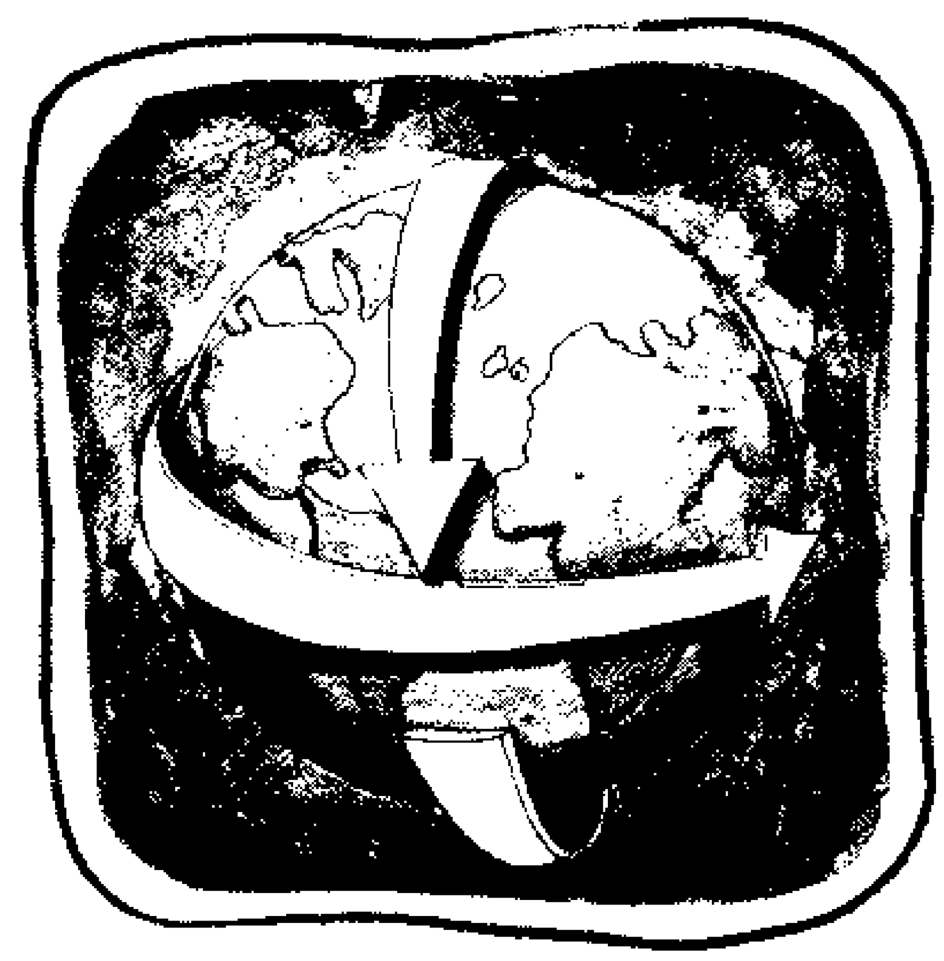
4 times 4.26 plus remainder .31 is equal to 17.35.

Comparing definitions, you can see that the RMD function and the MOD operator (page 44) are very similar. In fact, they both yield the same results when the arguments X and Y have the same sign. But they can give different results when X and Y are of opposite signs.

Whether you use RMD or MOD depends on the particular application you choose.

Example: Resolve $-726°$ to lie between $-360°$ and $+360°$ by ignoring multiples of 360°. Using the RMD function, given any θ in degrees:

$$\theta_{new} = \text{RMD}\ (\theta, 360) \text{ where } -360° < \theta_{new} < 360°$$



```
RMD(-726,360)
-6
```

With the RMD function, θ_{new} is equal to -6°.

Now use the MOD operator to resolve $-726°$ to lie between 0° and 360° by ignoring multiples of 360°. Given any θ in degrees:

$$\theta_{new} = \theta \text{MOD} 360 \text{ where } 0° < \theta_{new} < 360°$$

```
(-726)MOD360
 354
```

$\theta_{new} = 354°$

Using PI

The value of π approximated to 12 places (3.14159265359) is provided as a fixed constant in BASIC programming language. Merely type PI whenever you need it in a calculation. For example, to calculate 3π, type:

```
3 * PI
 9.42477796077
```

When you press (END LINE), the result is displayed.

Example: Calculate the surface area of Callisto, one of Jupiter's 12 moons, using the formula $A = \pi d^2$. Callisto has a diameter (d) of 3100 miles.

```
PI * 3100 ^ 2
 30190705.401
```

Area of Callisto in square miles.

Note that you don't have to include parentheses around 3100^2 because exponentiation is performed before multiplication.

Epsilon and Infinity

Two functions that prove useful in programs are EPS and INF. Both functions simply recall a constant; EPS recalls the smallest positive machine number and INF recalls the largest machine number. They are useful in comparisons when you want to use either a very small or a very large number, saving you the time of keying in the numbers yourself. (Recall how we used EPS in the Averaging Program in section 1.)

Examples:

```
EPS
 1.E-499
```

Smallest positive number that can be output: 1×10^{-499}.

```
INF
 9.99999999999E499
```

Largest number that can be output: $9.99999999999 \times 10^{499}$.

```
MAX(458, INF)
 9.99999999999E499

MIN(32, EPS)
 1.E-499
```

Random Numbers

Random numbers are extremely useful in statistical sampling theory—anytime you want a sequence of events or numbers that appear in an unpredictable order. The random number function, RND, returns a pseudorandom number greater than or equal to 0 and less than 1, each time it is executed.

Example:

```
RND
 .529199358633
```

A random number between 0 and 1 is displayed each time RND is executed.

```
RND
 4.35821814444E-2

RND
 .294922982088
```

Whenever you turn the power on or press (RESET), the same sequence of random numbers is generated. The reason for this is that the random number function uses the same 'seed' (i.e., the number upon which the sequencing is based) each time it is reset.

But, by using the RANDOMIZE statement, you can "scramble" the seed and thus, generate new sequences of random numbers. Or you can control the sequences of random numbers by specifying your own "seed."

To see how this works, type:

```
RANDOMIZE
```

When you press (END LINE), the HP-85 defines a new seed for the random number generator, based on the internal timing system. Now, execute the RND function several times, until it becomes evident that you have generated a new sequence of random numbers.

Each time you use the RANDOMIZE statement in this way, a new "seed" is defined, yielding a new sequence of random numbers.

You can control the sequence of numbers by specifying the "seed" with the RANDOMIZE statement. This enables you to regenerate the same sequence of numbers whenever you wish.

Example: Using the seed .423, generate the first three numbers of the random number sequence.

```
RANDOMIZE .423          Execute the RANDOMIZE
                        statement.
RND
 .543851130928          1st random number in the sequence.
RND
 .908089747097          2nd random number in the sequence.
RND
 .484429755623          3rd random number in the sequence.
```

Whenever you wish to use the same sequence of random numbers, use the same seed. To obtain a good seed, use any non-zero number within the range of the HP-85—the system will automatically convert the number to a seed between 0 and 1. A seed of zero will generate a constant sequence of zeros.

For any non-zero seed in the given range, 5×10^{13} values are generated before the sequence repeats. (The RANDOMIZE statement will always generate a non-zero seed if no parameter is specified.)

You are not limited to random numbers between numbers 0 and 1. In general, you can generate random integers from *a* through *b* using the formula IP(($b + 1 - a$) * RND + a). For instance, generate a random sequence of integers between 0 and 99, inclusive.

You could use an expression like the following:

```
IP(100*RND)             The integer part of a random number times
 5                      100. (These are the fourth, fifth, and sixth
IP(100*RND)             random numbers generated in the
 54                     sequence based on the seed above.)
IP(100*RND)
 8
```

Generally speaking, good statistical properties can be expected because the random number generator has been designed to pass an important test known as the spectral test.* Of course the statistics will vary somewhat from sequence to sequence depending on the starting seed since less than the full period will be used by you. But it should normally be quite good if a "statistically significant" sample size is considered.

Logarithmic Functions

The HP-85 computes both natural and common logarithms as well as their inverse functions. The logarithmic functions are:

Function and Argument	Meaning
LOG(*X*)	$\log_e X$; natural logarithm of a positive *X* to the base e (2.71828182846 to 12 place accuracy).
EXP(*X*)	e^X; natural antilogarithm. Raises e (2.71828182846) to the power *X*.
LGT(*X*)	$\log_{10} X$; common logarithm of a positive *X* to the base 10.

* Donald E. Knuth, *The Art of Computer Programming* (Massachusetts, 1969), V.2., §3.4

Of course, the common antilog (10^x) may be executed easily from the keyboard (`10^X`).

Example: What is the value of $\log_2 53$?

You can easily convert the logarithmic base using the following formula:

$$\log_a x = \frac{\ln x}{\ln a} = \frac{\log_e x}{\log_e a}$$

So, to find the logarithm, base 2, of 53, simply execute:

```
LOG(53)/LOG(2)
 5.72792045456                    log2 53.
```

Trigonometric Functions and Statements

Trigonometric Modes

When you are using trigonometric functions, angles can be assumed by the HP-85 to be in decimal degrees, radians, or grads. Unless you specify otherwise with one of the trigonometric mode statements, the HP-85 assumes that all angles are in radians. When you select a trigonometric mode, the HP-85 remains in that mode until you change it or the computer is turned off.

To select degrees mode, execute:

```
DEG                               There are 360 degrees in a circle.
```

To select grads mode, execute:

```
GRAD                              There are 400 grads in a circle.
```

To reset radians mode, press (RESET) or execute:

```
RAD                               There are 2π radians in a circle.
```

360 degrees = 400 grads = 2π radians

Trigonometric Functions

There are 12 programmable trigonometric functions provided by the HP-85, including inverses of several of the functions and conversion functions.

Function and Argument	Meaning
SIN(X)	Sine of X.
ASN(X)	Arcsine of X; $-1 \leq X \leq 1$. In 1st or 4th quadrant.
COS(X)	Cosine of X.
ACS(X)	Arccosine of X; $-1 \leq X \leq 1$. In 1st or 2nd quadrant.
TAN(X)	Tangent of X.
ATN(X)	Arctangent of X; in 1st or 4th quadrant.
CSC(X)	Cosecant of X.
SEC(X)	Secant of X.
COT(X)	Cotangent of X.
ATN2(Y,X)	Arctangent of Y/X, in proper quadrant; useful in polar/rectangular coordinate conversions.
Conversions:	
DTR(X)	Degrees to radians conversion.
RTD(X)	Radians to degrees conversion.

All trigonometric functions have one argument, except the ATN2 function, so to use them simply type the function name and then type the numeric expression, enclosed within parentheses.

Example: Find the cosine of 45 degrees.

```
DEG                         Sets the HP-85 to degrees mode.
COS(45)
 .707106781187              Result.
```

Example: Find the sine of 2/3π radians.

```
RAD                         Sets computer to radians mode.
SIN(2/3 * PI)
 .866025403786              Result.
```

Degrees/Radians Conversions

The DTR *(degrees to radians)* and RTD *(radians to degrees)* functions are used to convert angles between degrees and radians. To convert an angle specified in degrees to radians, type DTR followed by the angle within parentheses. For example, to change 45 degrees to radians:

```
DTR(45)
 .785398163397              Radians.
```

To convert the angle specified in radians to decimal degrees, type RTD followed by the angle within parentheses. Convert 4 radians to decimal degrees:

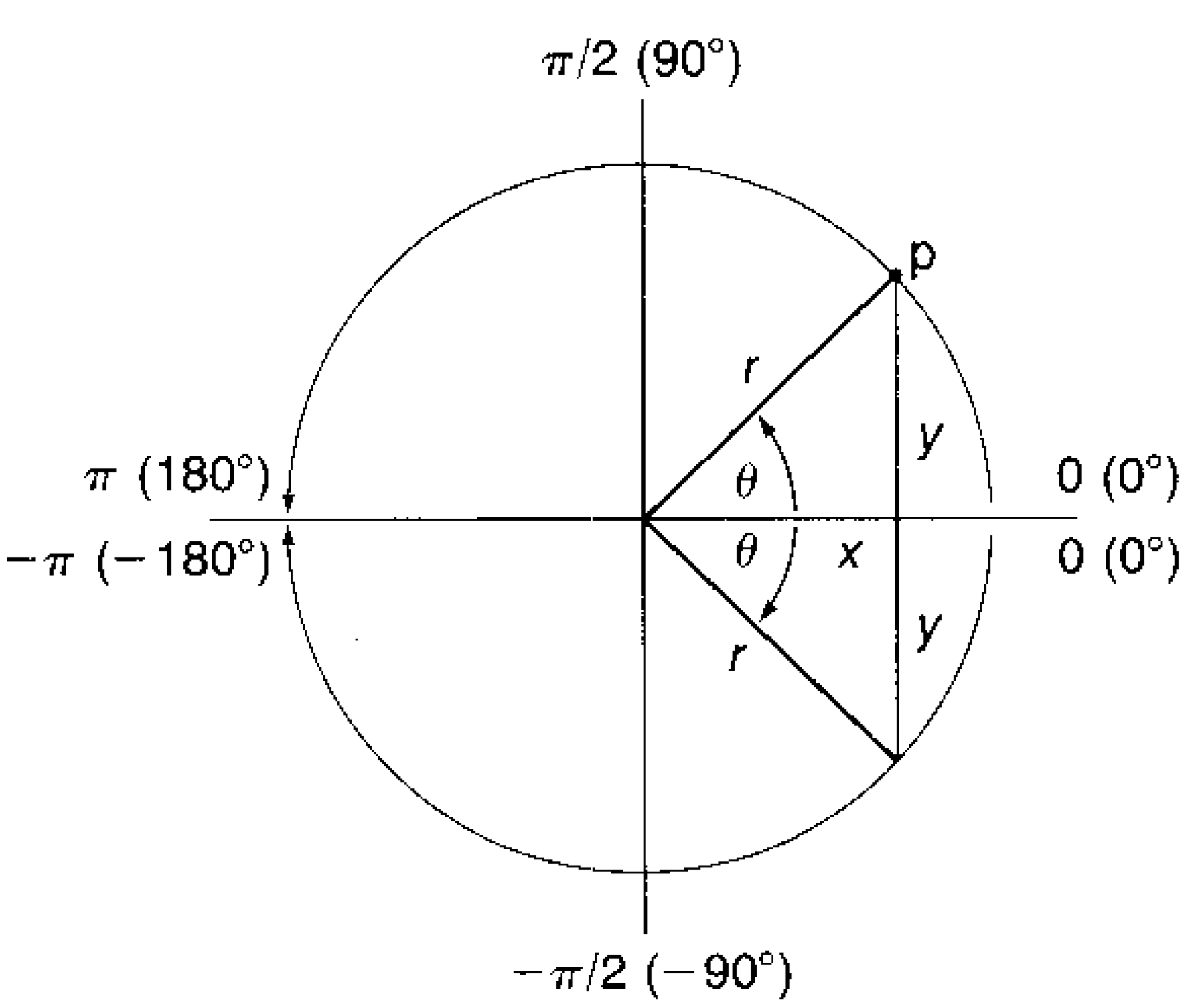
```
RTD(4)
 229.183118052              Decimal degrees.
```

Polar/Rectangular Coordinate Conversions

The ATN2 *(arctangent of x,y coordinate position)* function can be used for polar/rectangular coordinate conversions. Angle θ is assumed in decimal degrees, radians, or grads, depending upon the trigonometric mode first selected by DEG, RAD, or GRAD.

A point P can be represented in two ways: by the rectangular coordinate position (x,y) or by the polar coordinate position (r,θ).

In the HP-85, the ATN2 function produces an angle θ represented in the following manner:



To convert from rectangular (x,y) coordinates to polar (r,θ) coordinates (magnitude and angle, respectively), use the following equations:

$r = \sqrt{x^2 + y^2}$

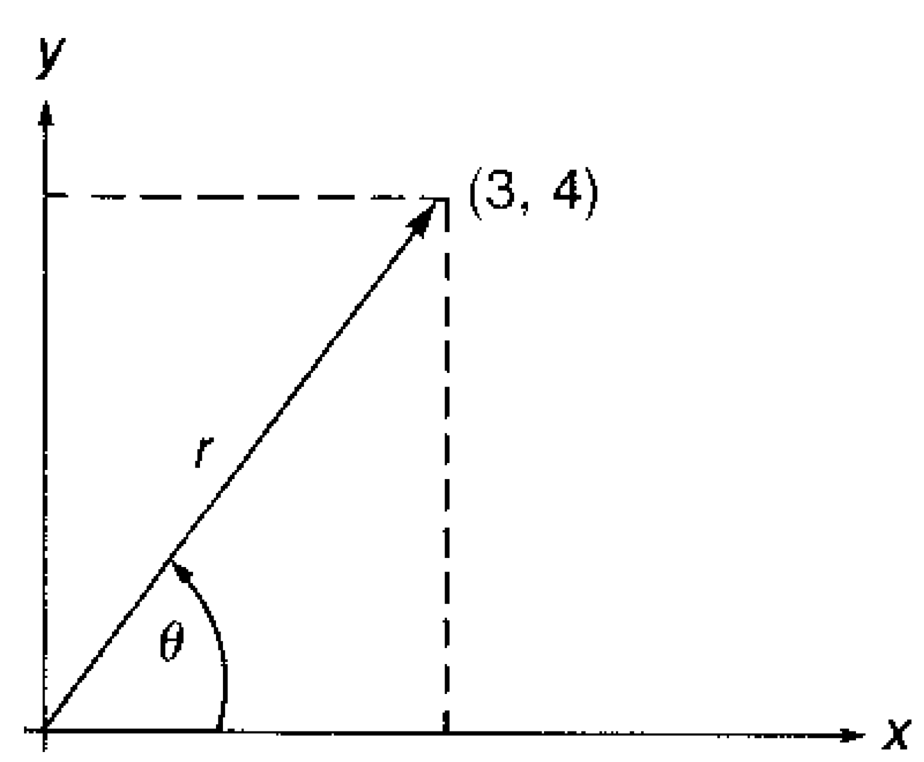
$\theta = \mathrm{ATN2}(y, x)$ where $-\pi < \theta \leq \pi$.

To convert from polar (r,θ) coordinates to rectangular (x,y) coordinates, use the following geometric properties:

$x = r\cos\theta$

$y = r\sin\theta$

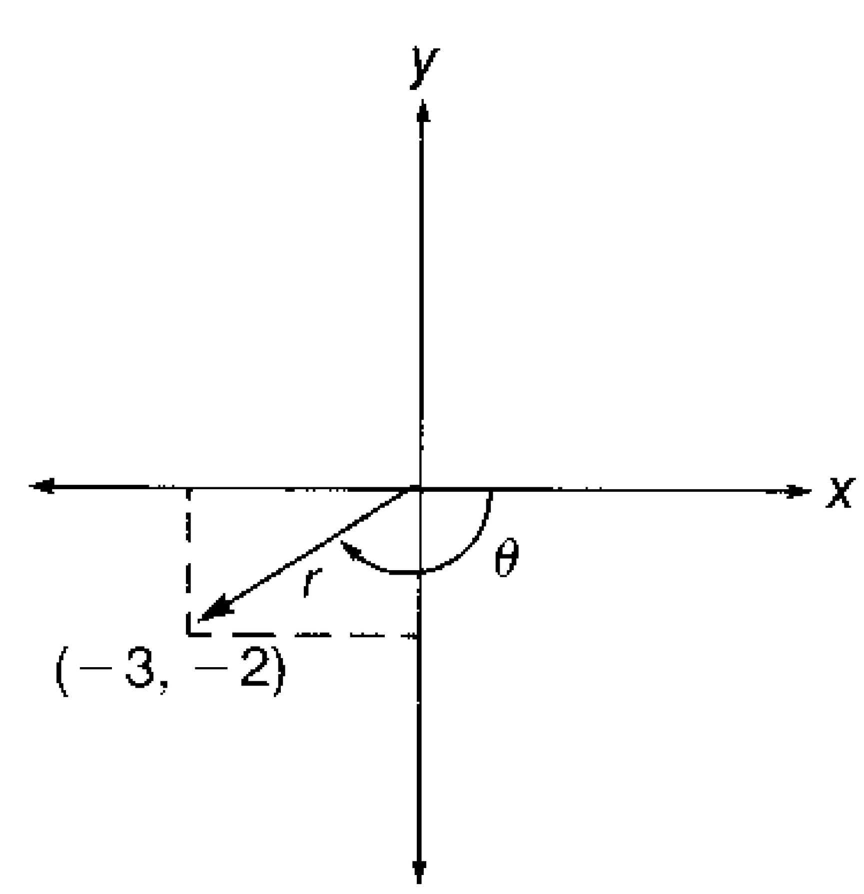
Example: Convert rectangular coordinates (3,4) to polar form with the angle expressed in decimal degrees.



`DEG`	Degrees mode selected.
`SQR(3^2 + 4^2)`	$r = \sqrt{x^2 + y^2}$.
`5`	Magnitude r.
`ATN2(4,3)`	$\theta = \mathrm{ATN2}(y,x)$; notice that we specify the y-coordinate value first.
`53.1301023542`	Angle θ in decimal degrees.

The `ATN2` function is also used to find the arctangent of an expression in the *proper* quadrant. The `ATN` function returns the principal value of the arctangent of an expression, in other words, the value in the first or fourth quadrant.

Example: Find the angle in the third quadrant whose tangent is 2/3. Express the angle in radians.



```
RAD
ATN2(-2,-3)

-2.55359005004
```

Set radians mode.
θ = ATN2(y,x), where x and y are any rectangular coordinates in the third quadrant with a tangent of 2/3.
Angle θ in radians.

(Note that `ATN(-2/-3)` would return the arctangent of 2/3 evaluated in the *first* quadrant: `.588002603548`.)

Total Math Hierarchy

Notice that functions are performed immediately after parentheses are evaluated in the order of execution for all mathematical operations.

Highest priority	`( )` parentheses
	Functions
	`^` (exponentiation)
	`NOT`
	`*`, `/`, `MOD`, `\` or `DIV`
	`+`, `-`
	All relational operators (`=`, `>`, `<`, `>=`, `<=`, `<>` or `#`)
	`AND`
Lowest priority	`OR`, `EXOR`

Recovering From Math Errors

Many math errors occur due to an improper argument or overflow. Such an error would normally halt the execution of a running program. The HP-85 provides default values for out-of-range results that occur using the following math functions, thus overriding the error condition and preventing the error from halting program execution. The system will alert you to the error by displaying a warning message and, if the result is to be output, the default value of the expression.

The default error processing condition is on when the system's power is turned on.

The errors and default values are:

Error (Number)	Default Values
Underflow (1)	0
Integer precision overflow (2)	+ or − 99999
Short precision overflow (2)	+ or − 9.9999E99
Real precision overflow (2)	+ or − 9.99999999999E499
`COT` or `CSC` of n*180°; n = integer (3)	9.99999999999E499
`SEC` or `TAN` n*90°; n = odd integer (4)	9.99999999999E499
Zero Λ negative power (5)	9.99999999999E499
Zero Λ zero (6)	1
Uninitialized numeric variable (7)	0
Uninitialized string variable (7)	""
Division by zero (8)	+ or −9.99999999999E499

For instance, try to divide a number by zero:

`34/0`	
`Warning 8 : /ZERO`	Beeps and displays a warning message.
`9.99999999999E499`	Answer; default value of expression.

Since the default condition is on at power on, the system beeps and displays a warning message to alert you to the error. But the cursor moves to the next line on the display after the warning so that, essentially, the error is ignored.

The `DEFAULT OFF` statement cancels the use of default values for math errors and sets the system to normal error processing. For instance, type:

`DEFAULT OFF`	Sets system to normal error processing.
`34/0`	Try dividing by zero again.
`Error 8 : /ZERO`	Beeps and displays an error message.

With `DEFAULT OFF`, such an error would halt the execution of a running program.

To reset the system to default error processing, execute:

```
DEFAULT ON
```

With `DEFAULT ON`, the math errors stated above do not halt the execution of a running program.

Part II
BASIC Programming With Your HP-85

Section 5

Simple Programming

If you have read the Getting Started section of this handbook, you've already seen that by using the programming capability of your HP-85, you save hours of time in long computations.

With your HP-85 Personal Computer, Hewlett-Packard has provided you with a Standard Pac containing 15 programs already recorded on a magnetic tape cartridge. You can begin using the programming power of the HP-85 by simply using any of the programs from the Standard Pac, or from one of the other Hewlett-Packard pacs in areas like finance, statistics, mathematics, engineering, or linear programming. The growing list of application pacs is continually being updated and expanded by Hewlett-Packard to provide you with a wide variety of software support. For the advanced programmer, Hewlett-Packard will supply plug-in ROMs to give your system additional capabilities and will provide peripheral devices with the necessary interfacing.

However, we at Hewlett-Packard cannot possibly anticipate every problem for which you may want to use your HP-85. In order to get the *most* from your personal computer, you'll want to learn how to program the HP-85 with BASIC programming language to solve your every problem. This part of the *HP-85 Owner's Manual and Programming Guide* introduces the BASIC language, the editing features of the HP-85, and gives you a glimpse of just how sophisticated your programming can become with the HP-85 Personal Computer.

After most of the explanations and examples in this part, you will find problems to work using your HP-85. These problems are not essential to your basic understanding of the computer, and they can be skipped if you like. But we urge you to work them. They are rarely difficult, and they have been designed to increase your proficiency, both in the actual use of the features of your HP-85 and in creating BASIC programs to solve your *own* problems. If you have trouble with one of the problems, go back and review the explanations in the text, then tackle it again.

In programming, there is no uniquely correct program to solve a particular problem. Any solution that yields the correct output is the right one, but we have included *sample* solutions to the problems in appendix F. Thus, you'll have programs to use, modify, and enhance—even if you're a beginning programmer. In fact, when you have finished working through this part and learned all the capabilities of the HP-85, you may be able to create programs that will solve many of the problems faster, or in fewer steps, than we have shown in our illustrations.

One more thing: this handbook has been written under the assumption that most of you have had *some* programming experience. If you have never written a program before, you may wish to become more familiar with BASIC programming through the optional HP-85 BASIC Training Pac. On the other hand, many of you may be quite experienced BASIC programmers, in which case the *HP-85 Pocket Guide, HP-85 BASIC Reference Card,* and appendix D, Glossary and BASIC Syntax Guidelines, will serve you best.

Now let's start programming!

Loading a Prerecorded Program

If you worked through Getting Started (pages 17 through 31), you learned how to create, enter, and record a BASIC program to compute the average of a set of numbers. Now look at a more complex program.

Insert the Standard Pac tape cartridge into the tape drive as we did earlier (page 22), printed side up, open edge toward the computer.

Next, load the Ski Game program from the Standard Pac:

1. Press the (LOAD) key, which displays LOAD on the CRT, or type LOAD.
2. Type the program name, enclosed within quotation marks; in this case type "SKI".
3. Press (END LINE) to execute the LOAD"SKI" command.

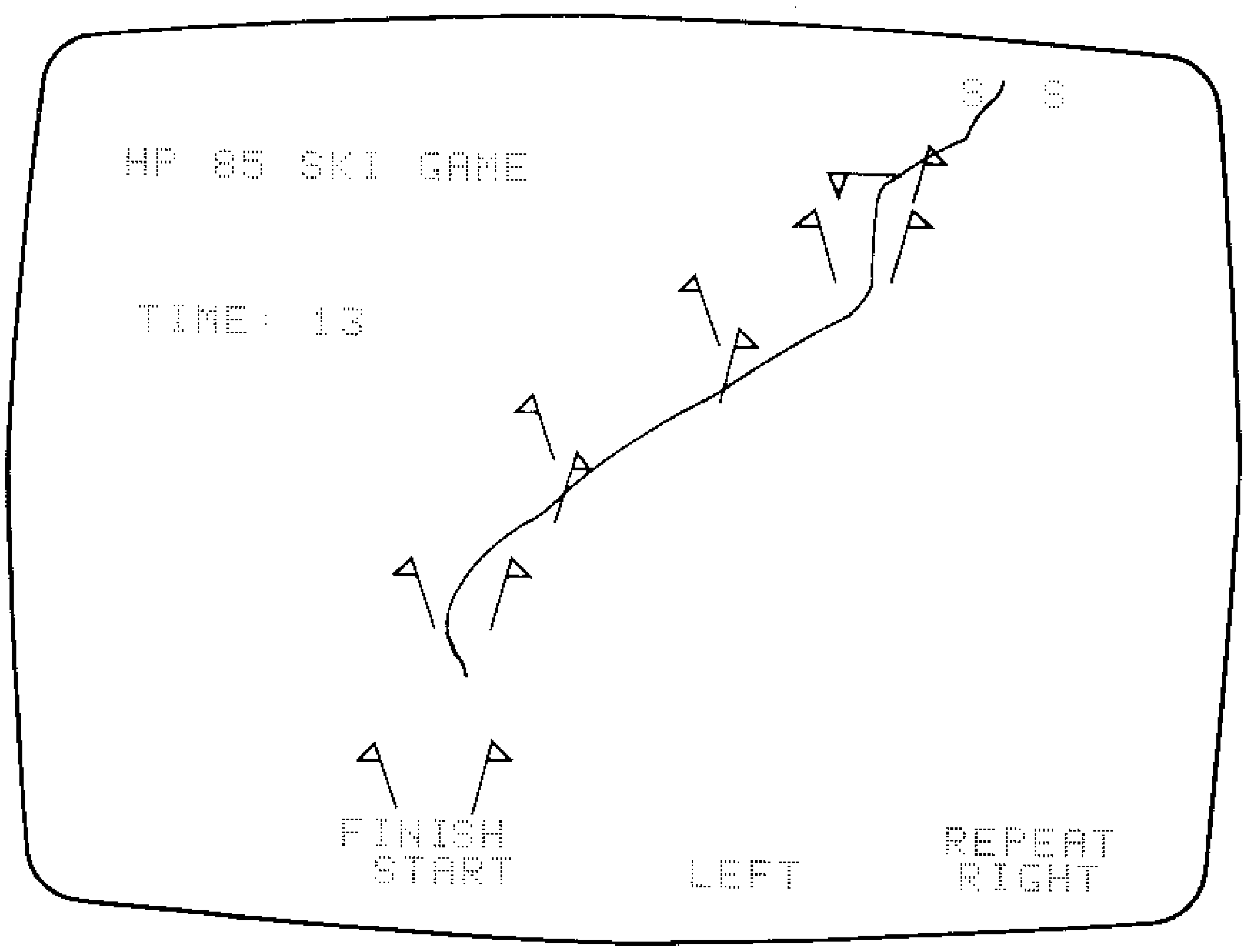
Now the system will search for the Ski Game program and load it from the tape into computer memory. The amber tape drive light will glow while the tape drive is in operation and the display will be turned off. You can easily see when a program has been loaded completely because the cursor will return to the display and the amber tape drive light will stop glowing. Once the Ski Game program has been loaded into computer memory, you can test your "skiing" skills by trying to descend a slalom ski course without missing any of the "gates."

The Game. The Ski Game simulates a skier descending a slalom course, with you in control of the skis. Before you begin your descent, the program asks you whether you wish to ski on a white or a black background, asks you to enter a course code (any number) so that you can ski the same course again or try a different one by specifying a different code, and then asks for your skiing ability. As the game begins, a "skier" comes shooting down the course determined by the flags. You control the direction of the skier by tapping the special function keys (k3) and (k4), labelled LEFT and RIGHT.

The object of the game, of course, is to have a perfect slalom run in record time, without missing any of the gates determined by the flags. (Record time on the most difficult course is about 8.5 seconds.)

After you have loaded the program, press (RUN) and then press (k1) (SET UP) to set up the ski course.

As soon as you press (k2) (START), the game begins. You're on the slope, racing against the clock—you're in control. After each ski run, the HP-85 will display your time and missed gates. Then you can either try the same course again (by pressing the special function key corresponding to REPEAT, (k8)) or specify a new course (by pressing the key corresponding to START, (k6)).



Stopping a Running Program

Remember, *you* are in control of the HP-85. Although the Ski Game program gives you the option of stopping the game, most of the Standard Pac programs will continue to run unless you halt the program.

Stop a running program by pressing the PAUSE key or almost any other key. The program can be continued after it ha been halted by the PAUSE key, by pressing CONT *(continue)*. Pressing almost any other key will halt the program *and* perform the indicated function of the key.

Listing a Program

The HP-85 will give you a listing of any program contained in computer memory at any time, on the display or on printer paper. To see a listing of the Ski Game program that is now loaded in the computer memory, press the LIST key. The LIST key will stop the running program and list the first full screen of the program on the display.

Each successive time that LIST is pressed, another full screen of program lines is displayed until the end of the program is reached. Following the last line of the program, the system displays the remaining number of memory locations.

You can obtain a printed listing of the program by pressing SHIFT P LST *(printer list)*. The program will be listed in its entirety, unless you press any key to halt the listing.

Now list 20 lines or so of the Ski Game program with (P LST); your printout should look like the one shown here.

(P LST)

```
 10 ON KEY# 1,"SET UP" GOSUB 174
    0
 20 ON KEY# 5," HELP " GOSUB 176
    0
 30 ON KEY# 2 GOSUB 1560
 40 ON KEY# 4 GOSUB 1540
 50 ON KEY# 8 GOSUB 1580
 60 ON KEY# 3 GOSUB 1520
 70 LDIR 0 @ CLEAR @ KEY LABEL @
     DISP "SKI GAME"
 80 DIM F(10,2),G(10,2),M(10),P$
    [10]
 90 V9=-1
100 F=0
110 IF NOT F THEN 110
120 IF F=1 THEN 1600
130 V9=-1
140 W=0 @ B=1 @ CLEAR @ DISP "EN
    TER BACKGROUND COLOR:0=W,1=B
    ";
150 INPUT V8
160 SCALE 0,255,0,191
170 IF V8 THEN V9=1
180 DISP "ENTER COURSE CODE"
190 INPUT S1
200 DISP "WHAT'S YOUR ABILITY:1
    TO 5       (1 IS EASY,5 IS HA
    RD)"
210 INPUT Q
220 IF Q<1 THEN 200
230 RANDOMIZE S1*.6142332571
```

(PAUSE)

The printer lists the program exactly as it appeared on the display except that the second and third lines of longer statements (like 10, 20, 80, 140, 200, etc.) appear indented under the first line of the statement, for greater readability. Also, blank lines are inserted every 60 lines (to the nearest complete line) for cutting to place lists in 11-inch notebooks.

What Is a BASIC Program?

A program is an organized set of instructions that tells the computer to accomplish certain tasks. Once a program has been written and loaded into computer memory, it can be executed as many times as you wish—usually at just the touch of the (RUN) key.

Statements

The instructions in a BASIC program are called statements. If you look at the Ski Game listing, you'll see that each statement (except assignments statements like `F=0` or `V9=-1`) contains one or more *keywords* which have a special meaning in BASIC. They identify operations to be performed *(executable statements)* or give the computer information it will need to execute other statements *(declaratory statements)*.

Here are some examples of BASIC keywords:

Executable	**Declaratory**
PRINT	DIM
DISP	COM
IF...THEN	REAL
INPUT	SHORT
READ	INTEGER
FOR	REM
NEXT	DEF FN
GOTO	FN END
GOSUB	DATA
RETURN	IMAGE
LABEL	ON KEY#
CLEAR	
BEEP	

Most executable statements can be executed from the keyboard without a statement number. Exceptions are noted.

Statement Numbers

Every statement in a program must be preceded by a unique statement number. These statement numbers can be seen on the left side of the Ski Game program listing, beginning with 10 and in increments of 10. However, statements may be numbered by any integer from 1 through 9999.

Statements are stored, by number, in ascending order. But you can type them in any order because statements are automatically sorted as they are entered.

Normal program execution proceeds from the lowest numbered statement to the highest numbered statement. The order of execution can be altered, however, as we'll see in sections 7 and 9.

The END statement should be the highest numbered statement in a program. It not only tells the computer *where* a program ends, but also *terminates* program execution. (You may also use the STOP statement; on the HP-85 both END and STOP perform exactly the same function. END is provided on the HP-85 for compatibility with other BASIC systems.)

Commands

A command is an instruction to the computer that is executed from the keyboard. Commands are used to manipulate programs and for utility purposes, such as listing programs and rewinding the tape. Most often, commands are not used in programs. But the HP-85 will allow you to program certain commands. (Refer to table below.)

Probably the two most important commands are SCRATCH and RUN. The SCRATCH command erases program memory and the RUN command starts executing the current program in memory. Both of these commands may be executed by pressing the key with the respective label or by typing the command name and pressing (END LINE). When you press the (RUN) key, that command is executed immediately. But when you type the command name, or press (SCRATCH) (which is a typing aid to display SCRATCH), it won't be executed until you press (END LINE).

Here is a table of the system commands. They are discussed in appropriate places throughout this manual.

Non-Programmable	Programmable
AUTO	CAT
CONT*	COPY*
DELETE	CTAPE
INIT*	ERASETAPE
LOAD	FLIP
REN	LIST*
RUN*	PLIST*
SCRATCH	PRINT ALL
STORE	REWIND*
UNSECURE	SECURE

Clearing Computer Memory

When you loaded the Ski Game program, the program was copied from the tape into computer memory. Before you key in a new program, you will first want to clear, or erase, the Ski Game program from the computer memory.

To clear a program from computer memory, you can either:

1. Press (SCRATCH) (END LINE) or type SCRATCH (END LINE). The SCRATCH command deletes the current program and all variables from computer memory.
2. Load another program from a magnetic tape cartridge. When you load a program into computer memory with the LOAD command, the system automatically clears computer memory before the new program is loaded.

Of course, whenever the system is turned off, it loses all contents of computer memory.

Now you are going to write your own program into the computer from the keyboard, so press (SCRATCH) (END LINE) to clear the HP-85 of the previous program.

Writing a Program

In Getting Started you created, entered, and ran two BASIC programs: a Pythagorean Theorem program, and an averaging program. In this section, we'll create, load, and run another program to show you how to use some of the features of the HP-85.

Before we do this, we'll define the conventions we use to describe program statements and system commands.

Conventions

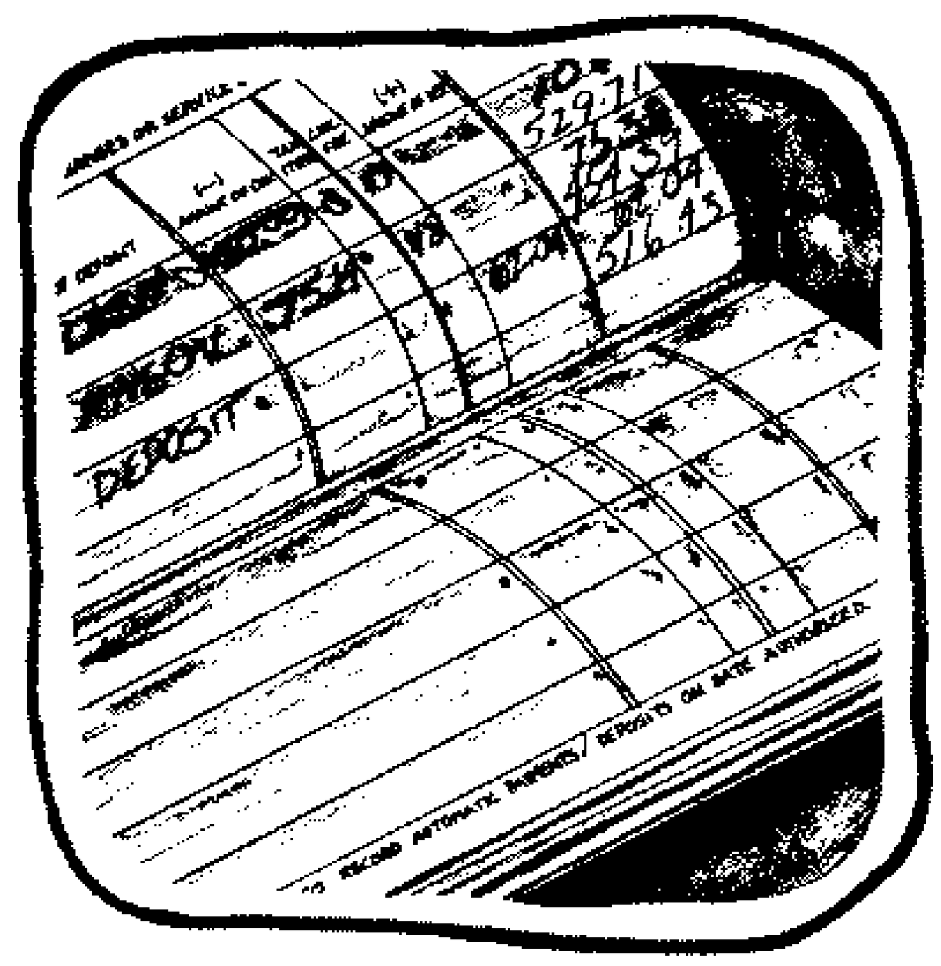
DOT MATRIX	All items in dot matrix must be typed exactly as shown; but you may use either capital letters or small letters in statements or commands.
[ ]	Items within square brackets are optional unless the brackets themselves are in dot matrix.
...	Three dots indicate that the previous item can be repeated.

For example: INPUT variable name [, variable name ...]

* The keys corresponding to these commands are immediate execute keys; i.e., when you press the key, that command is executed immediately.

The statement above tells us that INPUT must be spelled as shown (but you can use either capital letters or small letters) and at least one variable name must be specified with the INPUT statement. The information within the brackets tells us that more variable names can be specified and, if they are, they must be separated by commas since the comma is in dot matrix.

Now let's write a program to keep track of a checkbook balance. Remember that in order to write a program, you must first *define the problem* thoroughly. It may help to ask yourself the following questions:



1. What answer(s) do I want?
2. What information do I know already?
3. What method will I use to find the solution from what I know?
4. How can BASIC and the HP-85 help me solve the problem?

Let's answer these questions for a simple checkbook balancing program.

1. You want to find the balance of the checking account after each check or deposit.
2. You already know the initial balance and the amount of each check or deposit.
3. You must subtract the amounts of the checks and add the amounts of the deposits to the balance.
4. Here's a sample BASIC program, as it would appear listed on printer paper.

```
 10 REM **CHECKBOOK BALANCE**
 20 REM B is the balance.
 30 REM A is the check or deposi
    t amount.
 40 DISP "Initial Balance"
 50 INPUT B
 60 DISP "Check(-) or Deposit(+)
    Amount"
 70 INPUT A
 80 LET B=B+A
 90 PRINT B ! Print new balance
100 GOTO 60
110 END
```

This is only one way to solve the problem. Can you think of other programs that would accomplish the same tasks?

Again, you can see that each statement is preceded by a number and each statement begins with a *keyword* which identifies the type of statement. For example, this program contains 11 statements: three REM *(remark)* statements, two DISP *(display)* statements, two INPUT statements, one LET *(assignment)* statement, one PRINT statement, one GOTO statement, and one END statement.

These keywords will be discussed individually after we execute the program.

Entering a Program

Before you enter the program, let's examine three facets of entering program statements into computer memory: automatic numbering, the spacing in program statements, and the use of the (END LINE) key.

Automatic Numbering

The AUTO command enables statements to be numbered automatically, as they are entered and stored, saving you the time of typing the numbers yourself.

AUTO [beginning statement number [, increment value]]

To cut your typing time further, the AUTO command is provided as a typing aid with the AUTO key; when you press AUTO, the word AUTO appears in the display. Then you can specify the beginning statement number and the increment value. If neither parameter is specified, executing AUTO causes statement numbering to begin with 10 and be incremented by 10 as statements are stored. If only the beginning statement number is specified, numbering begins at that number and is incremented by 10 as program statements are stored. Press END LINE to execute the AUTO command. For example, executing:

```
AUTO 100,5
```

Causes numbering to begin with 100 and increment by 5.

To stop the auto numbering, backspace over the unwanted numbers and type NORMAL END LINE . Auto numbering will also be halted by any executable statement or command without a number. For instance, if, after you enter the program, you wish to run it immediately, simply press the RUN key.

Spacing

In general, spacing between characters is arbitrary; the HP-85 automatically sets proper spacing into each statement of a program whenever the program is listed.

Blanks are ignored in BASIC statements except when enclosed in quotes or when contained in remarks. When the HP-85 formats a statement, blanks are inserted or deleted so that all keywords are surrounded by a blank on either side.

For example:

```
100LETA=B*C
100 LET A = B * C
1   00LE     TA=    B*  C
```

All of the above statements are equivalent and would appear in a listing as:

```
100 LET A=B*C
```

The only place that a blank space should not be typed is immediately after the first letter of a keyword. If you attempt to enter the statement, the system will interpret the first letter as a variable name and will give you an error message. For example:

```
100 L ET A=B*C
Error 92 : SYNTAX
```

Statement Length

As we mentioned earlier (page 37), program statements can be up to 95 characters long including the line number—that's three full lines on the video display minus one space to press (END LINE).

But if you "pack" your statements by deleting all spaces between characters, be sure to take into account that the system will automatically insert spaces around keywords when the statement is listed or edited—the statement may be too long to be edited and reentered.

(The system will give you an error message if it does not understand.)

Entering Program Statements into Computer Memory

Program statements are entered into computer memory in the same way that any executable keyboard operation is entered, by pressing (END LINE). You must press (END LINE) after each program statement has been typed in. Pressing (END LINE) also causes the statement to be checked for syntax errors before it is stored. Should an error occur on entering a statement, simply correct or retype the statement, then reenter it. Refer to section 6, Program Editing.

In a long statement that requires more than one line, do not press (END LINE) until the statement is completely typed in; the system display will automatically wrap around onto the next line. Press (END LINE) only to enter a complete program statement into computer memory.

For example:

`10 PRINT "After you type a compl`	Do not press (END LINE) here ...
`ete program statement, you must`	... or here ...
`enter it by pressing ENDLINE."` (END LINE)	... but here.
`20 END` (END LINE)	And here.

Entering the Program

You can enter a program into computer memory in either of two ways:

1. By retrieving a copy of a previously stored program from magnetic tape cartridge.
2. By typing the program statements, including statement numbers, one at a time from the keyboard, pressing (END LINE) after each statement. (Remember, you don't need to type the statement numbers if you use `AUTO` line numbering.)

Since we do not have a program on tape that keeps track of a running checkbook balance, we will use the second method to enter our program.

If you have not already done so:

1. Press (SCRATCH) (END LINE) to erase previous programs from computer memory.
2. Press (CLEAR) to clear the display. This is not a necessary step to writing a program, but it will increase the legibility of the display.
3. Now, press (AUTO) (END LINE) since we want to take advantage of the automatic numbering system.

Finally, enter the checkbook balancing program by typing each statement shown in the sample program, pressing (END LINE) after each statement. When you have finished entering the program, the display will look like this:

```
10 REM **CHECKBOOK BALANCE**
20 REM B is the balance.
30 REM A is the check or deposit
 amount.
40 DISP "Initial Balance"
50 INPUT B
60 DISP "Check(-) or Deposit(+)
Amount"
70 INPUT A
80 LET B=B+A
90 PRINT B ! Print new balance
100 GOTO 60
110 END
120 _
```

The program for keeping track of a checkbook balance is now loaded into computer memory. Notice that AUTO statement numbering caused the number 120 to appear below the END statement of the program. Simply backspace over 120 to erase the number, and type NORMAL (END LINE) to stop AUTO statement numbering.

Running a Program

To run a program, you have only to press the (RUN) key.

For example, use the program now in computer memory to balance a checkbook with an initial balance of $1,004.25; checks written for the amounts of $14.53, $25.00, and $18.90; and deposits in the amounts of $52.50 and $120.00

(RUN)

Press (RUN) to start program execution.

```
Initial Balance
?
1004.25
Check(-) or Deposit(+) Amount
?
-14.53
Check(-) or Deposit(+) Amount
?
-25
Check(-) or Deposit(+) Amount
?
-18.90
Check(-) or Deposit(+) Amount
?
52.5
Check(-) or Deposit(+) Amount
?
120
Check(-) or Deposit(+) Amount
?
```

When a question mark appears, key in the balance and press (END LINE). Then enter the checks as negative numbers and the deposits as positive numbers. Press (END LINE) after the amount to enter the data to the program.

(PAUSE)

Press (PAUSE) to halt program execution.

Where is the record of the checkbook balance? You'll find that the printer has recorded the following on paper. (Press the PAPER ADV key to advance the paper if necessary.)

989.72	Initial balance − 14.53.
964.72	New balance − 25.00.
945.82	New balance − 18.90.
998.32	New balance + 52.50.
1118.32	New balance + 120.00.

Now let's see how the HP-85 executed this program.

Order of Program Execution

Statements are executed in order of ascending statement numbers.

```
10
:
:
60
:
:
100 GOTO 60
```

When you pressed RUN, the HP-85 began executing instructions sequentially by statement number starting with statement 10. When it reached statement 100 GOTO 60, the system returned to statement 60 and executed successively higher-numbered statements from there. The program continued to run until you pressed PAUSE to halt the program.

Fundamental BASIC Statements

Now let's examine the statements that composed our checkbook balancing program.

REMarks

Many times you may want to insert comments in order to make your program logic easier to follow. This can be done by using the REM *(remark)* statement or !, the comment delimiter.

REM [any combination of characters]

In our sample program, remarks are used to remind us that the variables A and B stand for amount of a check or a deposit and the checkbook balance, respectively:

```
10 REM **CHECKBOOK BALANCE***
20 REM B is the balance.
30 REM A is the check or deposit
   amount.
```

The comment delimiter, !, can be anywhere in a program statement after the statement number. All characters following a ! are considered part of a comment unless the comment delimiter, !, is within quotes.

In this way, program statements can contain comments. For instance, statement 90 in our sample program:

```
90 PRINT B ! Print new balance.
```

Comments, as you have seen, are useful only in a program listing. They do not affect program execution.

DISPlay

The `DISP` *(display)* statement allows text and variables to be output on the display.

`DISP` [display list]

The display list can contain variable names, numeric expressions, quoted text or messages, and the `TAB` function (covered later). These items must be separated by commas or semicolons.

In the checkbook balancing program, the following `DISP` statements appear:

```
40 DISP "Initial Balance"
:
60 DISP "Check(-) or Deposit(+)
   Amount"
```

As you have seen, when these statements are executed in a running program, they display, respectively:

```
Initial Balance
:
Check(-) or Deposit(+) Amount
```

You can combine quoted messages with variable names, but they must be separated from each other with commas or semicolons. For example:

```
N=5
S=175.60
DISP "THE AVERAGE OF THE";N;"NUM
BERS IS";S/N
THE AVERAGE OF THE 5 NUMBERS IS
 35.12
```

When these statements are executed ...

... this message is displayed.

What is the difference between using commas or using semicolons to separate items in a display list? Look at the following examples:

```
DISP 111,222,333,444,555,666
 111                  222
 333                  444
 555                  666
```

Commas cause wide spacing between display list items.

```
DISP 111;222;333;444;555;666
 111  222  333  444  555  666
```

Semicolons space items close together.

Notice the difference in spacing between the items. When an item is followed by a comma, the next item will be left-justified at either column 1 or column 22 on the display. Remember, every number has a leading blank or a minus sign and a trailing blank for spacing. If a number contains over nine digits and would start in column 22, it will be displayed in the first column of the next line.

When an item is followed by a semicolon, no additional blanks are inserted. For example:

```
DISP 100; -20; 77.3
 100 -20  77.3
```

All numbers are displayed with a leading blank or minus sign and a trailing blank for spacing.

Two or more commas after an item cause one or more character fields to be skipped.

For example:

```
DISP ,100,,200
                           100
                           200
DISP 100,,200
 100
 200
```

When a DISP statement appears without a display list, a blank line is displayed. For example:

```
 10 DISP
 20 DISP
 30 DISP "*********************"
 40 DISP "* SQUARINUT SQUBISE *"
 50 DISP "*********************"
 60 DISP
 70 DISP
 80 DISP "Peel and mince 3 large
     onions and set them aside."
 90 DISP "Slowly add 4 Tbsp. flo
    ur to 1/4 cup squarinut oil.
    "
100 END
```

When this program is executed, the following would be displayed.

```
*********************
* SQUARINUT SQUBISE *
*********************


Peel and mince 3 large onions an
d set them aside.
Slowly add 4 Tbsp. flour to 1/4
cup squarinut oil.
```

When the display list ends with a comma or semicolon, any future DISP statement output is appended to the current display line. For example:

```
10  DISP "ENTER DATE"
20  INPUT D1$
30  DISP "TODAY IS ";
40  DISP D1$
50  END
```

When these statements are executed in succession in a program ...

```
ENTER DATE
?
JUNE 1
TODAY IS JUNE 1
```

... and you enter JUNE 1 for the date ...
... this message is displayed.

The semicolon at the end of the statement 30 causes the message "TODAY IS" to be held in a special disp/print buffer. The buffer does not display (or with PRINT, print) its contents unless:

- Another DISP statement without a semicolon at the end of the message causes it to be output.

- An INPUT statement causes the buffer contents to be displayed or printed (as we'll see later).
- The buffer is filled with 32 characters, in which case it is automatically output.

For instance, if statement 40 also ends with a semicolon (in the example above), an extra DISP statement is required to output the message:

```
10  DISP "ENTER DATE"
20  INPUT D1$
30  DISP "TODAY IS " ;
40  DISP D1$;
50  END
```

If you run this program, the input prompt will be displayed and nothing else *appears* to happen.

```
ENTER DATE
?
JUNE 1
DISP
TODAY IS JUNE 1
```

Enter the date.
If you now execute DISP, the message will be displayed.

The extra DISP statement could also have been part of the program between statements 40 and 50.

PRINT

The PRINT statement allows text and variables to be printed by the HP-85's internal printer.

PRINT [print list]

Like the display list, the print list can contain variable names, numeric expressions, string expressions and quoted text, and the TAB function. All items must be separated by commas or semicolons.

Here are some examples:

```
PRINT 20;81.1569;32.9
PRINT "HYPOTENUSE=";5
PRINT "***!!!";"///^^^";"@@@***"
PRINT
PRINT
PRINT "***!!!","///^^^","@@@###"
PRINT
I,J,K,L,M=5
PRINT I,J,K,L,M

 20  81.1569  32.9
HYPOTENUSE= 5
***!!!///^^^@@@***


***!!!                  ///^^^
@@@###

 5                       5
 5                       5
 5
```

Notice that commas and semicolons perform in the PRINT statement just like in the DISP statement. A comma after an item causes the next item to be left-justified in either column 1 or column 22. A semicolon after an item suppresses additional blanks. Also note that when nothing follows the word PRINT in a statement, the paper advances one line. For more information about displaying and printed output, refer to section 10, Printer and Display Formatting.

INPUT: Assigning Values From the Keyboard

The INPUT statement allows values in the form of expressions to be assigned to variables from the keyboard at the request of a program. The INPUT statement is programmable only; it can't be executed from the keyboard.

INPUT variable name_1 [, variable name_2 ...]

As we have seen, when the INPUT statement is executed, a question mark (?) appears on the display. A value can then be input for each variable designated in the INPUT statement.

Remember our first example in section 1 (page 27):

```
60 INPUT L,W
```

The program called for the lengths of the sides of a right triangle.

When the program was executed, and the question mark appeared on the display, we input both values, separated by a comma, in one line like this:

```
?
7.5, 10 [END LINE]
```

Separate INPUT values with commas.

If we had tried to enter the values for the variables L and W one at a time, we would have received an error message, followed by another question mark and we'd have another chance to enter *both* values. An INPUT statement requests all values for the variables specified to be entered at the same time.

Values for string variables can be quoted or unquoted, but an unquoted string cannot contain a comma (since commas separate input items).

Let's look at some examples of entering strings:

```
60 DISP "YOUR NAME";
70 INPUT N$
80 DISP "MY NAME IS"; N$
```

When these lines are executed, the display shows:

```
YOUR NAME?
```

Now you can input up to 18 characters of your name in either of two ways;

Without quotation marks:

```
YOUR NAME?
HP-85
MY NAME ISHP-85
```

Since you did not leave any trailing blanks in DISP statement 60, then DISP statement 80 packs the characters together.

With quotation marks:

```
YOUR NAME?
" HP-85!"
MY NAME IS HP-85!
```

Use quotation marks if you wish to preserve leading or trailing blanks or use commas in your expression.

Whenever you assign character string expressions to string variables from the keyboard, you can use quotation marks at your option. Just remember that strings do not contain leading or trailing blanks unless you specify them explicitly with quotation marks.

Also notice where the question mark appeared in the examples above. If you place a semicolon after a message in a DISP or PRINT statement before an INPUT statement, the semicolon suppresses the carriage return so that the question mark appears on the same line as the message.

Thus, we could have written our checkbook balancing program like this:

```
40 DISP "Initial Balance";
50 INPUT B
60 DISP "Check or Deposit Amount";
```

Before an INPUT statement, a semicolon at the end of a DISP statement (or PRINT statement) suppresses the carriage return.

If this section of the program were executed, it would display:

```
Initial Balance?
:
Check or Deposit Amount?
```

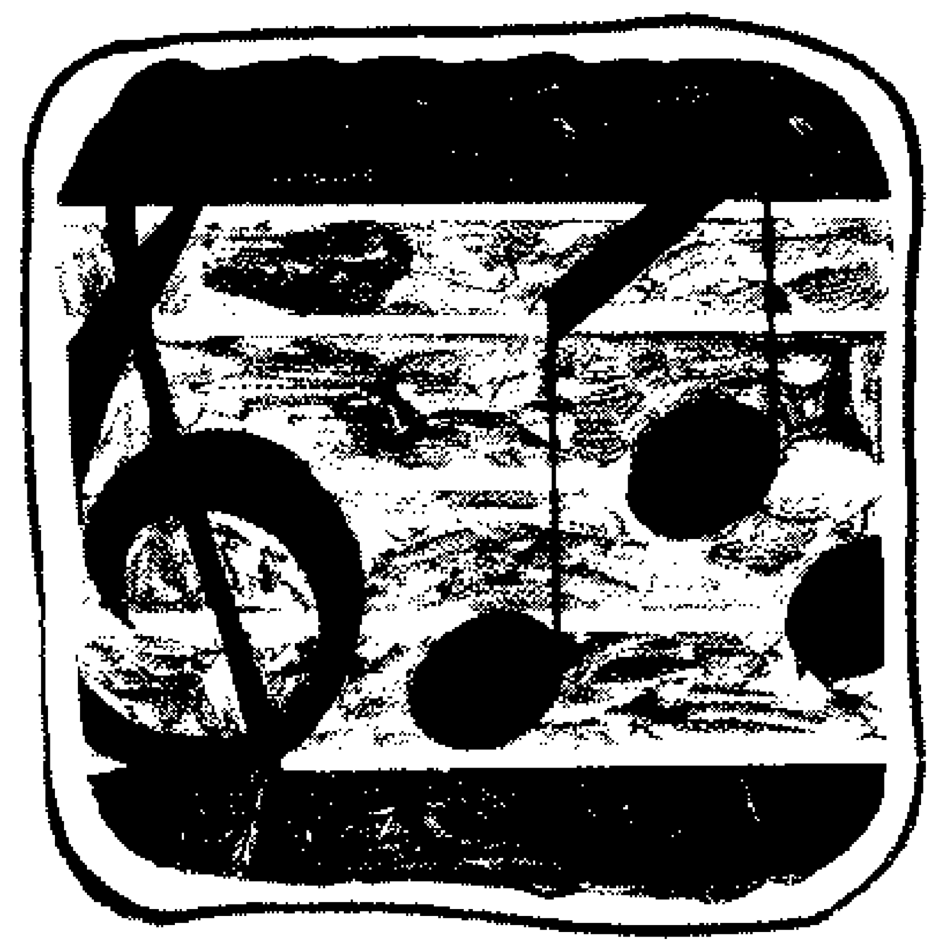
Question marks are on the same line instead of beneath the displayed line.

Pressing (END LINE) without entering values when numeric input is requested causes an error. You may input the null string ("") as response to a string input request.

BEEP

The BEEP statement can be considered a comment or an output statement. The BEEP statement is used to produce an audible tone of variable frequency and duration that can be used in a number of ways.

BEEP [tone , duration]



BEEP can signal that a particular computation or program segment is complete. It can be used to indicate audibly that the computer is ready for input, so that the operator does not have to remain at the keyboard. And, of course, it can be used for the sound itself; load the COMPZR program from the Standard Pac—you can actually compose "music" with the BEEP statement.

If no parameters are specified, the frequency is approximately 2000 hertz, and duration is 100 milliseconds. By specifying parameters, you can change the tone and the duration.

For example, we used BEEP in the hypotenuse program as an audible input prompt:

```
10 DISP "ENTER SIDE LENGTHS"
20 DISP "OF A RIGHT TRIANGLE,"
30 DISP "SEPARATED BY A COMMA."
40 DISP "THEN PRESS END LINE."
50 BEEP
60 INPUT L, W
70 D= SQR(L^2+W^2)
80 PRINT "HYPOTENUSE =";D
90 END
```

BEEP signals the operator for input.

The BEEP statement can be executed from the keyboard. For example, try several different values for tone and duration by executing the following statements. You can stop the sound at any time by pressing STEP PAUSE.

```
BEEP
BEEP 10, 50
BEEP 200, 30
BEEP 50, 100
```

The value for the tone and duration of BEEP can be a numeric expression. *Both parameters are rounded to integer values with the BEEP statement.* For example, run the following program to generate "random" music. (We discuss the FOR and NEXT statements in section 7). You can stop the program at any time by pressing PAUSE.

```
10 FOR I=1 TO 250
20 BEEP I*RND+1, 50
30 NEXT I
40 END
```

This program generates a "random" sequence of 250 audible tones.

Use the following formulas to compute BEEP parameters that produce a particular frequency and duration:

Tone (first parameter):

P1 = 613062.5/(11*F)-134/11 where F is desired frequency in hertz.

Duration (second parameter);

P2 = T*613062.5/(11*P1+134) where T is desired duration in seconds. (Or, simply, P2 = T*F when F is known.)

For example, to BEEP for approximately one-half second at a frequency of about 440 hertz, compute P1 and P2 as follows:

```
P1 = 613062.5/(11 * 440)-134/11
P1
 114.483987603
P2 = .5 * 440
P2
 220
```

So, the BEEP statement would be:

```
BEEP 114,220
```

Beeps at approximately 440 hertz for approximately 0.5 second.

LET: Assignments

Any numeric variable can be assigned a value using an assignment statement as we have seen in section 3. String variables can also be assigned string expressions using the assignment statement if the expression produces a string shorter than or equal in length to the size of the string variable. The keyword in an assignment statement is LET, but its use is optional.

[LET] simple variable$_1$ [, simple variable$_2$...] = numeric expression
[LET] string variable$_1$ [, string variable$_2$...] = string expression

The keyword LET is a reminder that the variable name is always to the left of the equals sign and the expression assigned to that variable is always to the right of the equals sign; you are "letting" the variable be changed to equal the value of the expression.

For example, the following statements are equivalent:

```
X=12
LET X=12
X = 3*4
```

The following statements using string variables are equivalent:

```
A$, B$="BUTTERFLY"
LET A$,B$= "BUTTER" & "FLY"
```

Remember that the string expression must be enclosed within quotes in a string variable assignment statement.

To check the current value of a variable, type in its name, then press (END LINE). For instance, using the above values for the variables:

```
X
  12
A$
BUTTERFLY
B$
BUTTERFLY
```

Pressing (END LINE) after the variable name yields its current value.

If a numeric variable is used in a computation and hasn't been assigned a value, a warning message is displayed and 0 is used as its value. Likewise, if a string variable is used before being assigned a value, a warning message is displayed and the null string is used as its value. In general, it is good programming practice to initialize variables (by initialize, we mean assign them their initial values) at the beginning of the program, as we did in the averaging program (page 29).

GOTO: Unconditional Branching

In our checkbook balancing program, you saw that the GOTO statement transferred program control back to the specified statement. This is known as an unconditional branch. GOTO statements are programmable only; they can't be executed from the keyboard.

GOTO statement number

If the specified statement is not an executable statement (e.g., a REM statement), control is transferred to the first executable statement following that statement.

As you may remember from the checkbook balancing program, the use of the GOTO statement caused the program to "loop" endlessly from statements 60 through 100:

```
60 DISP "Check(-) or Deposit(+) 
Amount"
70 INPUT A
80 LET B=B+A
90 PRINT B
100 GOTO 60
```

Branches to statement 60.

But we also saw that it is easy to stop the program—by pressing (PAUSE).

GOTO statements may branch to both higher numbered and lower numbered statements; for example:

```
10 X=5
20 GOTO 50
30 DISP "NEW X-VALUE";
40 INPUT X
50 PRINT "X EQUALS";X
60 GO TO 30
```

Branches to statement 50.

Branches back to statement 30.

The GOTO statement is the most simple form of branching.

Multistatement Lines

A symbol that you may have seen in the Ski Game program listing is the "at" symbol (@). The @ symbol enables you to type more than one statement on the same program line, thus shortening program listings and conserving memory. Remember you still cannot enter more than 95 characters (including the statement number) at a time.

Examine line 70 in the Ski Game program listing:

```
70 LDIR 0 @ CLEAR @ KEY LABEL @
     DISP "SKI GAME"
```

In the program line above, four statements have been joined together on the same program line, using the same statement number. The program could have been written like this:

```
70 LDIR 0                    Sets label direction.
73 CLEAR                     Clears display.
75 KEY LABEL                 Recalls key labels.
77 DISP "SKI GAME"           Displays message.
```

But, by using the @ symbol, the program was shortened by three program lines (nine bytes).

There are several things you must be careful about when you type multiple statements using the same statement number.

- If there is a `GOTO` statement in a multistatement line, it should be the last statement. For instance:

  ```
  20 GOTO 50
  .
  .
  .
  50 GOTO 60 @ PRINT "MISSED"
  60 END
  ```

  In order to reference the print statement in line 50, the statements need to be reversed; otherwise, the message will not be printed.

- If you join statements that involve relational tests or "decision-making" operations (like `IF ... THEN`) be sure that you are aware of what happens when the test comes up "true" or "false." For instance if you join a statement at the end of an `IF ... THEN` statement, that statement will not be executed if the relational test is false. This can be a definite programming advantage, but there may be times when you forget this fact and wonder why your program doesn't work the way you think it should.

- Declarations (such as `DIM`, `COM`, `REAL`, `SHORT` and `INTEGER`) can be made in multistatement lines but they must be the last statement in the line.

- Anything that follows `REM` or `!` is a remark. The following multistatement line may look good, but `D` will never be computed!

  ```
  50 R = 4.5 ! RADIUS @ D = 2*R
  ```

- Programs that have multistatement lines using the @ symbol may not be transportable to other computers using BASIC.

- Care should be taken to preserve readability with multistatement lines. For instance, `CLEAR @ KEYLABEL` is easily read and understood on one line. But it is possible to destroy readability by packing too much into a line. Readability is important, particularly with debugging procedures and documentation of your program.

Problems

5.1a. Write a program to convert a temperature in Celsius degrees to Fahrenheit according to the formula $F = 1.8*C + 32$. Use a LET assignment statement for the conversion calculation. The program should ask for the original Celsius temperature and label the corresponding Fahrenheit temperature.

b. Write a second program to convert a temperature in Fahrenheit degrees to Celsius. The equation is $C = 5/9*(F-32)$. Do not use the optional keyword LET in this program. Be sure to include an input prompt and an output label.

5.2 Janey Dair enjoys dropping her new Rebounder ball from the window of her room, delighting her friends who watch it bounce on the pavement below. Each rebound reaches a height equal to 65% of its previous height. Write a program that requests the height from which she drops her Rebounder, and displays the cumulative distance it has traveled each time it touches the pavement. Use a BEEP to represent each bounce prior to displaying the distance. The program should continue calculating the distances until it is manually interrupted with the PAUSE key. Observe the output to be sure that the total distance traveled approaches a limit, rather than increasing indefinitely.

5.3 In preparation for writing your first novel, you want to use the HP-85 to help you choose an interesting title. You decide to write a program that takes a noun and a proper noun as input, and prints two titles using the following forms:

THE (noun) OF (proper noun)
TO (proper noun) WITH THE (noun)

You may not win any literary awards, but you'll get some interesting titles.

5.4 World-famous jazz artist Bertha Blues wants to program the HP-85 to play a particular bass rhythm as an accompaniment for a work session. The rhythm consists of the repeated sequence of notes C130.81,G196, G98,G196 at 120 beats per minute (0.5 seconds per note). (The numbers following the notes specify the frequency in hertz.) Write a program that computes and prints the tone and duration parameters for the three appropriate BEEP statements and then breaks into its rhythmical rendition.

5.5 The factorial function ($x!$) is defined for positive integer values of x as

$$x! = x(x-1)(x-2)\ldots1.$$

An algebraic approximation is given by the equation

$$x! = e^{-x}x^{x}\sqrt{2\pi x}.$$

Write a program that, for any positive integer value of x, calculates and prints the $x!$ approximation using this method. (In section 7 you will see an easy method to compute the *exact* factorial function.)

5.6 During his spare time, Artemas Horologos repairs watches in his home workshop. He has decided that a program that calculates his bill would be very helpful. Write a program that requests the customer's last name, the number of hours Artemas has worked on a watch, and his cost for replacement parts. It should then print an individualized repair bill itemizing the charges for parts, for labor, and the total amount due. Artemas charges $8.50 per hour for his labor, and charges 10% more than his cost for parts.

Program Editing

Often you may want to alter or add to a program that is already loaded into computer memory. The HP-85 has been designed to make program editing as fast and easy as possible.

In this section, we will discuss program modification by adding, deleting, and editing program statements. And we'll introduce specific program editing commands to delete blocks of program statements, list specific parts of a program, and automatically renumber a complete program. Finally, we'll show how to interrupt the execution of a running program and how to continue execution at a specified statement.

Editing Program Statements

Edit program statements in the same way that you edit anything that appears on the display—with the display editing keys.

There are two ways to edit and change a statement that is already in computer memory.

1. Recall the program into the display by using the (ROLL) key or by listing the program on the display. Then using the display editing keys and cursor control keys, move the cursor to the desired statement, make the necessary changes in the program statement, and press (END LINE) to enter the changed statement into memory.
2. Retype the statement, including statement number, incorporating all the changes you wish to make. Then press (END LINE) to enter the statement into memory.

Remember, you can enter program statements in any order—the computer automatically sorts them by statement number as they are entered. The last statement entered with a given statement number is the one that is used in the program. When you edit a line or statement on the display, always check to see that there are no unwanted characters beyond the last character in the statement. If there are, move the cursor to the end of the good line and press (-LINE) to delete the unwanted characters before you press (END LINE) to enter the program statement.

Deleting Statements

You can delete program statements in either of two ways:

1. To delete an individual statement of a program, type the statement number and then press (END LINE).
2. To delete a section of a program, it is quicker to use the DELETE command.

The DELETE command is used to delete a statement or a block of statements from a program.

DELETE first statement number [, last statement number]

The DELETE command is provided on a key as a typing aid. When you press the (DEL) key, DELETE is displayed.

If only one statement number is specified with the DELETE command, then only that program statement will be deleted from program memory. If you specify both statement numbers, then that section of a program will be deleted.

Examples:

`30` (END LINE)	Deletes statement 30.
`DELETE 40` (END LINE)	Deletes statement 40.
`DELETE 60,90` (END LINE)	Deletes statements 60 through 90, inclusive.

Adding Statements

Add new statements to a program merely by typing and entering them into computer memory. Be sure that the statement number of a new statement positions it correctly in the program.

Often, it saves a good deal of typing by merely editing a similar statement of your program, changing the statement number, and then entering the new statement into program memory by pressing (END LINE). But don't forget to change the statement number!

Renumbering a Program

The `REN` *(renumber)* command is used to renumber a program that has already been entered into computer memory.

`REN` [beginning statement number [, increment value]]

Just as with the `AUTO` command, you can optionally specify the new starting statement number and the increment between statement numbers. If no parameters are specified, the program is renumbered so that statement numbering begins with 10 and is incremented by 10. If no increment value is given, the statement numbers will be incremented by 10.

Examples:

`REN`	Renumbers a program so that the first statement is numbered 10, and the statements that follow are numbered in increments of 10.
`REN 100`	Renumbers a program, beginning with 100 and incrementing by 10.
`REN 200,5`	Renumbers a program, beginning with 200 and incrementing by 5.

The `REN` command automatically renumbers an entire program, including any branches within a program. But the `REN` command will not change the parameters of the `PLIST` or `LIST` commands when they are included as program statements.

If you have a very large program or you use `REN` in such a way that the computer reaches line 9999 before it renumbers the whole program, then the computer will automatically start at the beginning of the program and renumber by 1, i.e., beginning with statement 1 and renumbering in increments of 1.

Listing a Modified Program

Up to this point, we have discussed two ways to list a program: by using the (LIST) key to list the program on the display, or by using the (P LST) key to list the program on the printer.

But you can also type these commands from the keyboard and then specify the section of a program you wish to have listed.

LIST [beginning statement number [,ending statement number]]
PLIST [beginning statement number [,ending statement number]]

If you type LIST, and specify one statement number before pressing (END LINE), listing begins with that statement and continues for one screen. If two statement numbers are specified, that section of statements between and including the two numbers is listed.

If you type PLIST and specify one statement number before pressing (END LINE), the program will be listed on the printer from that statement number to the end of the program (or until you press a key). If two statement numbers are specified, that section of the program is listed on the printer.

If you type either command and specify no statement numbers, and then press (END LINE), the command will be executed as if you had pressed either the (LIST) or the (P LST) key.

Examples:

`LIST 40,90`	Lists statements 40 through 90 on the display.
`LIST 90`	Lists statements beginning with 90 and continuing for one screen's worth of statements.

If the system cannot find the statement number, it will list the next higher statement up to the last statement number you specify. For instance, if your program is numbered from 10 to 150 in increments of 10:

`PLIST 5,45`	Lists statements 10 through 40 on the printer.

You can list one statement by specifying the same statement number for both parameters. For example:

`PLIST 90,90`	Lists statement 90 on the printer.

Both the LIST and PLIST commands are programmable. However, REN will not renumber programmed LIST or PLIST parameters.

One more function is associated with the LIST and PLIST commands: *following the list of the last program statement, the remaining number of memory locations (bytes) is output.* We'll discuss the system memory in section 8. For now, simply note that the number of the end of an entire program listing gives the available memory.

Interrupting Program Execution

Pausing

We have already seen that pressing (PAUSE) halts the execution of a running program. But actually it just suspends the execution of a running program. When you press (PAUSE) the current statement is completed and the program is paused at the next statement to be executed.

As we shall see, a pause can also be programmed using the PAUSE statement.

Although the specific function of (PAUSE) is to suspend the execution of a running program, pressing any key (except those noted below) will also halt the execution of a running program and perform the indicated function of the key.

For instance, if you press a typewriter key, such as (C), the system finishes executing the current statement, then halts and displays "C."

But if you happen to press (RUN) during the execution of a running program, the current statement is completed, the program is halted, and then the system displays RUN_. If you really want to rerun the program, execute RUN_ by pressing (END LINE). If you do not want to rerun the program press (CONT) to continue (see below).

Note: Whenever a running program is interrupted from the keyboard, the system beeps.

The following keys will perform the indicated functions *without halting the execution of a running program* or otherwise interrupting or disturbing the program:

(COPY)	Copies the current display to the printer.
(PAPER ADV)	Advances the paper in the printer.
(KEY LABEL)	Recalls special function key labels (if any).
(GRAPH)	Sets display to graphics mode.
(CLEAR)	Clears alphanumeric display.
(ROLL)	Rolls display contents up or down.

Continuing

If a program has been halted with the (STEP PAUSE) key or a PAUSE statement, it can be resumed from where it was halted by pressing the (CONT) *(continue)* key or by executing the CONT command. You can press (CONT) or execute the CONT command after almost any other program halt—as long as you have not deallocated the program. (A program would be deallocated if, for instance, you edited the program. You would then need to initialize the program, as we will see on the next page, before continuing.)

CONT [statement number]

The (CONT) key is an immediate execute key. Thus execution of a halted program is immediately resumed when you press the key.

You can continue program execution at a specific statement by typing CONT followed by the statement number and then pressing (END LINE). For example:

CONT 90	Continues program execution at statement 90.

Execution of a paused program can also be restarted at the beginning with (RUN) (or RUN), by executing CONT 0, or by (INIT) (CONT).

Whenever program execution has been paused, you can perform any normal keyboard activities. For instance, you can list the program in memory or perform some arithmetic calculations. And when you press (CONT), program execution resumes from where it paused (unless, of course, you have cleared the program from memory by executing (SCRATCH) or (LOAD)).

Initializing a Program

The RUN key (or RUN command) automatically initializes a program before running it. By "initialize" we mean that the system allocates memory to all program variables, sets (initializes) variables to undefined values, and sets the program pointer to the first statement of the program.

RUN [statement number]

As with the CONT command, you can optionally specify the starting statement by typing RUN followed by the statement number and pressing END LINE. For example:

`RUN 100`	Initializes and then runs a program beginning with statement 100.

If a program has been halted with a PAUSE command, computer memory remains allocated and the program pointer is set to the statement after the one it has just executed. Pressing CONT (or executing CONT) does not allocate or initialize program variables again. Execution merely resumes from where it left off.

If, for instance, you *edit* a program statement after you PAUSE the program, program variables are no longer allocated and the program cannot be continued with CONT or CONT. You must initialize the program and reallocate memory for variables by pressing the INIT key (or by executing the INIT command) before you press CONT. Execution resumes from the beginning of the program—not from where PAUSE halted it.

The INIT command allocates memory to all program variables, initializes variables to undefined values, and resets the program to begin executing from the lowest numbered statement. Using INIT and CONT together performs the same function as RUN.

Using PAUSE in a Program

The PAUSE statement can also be used in a program, as we mentioned earlier. Program execution is halted whenever the PAUSE statement is encountered in a program. The PAUSE statement does not cause the system to beep when it is halted.

Pausing is useful to control program execution. Continue a program halted by PAUSE with CONT (or CONT).

For example, enter the following program:

`10 REM *FUTURE VALUE*`	
`20 N=1`	Set N (number of years) to 1.
`30 DISP "Present Value";`	Ask for input.
`40 INPUT P`	Input present value.
`50 PRINT "Present Value =";P`	Print value.
`60 DISP "Interest Rate";`	Ask for input.
`70 INPUT I`	Input interest rate.
`80 PRINT "Interest Rate =";I`	Print value.
`90 PRINT "Future Value After"`	
`100 F=IP(P*(1+I)^N*100)/100`	Calculate future value truncated to hundredths.
`110 PRINT "Year";N;"is";F`	Print year and amount.
`120 N=N+1`	
• `130 PAUSE`	Pause.
`140 GOTO 100`	Go back to 100.
`150 END`	

Now run the program with a present value of $1000 and interest of 6% (.06).

```
(RUN)
Present Value?
1000                    Present Value = 1000
Interest rate?
.06                     Interest Rate = .06
                        Future Value After
                        Year 1 is 1060
(CONT)                  Year 2 is 1123.6
(CONT)                  Year 3 is 1191.01
(CONT)                  Year 4 is 1262.47
```

Press (CONT) to continue after the PAUSE in statement 130.

Here, PAUSE and CONT enable you to print one line at a time.

Delaying Program Execution

The WAIT statement is used to program a delay between the execution of two program statements.

WAIT number of milliseconds

The WAIT parameter can be any number within the range of the HP-85 but the minimum wait is 0 and the maximum wait is about 27 minutes (1,666,650 milliseconds). A negative number specifies a zero wait. The WAIT statement can be interrupted by (PAUSE) or almost any other key.

For instance, if you changed statement 130 in the Future Value program to:

```
130 WAIT 3000
```

The program would wait 3 seconds (3000 milliseconds) before it printed each future amount.

Error Messages

There are three types of errors that can occur during the development and execution of your program: "syntax" errors, "semantic" errors, and "run-time" errors.

Syntax errors may include such errors as a missing operator, a misspelled keyword, or an illegal constant or variable name. When you press (END LINE) after typing in a statement, it is immediately checked for syntax errors. If the statement contains no syntax errors, it is accepted and loaded into computer memory. If the statement contains a syntax error, an error message is displayed and the cursor is positioned below the first character at which the system detected an error.

You can correct the statement by inserting, deleting, or replacing characters as shown in section 2. If the error is not corrected, the statement will not be stored as part of the program, but no other harm is done. Move the cursor down the display to enter another statement or clear the line in which the error occurred.

The second type of error, a "semantic" error, occurs when you have finished loading the program into computer memory and you try to run it. Before the HP-85 attempts to run your program, it checks to verify that your program "makes sense." Semantic errors include errors such as a missing statement, duplicate user-defined functions, illegal array dimensions, etc. You are informed of all such errors before the program can be run. These errors can usually be corrected by adding, deleting, or correcting statements; they are not difficult to find because the system alerts you to them as soon as you try to run the program.

The third type of error occurs when the program is running. All "run-time" errors interrupt a running program and cause it to halt unless DEFAULT ON is in effect or you override the errors with an ON ERROR statement. With DEFAULT ON, the first eight errors listed in appendix E cause a *warning* message to be output, but program execution will not be halted; all other run-time errors cause the program to halt and an error message to be displayed. With DEFAULT OFF, all run-time errors halt program execution and display an error message.

Run-time errors can include referencing a nonexistent array element, attempting to use uninitialized data, READ-DATA variable mismatch, trying to write to a nonexistent file, etc.

Refer to section 13, Debugging and Error Recovery, for information on recovering from run-time errors.

Refer to appendix E for a complete list of error numbers and messages.

Problems

6.1 For problem 5.2 in the previous section, you wrote a program that computed the distance traveled by Janey Dair's Rebounder ball. If each rebound reaches a height of 65% of its previous height, the time interval to the next bounce is $\sqrt{0.65} = 0.806 = 80.6\%$ of the previous time interval. Enter your original Rebounder program, and then modify it to incorporate a WAIT statement that causes a delay between bounces according to the ratio given above. Let the interval between the first two bounces be 3 seconds(3000 milliseconds).

6.2 To illustrate the effect of an unbalanced force pulling sidewards on a moving object, physics teacher Millie Graham has devised a simple experiment. She fastens a string to the side of a 350-gram miniature rocket sled and secures the other end to a fixed pivot. When ignited, the rocket sled accelerates at the rate of 30 centimeters per second per second. As the sled accelerates, the string continually pulls it sidewards, and this imbalance causes it to move in a circle of radius r (centimeters), equal to the length of the string. Ms.Graham knows that the magnitude of this force f, exerted inwards on the sled by the string, is given by

$$f = \frac{350(30\cdot t)^2}{r}$$

where t is the time (in seconds) from when the rocket is ignited. The force, expressed in dynes, can be converted to pounds by multiplying by 2.25×10^{-6}. Write a program for Ms.Graham that requests the length of the string r and then prints the force exerted by the string (in dynes and pounds) at intervals of 1 second. Have the program halt execution after each second's output so that she can determine if the string's breaking strength (2.22×10^6 dynes, 5.0 pounds) would be exceeded. Run the program for various string lengths, using trial and error to find the shortest length (approximately) for which the string lasts at least 10 seconds.

Branches and Loops

Normal program execution is in sequential order from the lowest numbered statement to the highest numbered statement. As we have seen with the GOTO statement, branching alters this process by transferring control to a statement that is not in the sequential flow.

Branches, loops, and subroutines are three methods of altering the normal flow of program execution. This section covers unconditional branching with the ON ... GOTO statement, conditional branching with IF ... THEN ... ELSE, and a method of forming efficient loops with the FOR and NEXT statements. In section 9 we will continue our discussion of branching with subroutines, the special function keys, and user defined functions.

Most of the programs we have discussed to this point have contained *unconditional branches* using the GOTO statement. The GOTO statement is simple and direct; it transfers program control to the statement number that you specify. A GOTO statement used in this way is known as an *unconditional branch* because it *always* branches execution from the GOTO statement to the specified statement number. Now you will see how to use IF ... THEN statements—branching that depends on the outcome of the test.

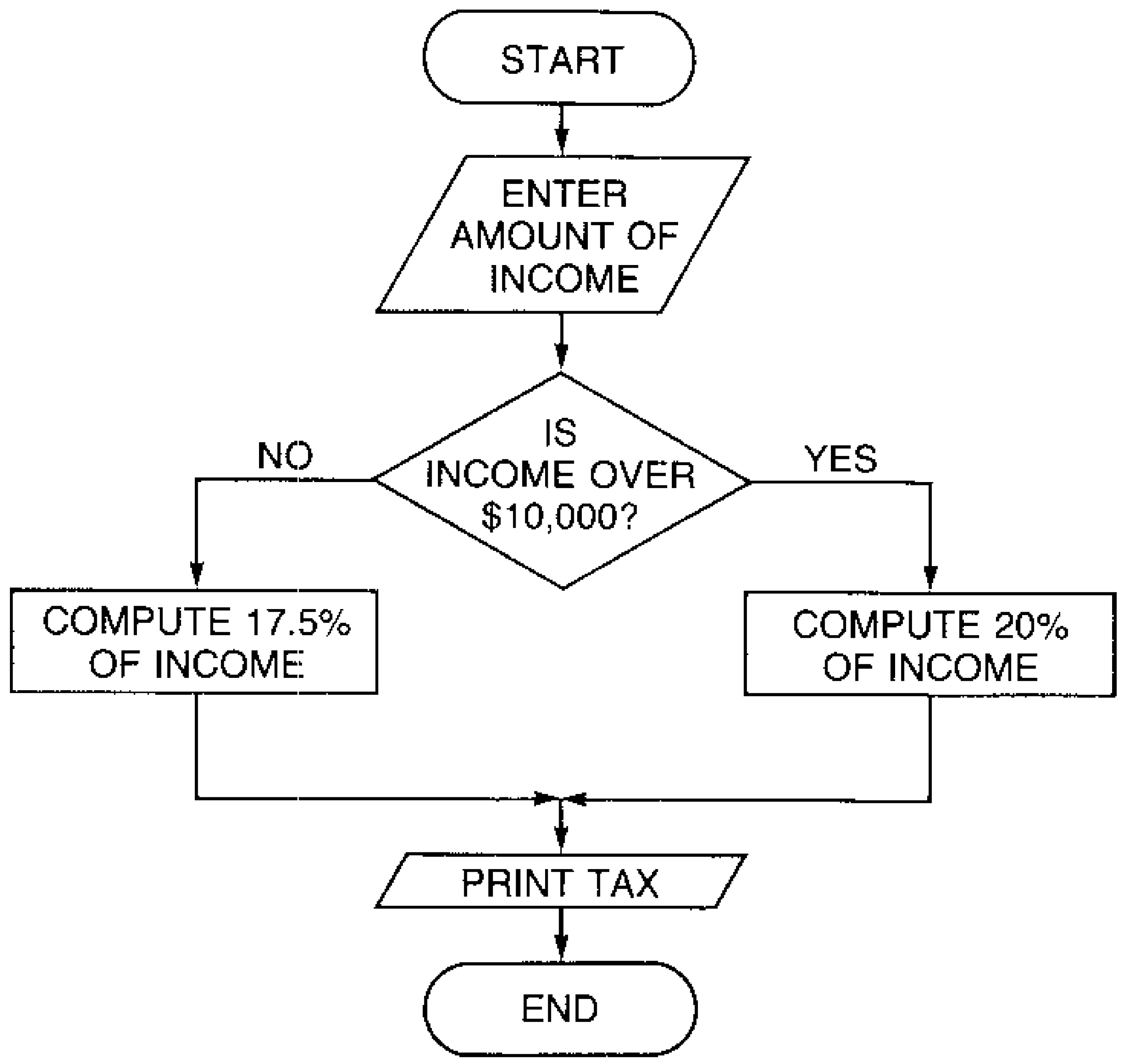
Conditional Branching

Often there are times when you want a program to make a decision. In the averaging program in section 1, we wanted the program to decide whether to branch to the end of the program to display the result, or whether to ask for more numbers to include in the average. As you may recall, the branch was dependent on the outcome of a specified condition, using the IF ... THEN statement. The HP-85 provides several forms of the IF ... THEN statement. One of them is:

IF numeric expression THEN statement number

The IF ... THEN statement makes a "decision" based upon the outcome of the numeric expression. If the expression is true, the THEN part of the statement is executed. If the outcome is false, execution continues with the statement following the IF ... THEN statement.

For example, suppose an accountant wishes to write a program that will calculate and print the amount of tax to be paid by a number of persons. For those with incomes of $10,000 per year or less, the amount of tax is 17.5%. For those with incomes of over $10,000, the tax is 20%. A flowchart for the program might look like this:



The diamond in the flowchart would be represented by an IF... THEN statement in a BASIC program. Thus a sample solution to the problem might be:

```
 10 DISP "INCOME";
 20 INPUT I
•30 IF I>10000 THEN 60
 40 PRINT "TAX=";I*.175
 50 GOTO 70
 60 PRINT "TAX=";I*.2
 70 END
```

"DO IF TRUE" rule:
IF TRUE THEN EXECUTE.
IF FALSE THEN GOTO NEXT STATEMENT IN PROGRAM.

As you can see, we used a relational operation in the IF... THEN statement. The IF... THEN statement is most often used with relational operators (=, <, >, <=, >=, <> or #), although the decision can be based on the value of any numeric expression as we shall see later.

If the condition is true, i.e., if the income is greater than $10,000, then program control is transferred to statement 60. If the condition is false—in this case, if the income is less than or equal to $10,000—then the rest of the IF statement is ignored and the program continues at statement 40.

Now, test your program with values of $20,000 and $9,000. We ran the sample program below in print all mode by executing the PRINT ALL command to print all inputs and outputs.

RUN

```
INCOME?
20000
TAX= 4000
```

Computed 20% of income.

RUN

```
INCOME?
9000
TAX= 1575
```

Computed 17.5% of income.

Remember from our discussion of the logical evaluation (page 53) that an operation is assigned the value of 1 if it is true and a value of 0 if it is false. Thus, in an IF... THEN statement, if the outcome of a numeric expression has a value other than 0, it is considered true; if it has a value of 0, it is considered false.

Example: Write a program to compute $^1/x$. Since division by zero yields an error, use an IF... THEN statement to check for a zero input. Then load the program and run it for values of 0 and 9.

Here is a sample solution to the problem:

```
 10 REM *RECIPROCAL*
 20 DISP "ENTER NUMBER"
 30 INPUT X
•40 IF X THEN 80
 50 DISP "THE RECIPROCAL OF ZERO"
 60 DISP "IS UNDEFINED!!!"
 70 GOTO 20
 80 PRINT "1 /";X;"=";1/X
 90 END
```

If X is any number other than 0, then the program branches to statement 80. The statement means the same as IF X#0 THEN 80. If X is 0, then execution continues with statement 50.

Before we ran this program, we executed the PRINT ALL command to print all inputs and outputs.

RUN

```
ENTER NUMBER
?
0
THE RECIPROCAL OF ZERO
IS UNDEFINED!!!
ENTER NUMBER
?
9
1 / 9 = .111111111111
```

Another form of the IF... THEN statement provides conditional execution of a statement without necessarily branching:

IF numeric expression THEN executable statement

Again, when the condition is true (or the value of the numeric expression is other than zero), the statement is executed. When the condition is false (the value of the numeric expression is zero), execution continues with the following statement.

All *executable* BASIC statements are allowed to follow THEN except the FOR, NEXT, and IF statements.

The following statements are not allowed after THEN because they are *declaratory,* not executable statements:

```
COM        IMAGE
DATA       INTEGER
DEF FN     OPTION BASE
DIM        REAL
FN END     SHORT
```

Example: Write a program to make Celsius/Fahrenheit temperature conversions such that:

1. If you enter a C, the temperature is converted from Celsius degrees to Fahrenheit according to the formula $F = 32 + 9/5 * C$.
2. If you enter an F, the temperature is converted from Fahrenheit to Celsius according to the formula $C = (F - 32) * 5/9$.
3. If neither C nor F is entered, nothing is printed.

If you wrote programs for problems 5.1.a. and 5.1.b., combine them. Use the second form of the IF... THEN statement in your program to determine which conversion is to be made.

Here is a listing of a sample solution:

```
10 ! *TEMPERATURE CONVERSIONS*
20 DISP "ENTER TEMPERATURE,F OR
   "
30 DISP "ENTER TEMPERATURE,C"
40 INPUT T,D$
•50 IF D$="F" THEN PRINT (T-32)*
    5/9;"C IS";T;"F"
•60 IF D$="C" THEN PRINT T;"C IS
    ";32+9/5*T;"F"
70 GOTO 20
80 END
```

Run the program to convert 0°C and 100°C to degrees Fahrenheit; 50°F and 98.6°F to degrees Celsius.

Here are the results printed from our program:

```
0 C IS 32 F
100 C IS 212 F
10 C IS 50 F
37 C IS 98.6 F
```

The ELSE Option

There's still more to the IF... THEN statement: ELSE. In the previous examples, if the numeric expression was evaluated as false, program execution continued with the next sequential statement following the IF... THEN statement. But if you specify the ELSE option with the IF... THEN statement, the program will instead perform the indicated ELSE instructions. This gives you tremendous power with conditional branching; six different forms of the IF... THEN statement are available.

IF numeric expression THEN statement number [ELSE statement number
or or
executable statement executable statement]

If the numeric expression is false and ELSE is specified, execution is transferred to the statement number following ELSE or the indicated ELSE statement is executed.

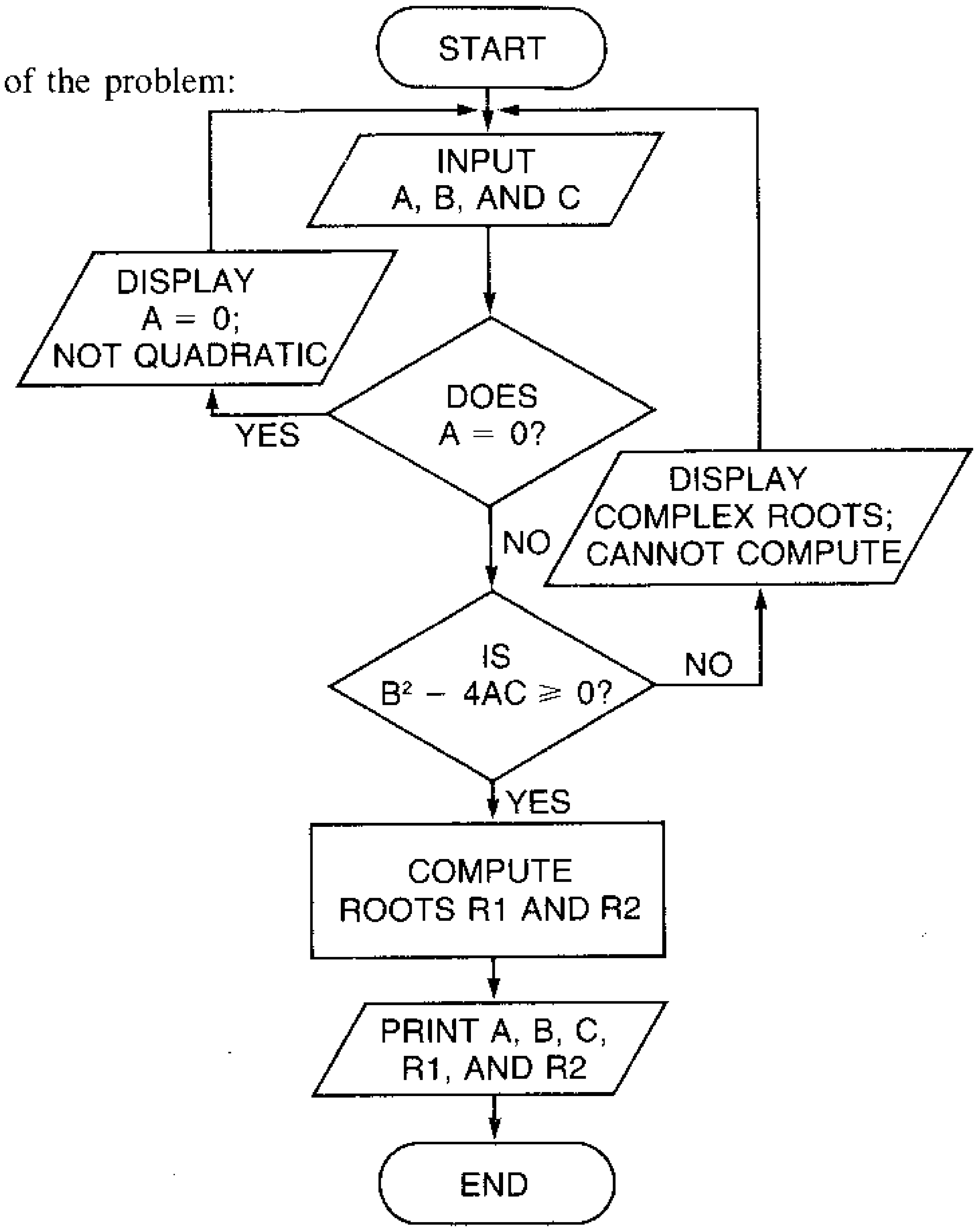
Let's look at an example.

Example: A quadratic equation is of the form $0 = ax^2 + bx + c$. If $a \neq 0$, its two roots may be found by the formulas

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Write a program to compute the roots of a quadratic equation given the values of the coefficients a, b, and c. If a is zero, display an error message and reenter new values. If $b^2 - 4ac$ is less than zero, then the square root of that value would give a warning message or an error. So make sure that $b^2 - 4ac$ is greater than or equal to zero before you compute the roots.

Here's a flowchart of the problem:



In this sample solution we use two forms of the IF... THEN... ELSE statement. Study it carefully, then load the program and run it.

```
 10 REM *ROOTS*
 20 DISP "IF A QUADRATIC"
 30 DISP "EQUATION IS OF THE"
 40 DISP "FORM 0=A*X^2+B*X+C"
 50 DISP "ENTER A,B,C"
 60 INPUT A,B,C
 70 D=B*B-4*A*C
• 80 IF A=0 THEN DISP "A=0;NOT QU
    ADRATIC.REENTER VALUES" ELSE
     100
 90 GOTO 60
•100 IF D>=0 THEN 120 ELSE DISP "
    COMPLEX ROOTS:  CANNOT COMPU
    TE.  REENTER VALUES."
110 GOTO 60
120 R1=(-B+SQR(D))/(2*A)
130 R2=(-B-SQR(D))/(2*A)
140 PRINT "COEFFICIENTS=";A;B;C
150 PRINT "ROOTS=";R1;R2
160 END
```

If A=0, displays message, then continues to next statement. If A≠0, reads ELSE 100 and program branches to statement 100.
If D>=0, branches to statement 120.
If D<0, displays ELSE message then continues to statement 110.

The instruction following ELSE in an IF ... THEN statement may be a statement number or an executable statement. Again, the same stipulations hold for ELSE as THEN; you may use any executable statement except FOR, NEXT, and IF and you may not use declaratory statements.

Run the program to find the roots of the equation $x^2 + x - 6 = 0$. Then run the program again to test the decisions with $x + 1 = 0$ and $x^2 + 2x + 2 = 0$; finally the roots of $3x^2 + 2x - 1 = 0$.

```
PRINTALL
RUN

IF A QUADRATIC
EQUATION IS OF THE
FORM 0=A*X^2+B*X+C
ENTER A,B,C
?
1,1,-6                                  Coefficients of x²+x-6=0.
COEFFICIENTS= 1  1 -6
ROOTS= 2 -3                             Result.

RUN
IF A QUADRATIC
EQUATION IS OF THE
FORM 0=A*X^2+B*X+C
ENTER A,B,C
?
0,1,1                                   Coefficients of x+1=0.
A=0;NOT QUADRATIC.REENTER VALUES        Displays message.
?                                       Asks for new input.
1,2,2                                   Coefficients of x²+2x+2=0.
COMPLEX ROOTS:  CANNOT COMPUTE.         Displays message.
REENTER VALUES.
?                                       Asks for new values.
3,2,-1                                  Coefficients of 3x²+2x-1=0.
COEFFICIENTS= 3 2 -1
ROOTS= .333333333333 -1                 Result.
```

For a more efficient and accurate method of finding the roots of a quadratic equation, refer to the Polynomial Evaluation program in your HP-85 Standard Pac.

The Computed GOTO Statement

There is one more form of *unconditional* branching that you should be aware of: the ON... GOTO or computed GOTO statement.

ON numeric expression GOTO statement number list

The ON... GOTO statement enables you to transfer program control to one of one or more statements, depending on the value of a numeric expression.

The numeric expression is evaluated and rounded to an integer. A value of 1 causes control to be transferred to the first statement specified in the statement list; a value of 2 causes control to be transferred to the second statement specified in the list, and so on. A value less than 1 causes an error. A value greater than the number of statements in the list also causes an error.

Essentially, the ON... GOTO statement is a combination of the IF statement and the GOTO statement.

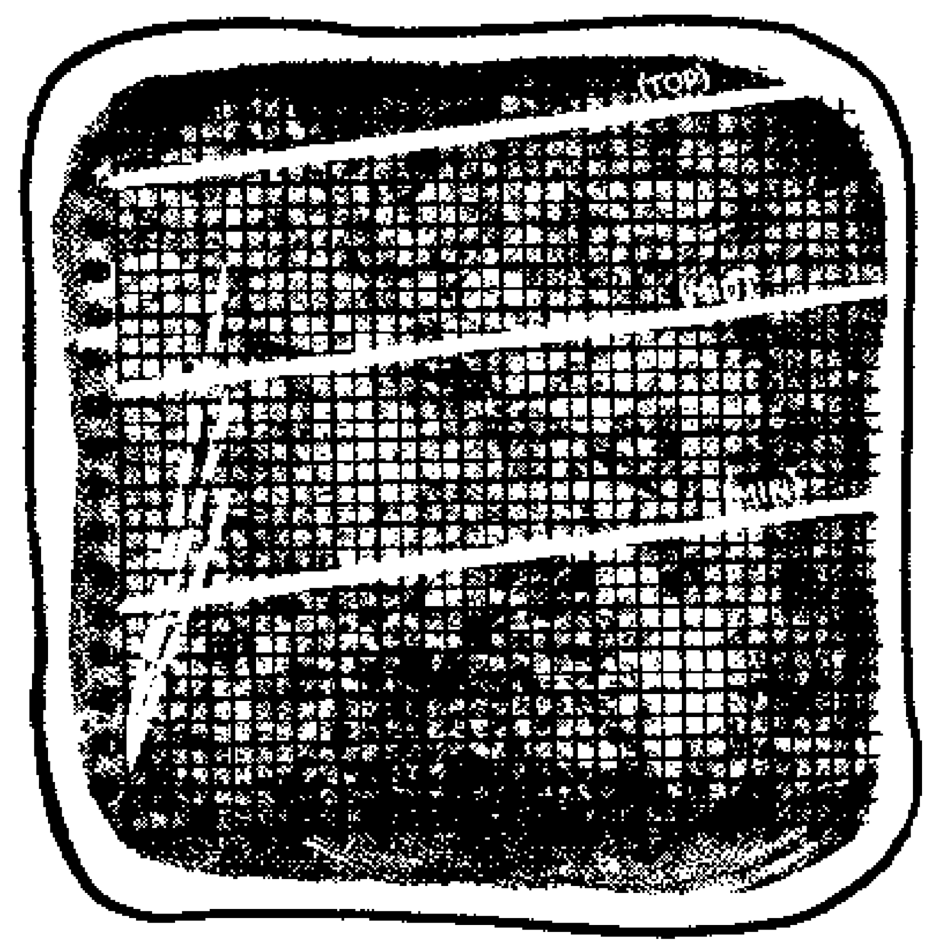
For example:

```
20 ON R GOTO 25, 80, 150
```

This statement says: if R=1, go to statement 25, if R=2, go to statement 80, and if R=3, go to 150. But if R<1 or if R>3, an error would occur.

Look at the following application of an ON... GOTO statement:

Example: The payroll clerk of a small firm wishes to write a program to compute the weekly wages of the employees according to the following payscales:



Payscale	Hourly Wage
1	$4.75
2	$5.50
3	$6.25

Also, overtime must be taken into account. If the employee works more than 40 hours in the week, the remaining hours should be multiplied by 1.5.

Sample Solution:

```
 10 PRINT "NAME"
 20 PRINT "HOURS","WAGES"
 30 PRINT
 40 E=0
 50 DISP "LAST NAME,FIRST INIT."
 60 INPUT N$
 70 DISP "HOURS WORKED";
 80 INPUT H
 90 IF H>40 THEN E=H-40
100 H1=H-E+E*1.5
110 DISP "PAY SCALE 1,2,OR 3";
120 INPUT P
•130 ON P GOTO 140,160,180
140 W=4.75*H1
150 GOTO 190
160 W=5.5*H1
170 GOTO 190
180 W=6.25*H1
190 PRINT N$
200 PRINT H,W
210 PAUSE
220 GOTO 40
230 END
```

Since we have not dimensioned N$, the name can be no longer than 18 characters. (We'll discuss this later.)

Computes overtime.

Computed GOTO branches to 140 if P=1, 160 if P=2, and 180 if P=3.

Computes wages according to desired pay scale.

PAUSE after wages are printed for each employee. When the program is running, press (CONT) to continue.

Run the program for the following list of employees. Remember that if a comma is part of your string input, the string expression must be enclosed within quotes (e.g., enter "JONES, J." for the first name).

Name	Hours	Payscale
Jones, J.	43	2
Smith, K.	52	3
Fender, L.	40	1
Morris, D.	44	2

Your printout should look like this:

```
NAME
HOURS              WAGES

JONES,J.
  43               244.75
SMITH,K.
  52               362.5
FENDER,L.
  40               190
MORRIS,D.
  44               253
```

Note: If the value of the numeric expression is less than one or greater than the number of statement numbers in the list, Error 11 (argument out of range) occurs.

In the following example when statement 20 is executed for the third time, the value of I exceeds the number of statement numbers in the list.

```
 10 I=1
•20 ON I GOTO 30,30,60
 30 DISP "I=";I
 40 I=I*2
 50 GOTO 20
 60 DISP "I IS GREATER THAN 3"
 70 END
```

Running the program:

(RUN)

```
I=1
I=2
Error 11 on line 20 : ARG OUT OF
 RANGE
```

FOR-NEXT Loops

Repeatedly executing a series of statements is known as looping. We have seen several loops in programs; the future value program contained a loop—as did the checkbook balancing program.

A clear and efficient way to create loops is to use the FOR and NEXT statements. The FOR and NEXT statements are used to enclose a series of statements, enabling you to repeat those statements a specified number of times.

FOR loop counter = initial value TO final value [STEP increment value]

⋮

NEXT loop counter

The FOR statement defines the beginning of the loop and specifies the number of times the loop is to be executed. The loop counter must be a simple numeric variable.

The initial, final, and increment values can be any numeric expression. If the increment value is not specified, the default value is one.

FOR-NEXT
loop range

```
 50 FOR I=1 TO 5
 60 PRINT I
 70 NEXT I
 80 PRINT "FINISHED WITH LOOP;"
 90 PRINT "I NOW EQUALS";I
100 STOP

RUN
 1
 2
 3
 4
 5
FINISHED WITH LOOP;
I NOW EQUALS 6
```

The FOR-NEXT loop will be executed five times: when I=1,2,3,4, and 5. Each time the NEXT statement is executed, the value of I is incremented by 1. But when the value of I passes the final value, that is, when I=6, the loop is finished, and execution continues with the statement following the NEXT statement (in this case, 80).

The FOR statement does three things for your program:

1. It sets the loop counter to the initial value.
2. It tells the computer what the final value for the index may be (and thereby when to stop looping).
3. It tests to see if the counter has gone beyond the final value. If so, the program exits the loop; if not, the program continues with the statement following the FOR.

The NEXT statement does two things for your program:

1. It increments the loop counter.
2. It defines the end of the loop. After the program has completed the loop the number of times specified, execution continues with the statement following NEXT (if, of course, you have not branched elsewhere within the loop).

Example: Use a FOR-NEXT loop to compute and print the area of a circle with an integer radius from 15 centimeters to 20 centimeters, according to the formula $A = \pi r^2$.

```
AUTO
•10 FOR R=15 TO 20
 20 A=PI*R*R
 30 PRINT "RADIUS=";R;"AREA=";A
•40 NEXT R
 50 STOP
```

Notice that the initial value does not have to be 1.

Again, we set the initial value and the final value with the FOR statement. When R exceeds 20, program execution is transferred to the statement following the NEXT statement.

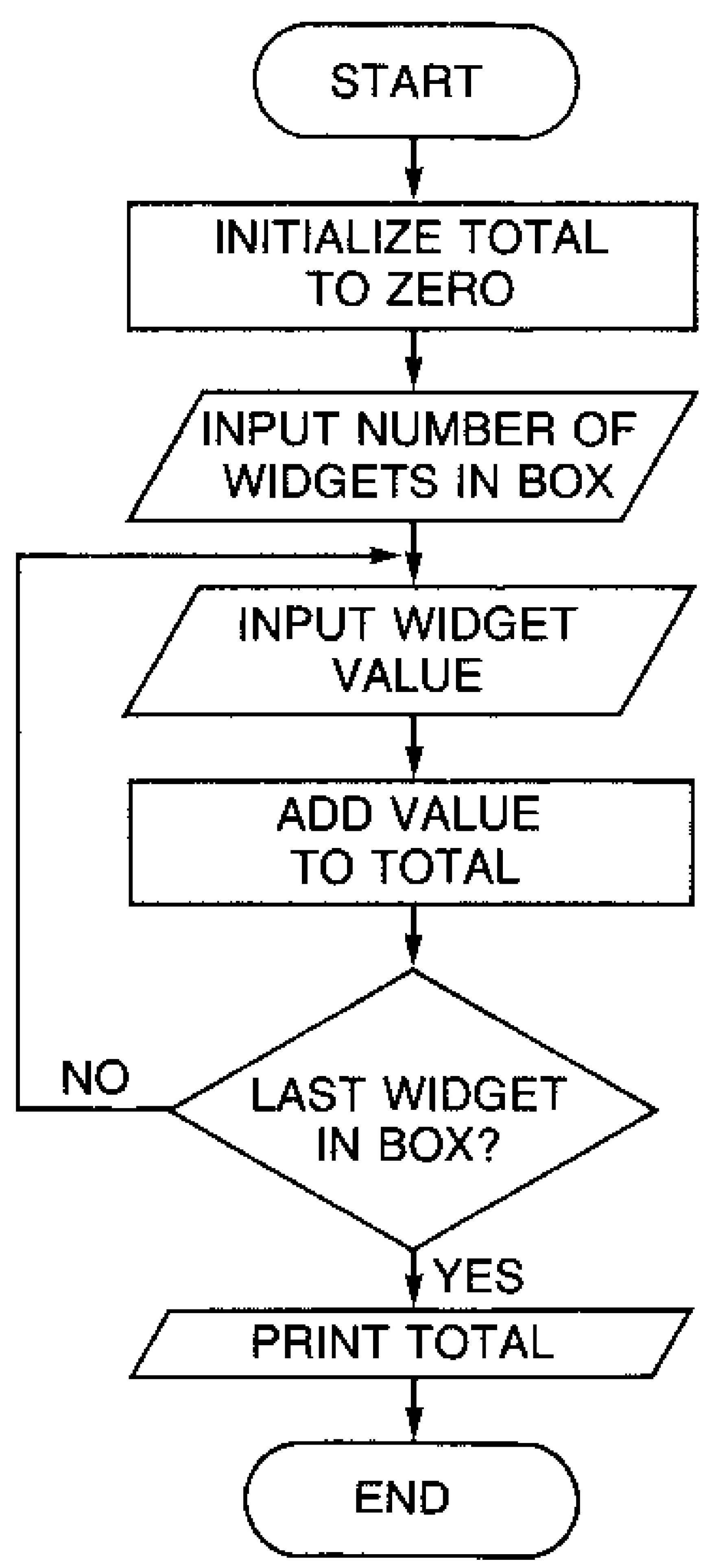
```
RUN
RADIUS= 15 AREA= 706.858347059
RADIUS= 16 AREA= 804.247719318
RADIUS= 17 AREA= 907.920276887
RADIUS= 18 AREA= 1017.87601976
RADIUS= 19 AREA= 1134.11494795
RADIUS= 20 AREA= 1256.63706144
```

The loop is executed 6 times from R=15 through R=20.

You can also use variables or numeric expressions to specify the initial or final values.

Example: Suppose you are a widget maker. The shipping department in the widget factory can pack widgets in a variety of ways—rarely do two boxes contain the same number of widgets. Since widgets come in various shapes and sizes, the value of each widget varies. But you want to insure the box for the true value of the widgets inside. Write a program to accept the number of widgets in a particular box and then accept the value of each widget in the box, compute the total, and print the value to be insured.

Your flowchart might look like this:



Sample Program:

```
  10 REM *WIDGETS*
  20 T=0
  30 DISP "ENTER NUMBER OF WIDGET
     S"
  40 INPUT N
• 50 FOR I=1 TO N
  60 DISP "WIDGET VALUE";
  70 INPUT W
  80 T=T+W
• 90 NEXT I
 100 DISP "TOTAL VALUE OF BOX=";T
 110 END
```

Here, you actually input the final value of the loop.
FOR-NEXT loop range.

Now run the program for a box of five widgets, with individual values of $3.50, $4.95, $2.60, $18.50, and $5.10:

```
RUN
ENTER NUMBER OF WIDGETS
?
5
WIDGET VALUE?
3.50
WIDGET VALUE?
4.95
WIDGET VALUE?
2.60
WIDGET VALUE?
18.50
WIDGET VALUE?
5.10
TOTAL VALUE OF BOX= 34.65
```

As we mentioned earlier, it is possible to use expressions in the FOR statement as either the initial value or the terminating value. For example, you could have a problem that requires you to have statements like the following:

```
130 FOR I=N/2 TO N*2-1
```
or
```
266 FOR J=1 TO N*8
```
or
```
470 FOR K=2*J TO 1000
```

In these instances, when the FOR statement is executed, the first expression is evaluated and the *numeric value* is stored as the initial value; then the second expression is evaluated and that *value* is stored as the terminating value. After the initial and terminating values have been stored, the variable N can be changed without altering either of them. But the loop counter variable should not be changed within a loop by an assignment statement.

For example, this program has problems:

```
100 FOR I=1 TO 10
110 I=3
120 NEXT I
130 END
```

This program would create an infinite loop, like those we've seen before, except worse since nothing is displayed or printed. The variable I is reset below the final value each time the program executes the loop.

Changing the Increment Value

In all of the FOR-NEXT loops above, the computer increments the counter by 1 each time through the loop. But you are not limited to just 1. You can use any stepping value, positive, negative, or non-integer, with the STEP parameter.

For example, suppose you wish to print the odd integers from 1 to 10. You could use a FOR-NEXT-STEP loop like this:

```
10 FOR I=1 TO 10 STEP 2
20 PRINT I;
30 NEXT I
40 PRINT
50 END

RUN

 1 3 5 7 9
```

The initial value of I is 1. Each time NEXT I is executed, I is incremented by 2. In this program, I = 1, 3, 5, 7, and 9. When I reaches 11, the loop is exited and the program ends. Since the PRINT statement in line 20 ends with a semicolon, an extra PRINT statement completes the print message and outputs it to the printer.

You can also decrement the loop counter.

Example: Write a program that requests a number and computes its factorial. A factorial is an integer multiplied by all of the other integers below it (down to 1). For instance, $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$. (Also consider limiting the size of the number a user may give. What happens if a negative number or a noninteger is entered?)

Enter the following program into the system:

```
 10 REM *FACTORIAL*
 20 DISP "FACTORIAL OF";
 30 INPUT N
 40 IF FP(N)=0 AND N>=0 THEN 70
 50 DISP "POSITIVE INTEGERS ONLY"
 60 GOTO 20
 70 F=1
 80 FOR P=N TO 1 STEP -1
 90 LET F=F*P
100 NEXT P
110 DISP "FACTORIAL=";F
120 END
```

Check to make sure the number is a positive integer.

Loop counter is decremented from N to N−1, and so on.

Now run the program to find the factorials of 4 and 24.

```
RUN
FACTORIAL OF?
4
FACTORIAL= 24
```

Result.

```
RUN
FACTORIAL OF?
24
FACTORIAL= 6.20448401736E23
```

Result. Remember, the computer overflows with numbers larger than $9.99999999999 \times 10^{499}$.

Let's see how the factorial of 4 was computed. After you input 4, the initial value of the FOR statement was set to 4. So the program read the statement as:

```
FOR P=4 TO 1 STEP -1
```

The values for F were computed as follows:

F=1*4	First time through loop.
F=4*3	Second time through loop; P=4−1.
F=12*2	Third time through loop; P=3−1.
F=24*1	Last time through loop; P=2−1.

When the NEXT statement decrements the P value to 0, the loop is exited.

Nested Loops

When one loop is contained entirely within another, the inner loop is said to be nested. A loop can be contained within a loop that is contained within a loop ... (up to 255 nested loops), as long as the loops do not overlap each other.

A FOR-NEXT loop cannot overlap another FOR-NEXT loop, for instance:

Incorrect Nesting

```
10 PRINT "I","J"
20 FOR I=1 TO 3
30 FOR J=4 TO 6
40 PRINT I,J
50 NEXT I
60 NEXT J
70 END
```

Correct Nesting

```
10 PRINT "I","J"
20 FOR I=1 TO 3
30 FOR J=4 TO 6
40 PRINT I,J
50 NEXT J
60 NEXT I
70 END
```

In the incorrect nesting example, the I loop is activated and then the J loop is activated. But the J loop is cancelled when NEXT I is executed because it's an inner loop. When the I loop is completed and NEXT J is accessed, Error 47 on line 60 is displayed. This is because the J loop was cancelled and was not reactivated after the last I loop.

Run the correct nesting example now to view the looping process:

```
RUN
  I    J
  1    4
  1    5
  1    6
  2    4
  2    5
  2    6
  3    4
  3    5
  3    6
```

The J loop is completed before I is incremented.

Now I=2 and the program runs through the J loop again.

Finally I=3; the J loop is executed once again. When I reaches a value of 4, the program halts.

FOR-NEXT Loop Considerations

- Execution of a FOR-NEXT loop should always begin with the FOR statement. Branching into the middle of a loop (with statements like GOTO or IF) will produce Error 47 if the NEXT statement is executed before the program executed the corresponding FOR statement.
- Execution of a loop normally ends with a NEXT statement. It is permissible to transfer program control out of the loop by a statement within the loop. After an exit is made through a branch within the loop, the current value of the counter is retained and is available for later use in the program. In this case, it is permissible to reenter the loop either at a statement within the loop, or at the FOR statement (thereby reinitializing the counter).
- A FOR-NEXT loop will *not* be executed if the initial value is greater than the final value when a positive STEP value is used, or if the initial value is less than the final value when a negative STEP value is used.
- An often overlooked aspect of FOR-NEXT looping is that the actual value of the counter when the loop is complete does not equal the final value. The NEXT statement increments or decrements the loop counter *past* the final value before the loop is exited. (We'll see an example of this in the graphics section, Padding the FOR-NEXT loop.)

Problems

7.1 As an avid sports fan, you decide to write a program that will help you keep score during an important basketball game between the Aakerville Aardvarks and the Wiggenberg Wombats. You can enter an *A* or *W* to signify a field goal (worth 2 points) for the appropriate team, and an *a* or *w* to signify a free throw (worth 1 point). The score should be printed after each entry.

7.2 The common game of "Buzz" offers a challenge to a person's number skills. This version, called "Beep," requires you to program the HP-85 to successfully complete the same game. The game consists of counting (displaying) numbers from 1 to 100. However, for any number that is evenly divisible by 7 or contains a 7, the display should leave a blank and the HP-85 should "beep." If the number both contains a 7 and is evenly divisible by 7, two "beeps" should be sounded.

Hint: The "ones" digit of a two-digit integer can be found as 10*FP(X/10) or XMOD10.

7.3 Here is a check to see whether you and the HP-85 can communicate using "mental" telepathy. Write a program that uses the RND random number generator to "pick" a number from 1 to 5, waits for 5 seconds while "concentrating" on the number, and then requests from you the number that comes to your mind. The display should indicate whether your entry is correct or incorrect. After every 10 picks, the printer should list a summary of your accuracy and indicate whether it is better or worse than that expected by chance (20% accuracy). The random "picks" can be generated by P=IP(1+5*RND).

7.4 Boy Scout Jeffrey Goodfellow is preparing for his compass-course test, in which he must follow several legs of a course and attempt to be within an allowable error of the finish point. Each leg of the course is defined by a magnetic bearing (θ) to be followed and the distance (d) to be traveled. Jeffrey realizes that each leg can be converted to northerly and easterly distances (d_n and d_e) according to:

$$d_n = d \times \cos(\theta)$$

$$d_e = d \times \sin(\theta)$$

If the northerly and easterly distances are summed for all the legs in the course, these two sums can be used to determine the direct bearing (θ_f) and distance (d_f) of the finish point relative to the starting point:

$$\theta_f = \arctan\,(d_e/d_n)$$

$$d_f = \sqrt{d_e^2 + d_n^2}$$

If he had a program to perform these calculations, Jeffrey could check his accuracy during his practice sessions. Write a program that requests the bearing and distance for each leg of the compass course. (A distance of zero should indicate that all of the legs have been entered.) The program should produce a listing of the bearings and distances, and then give the direct bearing and distance of the finish point relative to the starting point. Use the `ATN2(Y,X)` function to compute θ_f so that the proper angle is chosen. (If θ_f is negative, add 360° or use θ_fMOD360 to obtain the bearing in the correct form.)

Hint: Don't forget the `DEG` statement.

7.5 Your medical supplies business has offices in Britain, France, and the United States. With such an arrangement, you must frequently convert monetary values among the three currency systems: British pound, French franc, and U.S. dollar. In order to facilitate these conversions, you decide to write a program to compute them for you. Each currency system is to be denoted by a code number. The program is to be initialized each day by entering equivalent monetary values in each currency. Each required conversion should begin by entering the currency code and amount to be converted; the currency system and equivalent amount is to be printed for each system. On a certain morning, 1 British pound is equivalent to 8.3981 French francs and 1.8248 U.S. dollars. At these rates, find the equivalent values of a patient lift worth 284 British pounds and a hospital bed valued at 1205 U.S. dollars.

7.6 The mayor of Dimsburg has directed Elmo Rumple, the town statistician, to study the problem of motorists having to stop at all three of Dimsburg's traffic lights. Elmo confines his analysis to those motorists who are delayed at all three lights. He assumes that each car arrives randomly at each red light, indicating that the delay at each light is uniformly distributed between 0 and 1 minute (the duration of a red signal). The total delay in Dimsburg is therefore the sum of the three uniformly-distributed delays. Elmo wants to compute the probability that this delay is shorter than various time intervals. From his vast experience, he knows that this probability is given by the following function (called a distribution function).

$$\text{Prob(delay} < \text{T)} = \begin{cases} 0 & \text{for T} < 0 \\ \text{T}\wedge 3/6 & \text{for } 0 \leq \text{T} < 1 \\ .5-\text{T}*(1.5-\text{T}*(1.5-\text{T}/3)) & \text{for } 1 \leq \text{T} < 2 \\ -3.5+\text{T}*(4.5-\text{T}*(1.5-\text{T}/6)) & \text{for } 2 \leq \text{T} < 3 \\ 1 & \text{for T} \geq 3 \end{cases}$$

Help Elmo by writing a program to compute the probability of a total delay that is less than any specified time. (Use an `ON... GOTO` statement to branch to the proper equation.)

Section 8

Using Variables: Arrays and Strings

As we mentioned earlier, there are three types of numeric variables available with the HP-85: REAL (full precision), SHORT, and INTEGER numbers. A fourth type of variable deals with character strings. Numeric variables can have two forms: simple (non-subscripted) and array (subscripted). Strings may also be subscripted, but not in the form of an array.

In this section, we discuss array and string variables, their functions, and how to use them.

Array Concepts

An array variable (or simply, an array) is a collection of data items of the same type under one name. An array may have one or two dimensions. For instance, a one-dimensional array (often called a vector) might be thought of as a *list* of items; there may be several rows but only one column. A two-dimensional array (often called a matrix) is like a *table* of values; there may be several rows and several columns of items.

Suppose we have the following list of numbers:

$$\left.\begin{array}{r} 1 \\ 4 \\ 9 \\ 16 \\ 25 \end{array}\right\}$$

We could store this list of numbers, in the order shown, in a one-dimensional array.

If we name this set of numbers array S, we can specify the individual elements of S by using subscripts.

If numbering of array subscripts begins with 0, the elements of array S are specified as:

S(0)=1
S(1)=4
S(2)=9
S(3)=16
S(4)=25

Subscripts	Array Elements
0	1
1	4
2	9
3	16
4	25

If the numbering of array S begins with 1, then the elements of array S are:

S(1)=1
S(2)=4
S(3)=9
S(4)=16
S(5)=25

Subscripts	Array Elements
1	1
2	4
3	9
4	16
5	25

We need to use a two-dimensional array to store the values in the following table:

Number	Square	Square root	Factorial
1	1	1	1
2	4	1.41421356237	2
3	9	1.73205080757	6

This table contains 3 rows and 4 columns for a total of 12 values.

If subscript numbering begins at 0, the elements are identified as follows:

D(0,0) =1	D(0,1) =1	D(0,2) =1	D(0,3) =1
D(1,0) =2	D(1,1) =4	D(1,2) =1.41421356237	D(1,3) =2
D(2,0) =3	D(2,1) =9	D(2,2) =1.73205080757	D(2,3) =6

Array D

Subscript	0	1	2	3
0	1	1	1	1
1	2	4	1.41421356237	2
2	3	9	1.73205080757	6

Each element in array D is specified by its location in the array with two subscripts, separated by a comma, and enclosed within parentheses. The first subscript designates the "row" in the array; the second subscript designates the "column."

If numbering of the subscripts begins with 1, array D would be represented:

Subscript	1	2	3	4
1	1	1	1	1
2	2	4	1.41421356237	2
3	3	9	1.73205080757	6

Thus, 9 would be represented as D(3,2); 6 would be represented as D(3,4).

Array names are the same as simple variable names; an array name may be a letter from A through Z, or a letter immediately followed by a digit from 0 through 9. But whenever an array is specified, it must be followed by subscripts enclosed within parentheses, otherwise it specifies a simple variable. The range of each subscript is an integer from 0 through 32767, but the maximum array size is determined by available memory. A non-integer subscript is rounded to the nearest integer if it is within the dimensioned range of subscripts.

Arrays are extremely convenient for handling large groups of data within a program because a group of different values are known under the same name. The different values (or *elements* of the array) are distinguished in name by subscripts to the array name.

An array name followed by a single subscript enclosed within parentheses specifies a one-dimensional array or an element of that array. An array name followed by two subscripts separated by a comma, both enclosed within parentheses, specifies a two-dimensional array or an element of that array. (No more than two subscripts are allowed.) Whether the array name is understood as the whole array or as a specific element depends on the type of statement that is used. Declaration statements refer to the whole array, executable statements usually refer to an array element.

Declaring and Dimensioning Variables

Five variable declarative statements are available to dimension arrays and strings and declare the precision of numeric variables:

```
COM
DIM
INTEGER
SHORT
REAL
```

They can be anywhere in a program (except that in multistatement lines, they must be the last statement in the line). The subscripts of a variable must specify the physical or maximum size of the variable.

Lower Bounds of Arrays

Earlier we saw that subscript numbering can begin with 0 or 1. The HP-85 assumes that all array subscripts begin at 0 unless you specify otherwise with an `OPTION BASE` statement.

When dimensioning arrays, you may want to specify that the lowest numbered subscript be 1 rather than 0.

```
OPTION BASE 1
```

This statement must come before any array variables are referenced in a program. `OPTION BASE 1` tells the computer to begin numbering all subscripts of arrays with 1.

The real advantage to using `OPTION BASE 1` is that you can refer to an array element directly by its position in the array without wasting element 0. Thus, the first element in a one-dimensional array S is S(1) rather than S(0); the second element is S(2) rather S(1) and so on. And if array S contained 10 elements, it would be declared as S(10) rather than S(9).

If `OPTION BASE 1` is not declared in a program, you may wish to include the statement `OPTION BASE 0` for documentation purposes. But this is not necessary since `OPTION BASE 0` is the *default* array counting system at power on. There may only be *one* `OPTION BASE` statement in a program.

The `OPTION BASE` statement cannot be executed from the keyboard.

The DIM Statement

The `DIM` *(dimension)* statement is used to dimension (allocate memory) and reserve memory for full precision numeric arrays. It is also used to dimension and reserve storage space for strings.

`DIM` item [, item...]

The item can be:

- A numeric array, with subscripts enclosed within parentheses.
- A string, with the number of characters enclosed within brackets.

The `DIM` statement specifies the upper bound of an array and the maximum number of characters that a character string may have.

Remember that the HP-85 assumes that the lower bound of an array is 0 unless you specify it to be 1 with `OPTION BASE`.

Examples:

`10 DIM A(100)`	Declares a one-dimensional array A of 101 elements; A(0),..,A(100).
`20 DIM B(3,2),C$[56]`	Declares a two-dimensional array B of 12 elements (4 by 3) and a character string C$ of 56 characters maximum. (Refer to our discussion of strings on pages 51 and 52.)

With `OPTION BASE 0` the number of elements in each dimension of a numeric array is calculated by adding one to each upper bound subscript. Then the resulting values are multiplied together to yield the total number of elements in a two-dimensional array.

Examples:

`5 OPTION BASE 1`	Declares the lower bound of all arrays to be 1.
`10 DIM A(100),B(3,2),C$[56]`	Dimensions an array A with 100 elements array B with 6 elements (3 by 2), and a string C$ with 56 characters.

The memory allocated to a character string is not affected by `OPTION BASE`. In a `DIM` statement, the number within brackets always refers to the number of characters allocated to the string. The maximum number of characters that may be specified for a string is 32767, but this is limited by available memory.

Type Declaration Statements

All numeric variables (simple and array) are assumed to be full precision variables (type `REAL`), unless they appear in a type declaration statement. A type declaration statement specifies the type of variable, `REAL`, `SHORT`, or `INTEGER`.

`INTEGER` numeric variable$_1$ [`(`subscripts`)`] [`,` numeric variable$_2$ [(subscripts)]...]
`SHORT` numeric variable$_1$ [`(`subscripts`)`] [`,` numeric variable$_2$ [(subscripts)]...]
`REAL` numeric variable$_1$ [`(`subscripts`)`] [`,` numeric variable$_2$ [(subscripts)]...]

The `INTEGER` statement dimensions and reserves memory for integer precision variables—simple and array.

The `SHORT` statement dimensions and reserves memory for short precision variables—simple and array.

And the `REAL` statement dimensions and reserves memory for full precision variables—simple and array.

Since the `DIM` statement is used to dimension full-precision variables, and undeclared simple variables are assumed to be full-precision, the `REAL` statement is only useful for documentation purposes.

Examples:

```
10 INTEGER A,B,C(10)
```
Declares variables A and B to be integers; declares and dimensions array C to 11 integer elements, assuming OPTION BASE 0.

```
20 SHORT P(20,25),P1,P2
```
Declares and dimensions short-precision elements for array P; declares variables P1 and P2 to be short-precision.

```
30 REAL X5,D(4,4)
```
Declares array D and variable X5 to be type REAL.

The COM Statement

The `COM` *(common)* statement is used to dimension and reserve variables to be held in common in two or more programs. `COM` is primarily used with the `CHAIN` statement (section 11) to pass variables between programs. A `COM` statement may also be used to deallocate a program before it is stored (refer to page 263).

The variables in common must agree in type and size between programs that are `CHAIN`ed.

`COM` item [, item,...]

The item can be:

- A simple numeric variable.
- A subscripted array.
- A string with number of characters enclosed within brackets.

In addition, any one of the type words— `INTEGER`, `SHORT`, or `REAL`—may precede one or more variables.

Example:

```
25 COM A,B(4,3),C$[5],D,INTEGER
   E,F$[24],G,SHORT H(5), J
```

The variables A,B(4,3), and D are full precision. Full precision is assumed at the beginning of the `COM` list and for numeric variables declared after a type `REAL` declaration. From left to right in a given `COM` list, all variables following a numeric type word have that precision until another type word appears in the list. Thus, both H(5) and J are short precision.

`COM` statements in separate programs that are linked with the `CHAIN` statement must agree in number and type of variable. Variables held in common are reset to undefined values by executing `SCRATCH`, `RUN`, or `INIT`.

About Variable Declarations

- The `COM` statement must be used in a program, not from the keyboard, and may not appear within a function definition.
- The location in a program of `DIM`, `COM`, type declaration is arbitrary, though they must be after an `OPTION BASE` statement and before any other reference to the dimensioned variable.

- The `DIM` statement need not be used to assign memory space for strings with 18 characters or less or for arrays that have upper bounds of 10 or less. Thus, you do not need to use `DIM` with an array A(5,5) (of 25 or 36 elements depending on the lower bound) or a string C$="SQUARES". But array A(5,5) will be implicitly dimensioned to be A(10,10) and string C$ will be implicitly dimensioned to have 18 characters rather than 7 (the number of characters in "SQUARES"). Thus, you may wish to use `DIM` to conserve memory with small arrays and strings.
- A program can have more than one `DIM`, `COM`, or type declaration statement, but the same variable name can be declared only once in a program. Therefore, arrays of differing dimensions or variables of different types cannot have the same name. But the same name may be used for a simple numeric, a string, and a numeric array.

String Expressions

The simplest form of a string expression is text within quotes. This is called a *literal* string and can be made up of any characters excluding quotation marks.

For example, execute the first two statements:

```
C$= " STRING "
DISP C$;C$
 STRING   STRING
```

This string expression contains eight characters; two spaces and the word "STRING". Quotation marks are not included in a literal string because they mark the beginning and end of the string.

The forms that a string expression can take are:

- Text within quotes.
- String variable name.
- Substring.
- String concatenation operation (`&`).
- String function.
- Any logical combination of the above.

As with numeric expressions, a string expression can be enclosed in parentheses if necessary.

In this section, we discuss substrings and string functions.

Thus far, you have learned to assign a literal string to a string variable and to join two strings together using the ampersand (`&`) as the string concatenator (page 52).

You have also seen that unless the size of a string variable is specified in a `DIM` statement, it is implicitly dimensioned to be a maximum of 18 characters in length.

```
10 DIM A$[15], F$[28], H$[100]
```

The statement above dimensions string variable A$ to be a maximum of 15 characters, F$ to be a maximum of 28 characters, and H$ to be a maximum of 100 characters. Brackets (not parentheses) must surround the number of characters to be included in the string variable.

Substrings

A substring is a part of a string made up of zero or more contiguous characters. A substring is specified by placing subscripts in brackets after the string name. There are two forms a substring can have:

- String variable name [character position]

  The character position is a numeric expression which is rounded *(not truncated)* to an integer. The substring is made up of that character and all following it.

- String variable name [beginning character position , ending character position]

  This substring includes the beginning and ending characters and all in between. The character positions must be within the dimensioned number of characters. If the first subscript is exactly one greater than the second subscript, the null string (" ") is specified.

Example: Suppose we dimension and assign string A$ as follows:

```
DIM A$[25]
A$="A STRING OF 25 CHARACTERS"
```

Spaces and blanks are also characters.

Now look at the various examples of substrings of A$:

```
A$[5] = RING OF 25 CHARACTERS
A$[15] = CHARACTERS
```

One subscript denotes a substring from that character position to the end of the main string.

```
A$[1,8] = A STRING
A$[13,14] = 25
A$[5,8] = RING
```

Two subscripts denote a substring that includes the characters in the positions specified and all characters in between.

Modifying String Variables

There are a variety of ways that you can modify a string or substring by another string or substring. For instance, a part of a string can be changed or characters can be added or deleted. The modifying string can be any string expression.

The length and content of a modified string depend not only on the characteristics of the modifying string, but also on the number of subscripts given for the original string.

Replacing a String

You can replace the complete string of characters with another string using an assignment statement.

For example:

Press	Display	
A$ = "HELLO" (END LINE)	A$ = "HELLO"	Assigns string to A$.
B$ = "GOODBYE" (END LINE)	B$ = "GOODBYE"	Assigns string to B$.
B$ = A$ (END LINE)	B$ = A$	Assigns B$ the expression in A$.
B$ (END LINE)	B$	Recalls B$ to verify.
	HELLO	

As you can see, B$ was reassigned the string in A$. When no subscripts are specified for either variable, the string is completely replaced with the new string. You can also reassign string variables by typing the new string within quotes.

For example:

Press	Display	
A$ ="HI" (END LINE)	A$="HI"	A string variable contains the characters
A$ (END LINE)	A$	most recently assigned to it.
	HI	
A$ ="BYE" (END LINE)	A$="BYE"	
A$ (END LINE)	A$	Recalls A$ to verify.
	BYE	

Replacing Part of a String

After you have assigned a character string to a variable, you can replace one substring with another substring. The original string can be lengthened or shortened. But if you attempt to lengthen the string beyond its dimensioned length, you will cause an error.

Change substrings by specifying the subscripts of the characters to be changed and the new substring.

For example:

Press	Display	
H$ ="HAPPENING" (END LINE)	H$="HAPPENING"	Assigns substring to H$ beginning with
H$[7] ="STANCE" (END LINE)	H$[7]="STANCE"	character 7.
H$ (END LINE)	H$	
	HAPPENSTANCE	Lengthened string.
H$[5] ="ILY" (END LINE)	H$[5]="ILY"	Assigns substring to H$ beginning with character 5.
H$ (END LINE)	H$	
	HAPPILY	New string.

If characters added to a string are not contiguous (in other words, some character positions are left unassigned), blank spaces will fill the unassigned characters in the string.

For example:

```
W$="C. "
W$[5]="JACKSON"
W$
C.  JACKSON
```

Since the third and fourth character positions of W$ have not been assigned characters, they are filled with blank spaces.

You can also replace the beginning or the middle of a string with another substring. Do this by using two subscripts to specify the first and last character positions of the substring to be replaced.

If the new substring is shorter than the substring that you replace, the remainder of the new substring is replaced with blanks; if the new substring is longer than the one you replace, the remainder of the new substring is truncated.

For example:

Press	Display	
Z$ ="HEPTAGON" (END LINE)	Z$ = "HEPTAGON"	
Z$[1,3] = "PEN" (END LINE)	Z$[1,3] = "PEN"	Replaces characters 1 through 3 of Z$ with specified substring.
Z$ (END LINE)	Z$	
	PENTAGON	
Z$[1,4] = "HEX" (END LINE)	Z$[1,4] = "HEX"	Since you replaced characters 1 through 4 with a string of length 3, the fourth character is a blank.
Z$ (END LINE)	Z$	
	HEX AGON	

Press	Display	
Z$[1,4]= "DODEC" (END LINE)	Z$[1,4]="DODEC"	If you try to replace four characters with five characters, the fifth character is truncated.
Z$ (END LINE)	Z$	
	DODEAGON	

Another way to specify the null string is to make the first subscript one larger than the second subscript in a substring. Thus the following statements are equivalent:

```
N$=""
N$=A$[4,3]
N$=C$[8,7]
```

Each specifies no blanks, no characters. (A$ and C$ must have been previously assigned values or an error occurs.)

String Functions

The HP-85 provides seven different functions to enable you to determine the length of a string and analyze and manipulate its contents.

These functions are:

String Function (Parameter)	Meaning
`LEN`(string)	Length of string.
`POS`(string 1, string 2)	Position of string 2 in string 1.
`VAL`(string)	Returns the numeric value of a string expression composed of digits.
`VAL$`(numeric expression)	Generates a string representing the numeric value of a numeric expression.
`CHR$`(numeric expression)	Converts a numeric expression to the appropriate character.
`NUM`(string)	Returns the decimal value of the first character of the string.
`UPC$`(string)	Converts all lowercase letters in string to uppercase letters.

The Length Function

The `LEN` *(length)* function returns the number of characters in a string expression.

`LEN`(string expression)

The current length of a string expression is returned. Remember, a string variable isn't always "full"; the length isn't necessarily the maximum length that you give it in a `DIM` statement.

Examples:

`LEN("123")`	Length of string "123".
`3`	Result of `LEN` function: 3 characters long.
`A$="length*width"`	Assigns string to variable A$.
`LEN(A$)`	Finds length of A$.
`12`	Result: 12 characters long.
`LEN(A$[1,6])`	Finds length of first six character positions in A$.
`6`	Result: 6 characters long.

Notice that the string expression may be quoted text, a string variable name, or a substring. The expression must be enclosed within parentheses.

Example: Write a program that will let you enter a character string of up to 40 characters in length. Then, using the `LEN` function, compute and display the word with the characters in reverse order. For instance, if you input `CAT`, the program should display `TAC`.

Program	Explanation
`10 DIM W$[40], R$[40]` `20 R$=""`	Dimensions the string variables to be a maximum of 40 characters long. Initializes R$ to the null string.
`30 DISP "WORD";`	Displays a message to prompt an input.
`40 INPUT W$`	Inputs a word.
`•50 FOR I=LEN(W$) TO 1 STEP -1`	Uses length of word for loop counter and counts in reverse order.
`60 R$=R$&W$[I,I]`	With the string concatenator, adds characters to variable R$ in reverse order.
`70 NEXT I`	Defines end of `FOR-NEXT` loop.
`80 DISP R$`	Displays reversed word.
`90 END`	

After you enter the program above, try spelling some words backwards!

(RUN)

```
WORD?
CAT
TAC
```

The program reverses the order of the characters in the string—including spaces between words.

(RUN)

```
WORD?
YELLOW PAGES
SEGAP WOLLEY
```

(RUN)

```
WORD?
INTERNATIONAL
LANOITANRETNI
```

The Position Function

The `POS` *(position)* function determines the position of a substring within a string.

`POS(`in string expression `,` of string expression `)`

If the second string is contained within the first, the `POS` function returns the position of the first character of the second string within the first string. If the second string is not contained within the first string, or if the second string is the null string, the value returned by the function is zero. If the second string occurs in more than one place within the first string, only the first occurrence is given by the function.

Examples:

Example	Explanation
`POS("ABABC1234","123")` `6`	Finds position of second string in first string. Result: second string begins at sixth character position.
`POS("ABABC1234", "AB")` `1`	Result: first occurrence of "AB" within first string.
`A$="COMPOSER"` `B$="POSE"`	
`POS(A$,B$)` `4`	Position of B$ in A$. Result: B$ begins at fourth character position of A$.

Be sure to separate the string expressions by a comma.

Converting Strings to Numbers

Normally, the characters in a string are not recognized as numeric data and can't be used in numeric calculations. That's because you usually want the string to be quoted literally, character for character.

With the VAL *(value)* function the numeric value of a string or a substring of digits, including an exponent, can be used in calculations.

VAL(string expression)

For example, suppose

```
A$="ROMEO, J; 257684321"
```

If you want to obtain the numeric value rather than the literal substring of "257684321", you must use the VAL function:

```
VAL(A$[10])
 257684321
```

Gives *numeric value* of A$ from character 10 to end of string.
This is a number, *not* a string. Note that the system indents positive numbers; i.e., the space before the number is for the sign (if any). Now, this number can be assigned to a numeric variable and it can be used in numeric calculations.

```
A=VAL(A$[10])

A$[10]
257684321
```

This is a *substring* of A$. A$[10] is not a numeric value. Notice that no space precedes the number to specify the sign. The string cannot be assigned to a numeric variable nor can it be used in numeric calculations.

When you use the VAL function, the first character in the string to be converted must be a digit, a plus or minus sign, a decimal point, or a space. A leading plus sign or space is ignored; a leading minus sign is taken into account. The remaining characters in the string or substring must be digits, a decimal point, or an E. An E character after a numeric and followed by digits (including sign) is interpreted as an exponent of 10.

Examples:

```
VAL("4E-2")
 .04
```

The function outputs the number in standard format.

```
VAL("-1234.567")
-1234.567
```

A string can contain more than one number. All contiguous numerics are considered a part of the number until a non-numeric is reached in the string.

Example:

```
B$="43 SCORE 59"
VAL(B$)
 43
```

As long as the first character is a numeric, the VAL function converts the string to a number until it reaches a non-numeric character.

```
VAL(B$[9])
 59
```

But you can convert the remaining numerics in the string by subscripting the string variable. Here we specify the numeric value of B$ from character position nine to the end of the string.

Converting Numbers to Strings

The VAL$ function is nearly the inverse of the VAL function. With the VAL$ function, you can convert a number to a string representation of the number in standard format.

VAL$(numeric expression)

Examples:

```
V$=VAL$(120)
V$
120
```

Result of executing V$; V$="120".

```
W$=VAL$(4*8)
W$
32
```

W$ = "32".

```
X$=VAL$(SQR(64))
X$
8
```

X$ = "8".

Character Conversions

If you look at the table in appendix C, you'll see that a decimal number corresponds to every character, symbol, and key. The numbers range from 0 through 255. There are three functions, CHR$, NUM, and UPC$, that enable you to convert a number to its corresponding character, convert a character to its corresponding decimal number character code, and convert small letters to capital letters.

Numbers to Characters

The CHR$ *(character)* function converts a numeric value in the range −32768 through 32767 into a string character. Any number outside the range 0 through 255 is converted MOD256 to that range. Any number outside the range −32768 through 32767 is "⊢" (the same as CHR$(255)).

CHR$(numeric expression)

Examples:

```
CHR$(35)
#
CHR$(126)
Σ
CHR$(16)
θ
CHR$(8)
Δ
CHR$(136)
Δ
```

One of the most used numbers is 34 (this is the decimal number for a quotation mark). Often you may want to use the quotation mark in a PRINT or DISP statement. Since the beginning and end of a literal message is defined by a quotation mark, you cannot use the mark itself. Instead, use CHR$(34):

```
DISP "The answer is, ";CHR$(34);
"YES";CHR$(34);"; you lose!"
The answer is, "YES"; you lose!
```

Characters to Numbers

The NUM (*numeric*) function converts an individual string character to its corresponding decimal value.

NUM(string expression)

Thus you can find the decimal number code of the corresponding character without having to look it up in the table in appendix C.

If more than one character is included in the string expression, the NUM function finds the decimal equivalent of the first character.

Examples:

NUM("¿") 1	To display ¿, type A while holding down the CTRL key (A^c).
NUM("#") 35	
NUM("θ") 16	To display θ, type P while holding down the CTRL key (P^c).
NUM("$^&*") 36	Converts only first character of string.

Lowercase to Uppercase Conversion

The UPC$ *(uppercase)* function enables you to convert a string with lowercase letters to a string composed of all uppercase letters.

UPC$(string expression)

Examples:

`M$="yes"`	Assigns M$ the string shown in lowercase letters.
`N$=UPC$(M$)`	Assigns N$ that string in uppercase letters.
`N$`	Recalls N$.
`YES`	
`UPC$("SOMEupsomeDOWN")`	The string need not be composed of all lowercase letters to be converted to all uppercase letters.
`SOMEUPSOMEDOWN`	

As you may have noticed from the table of characters in appendix C, lowercase letters have different decimal values than uppercase letters. The uppercase function allows strings to be compared without regard to upper and lowercase. For example, part of a program might be:

```
:
30 INPUT A$
40 IF UPC$(A$[1,1])="Y" THEN 80
```

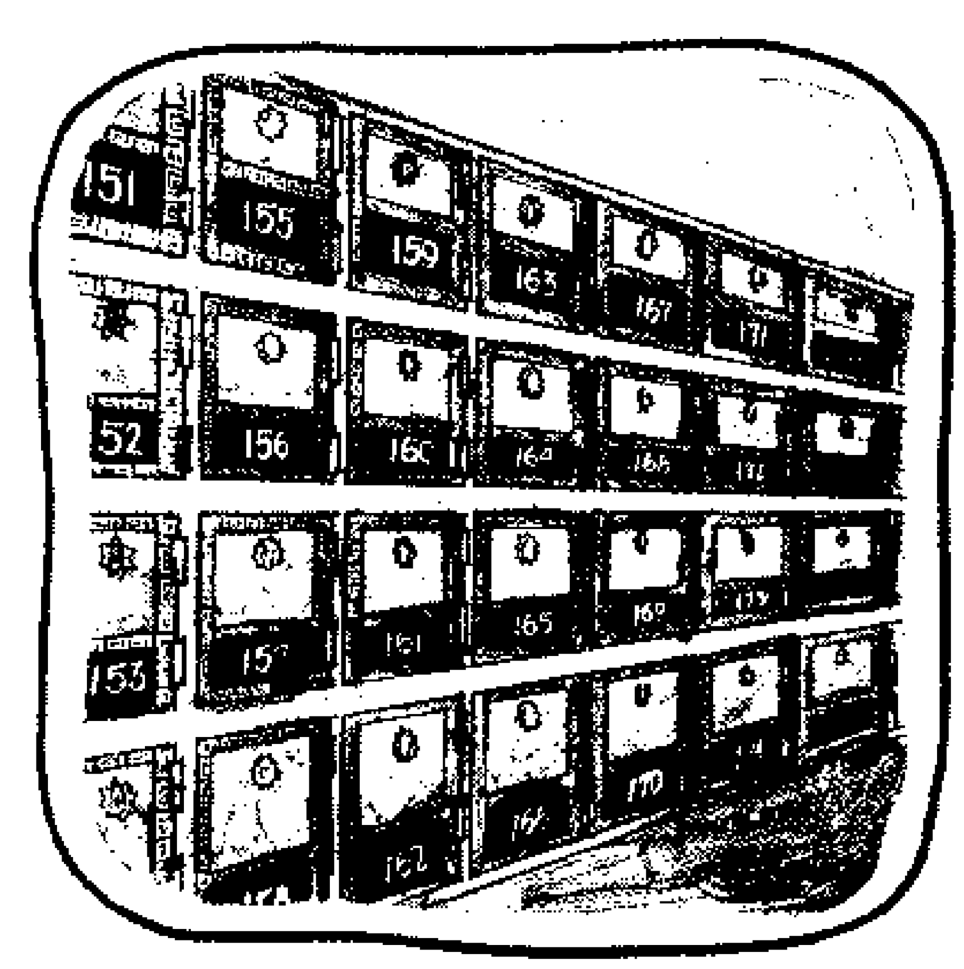
User may enter Y, y, yes, YES, etc., and the program will branch to statement 80.

Assigning Values to Variables in a Program

You can assign values to variables using a program statement or by an input from the keyboard. Thus far, we have discussed the LET *(assignment)* statement and the INPUT statement with regard to simple variables. This section covers assignments to the elements of arrays, initializing variables, and three more statements that are used for assigning values to variables: READ, DATA, and RESTORE. These statements are useful when you have a large amount of data that is reused in different places in the program.

Assigning Values to Array Elements

Elements of an array are assigned values in the same manner as simple variables: from the keyboard or within a program. But a particular element must be referenced by its subscripts. For instance, M(1,2) refers to an element in array M and may be assigned a value and used in calculations like a simple variable.



Example:

```
10 OPTION BASE 1
20 DIM M(3,4)
•30 LET M(1,2)=10
40 A=M(1,2)/7
50 PRINT M(1,2),A
60 END
```

Dimensions a 3 by 4 array M.
Assigns element M(1,2) the value 10.
You can use this element in calculations.

If we had not dimensioned array M, it would have been implicitly dimensioned with upper bounds of 10 for each subscript.

The program below enables you to input values from the keyboard. The FOR-NEXT loop is the most efficient means of manipulating array variables.

Example:

```
10 OPTION BASE 1
20 DIM A(5)
30 FOR I=1 TO 5
•40 INPUT A(I)
50 NEXT I
60 FOR I=1 TO 5
70 PRINT "A(";I;")=";A(I)
80 NEXT I
90 END
```

You must assign each array element its value, individually.
Assigns the elements of array A the values you input.

Then prints the array elements.

Run the program, now, with the numbers 33, 48, –16, 3, and 10.

```
PRINTALL
RUN
?
33
?
48
?
-16
?
3
?
10
A( 1 )= 33
A( 2 )= 48
A( 3 )=-16
A( 4 )= 3
A( 5 )= 10
```

You can see that assigning values to array elements with a FOR-NEXT loop is indeed faster and easier than using an assignment statement for each element, especially with large arrays.

Let's see how this is done with two-dimensional arrays:

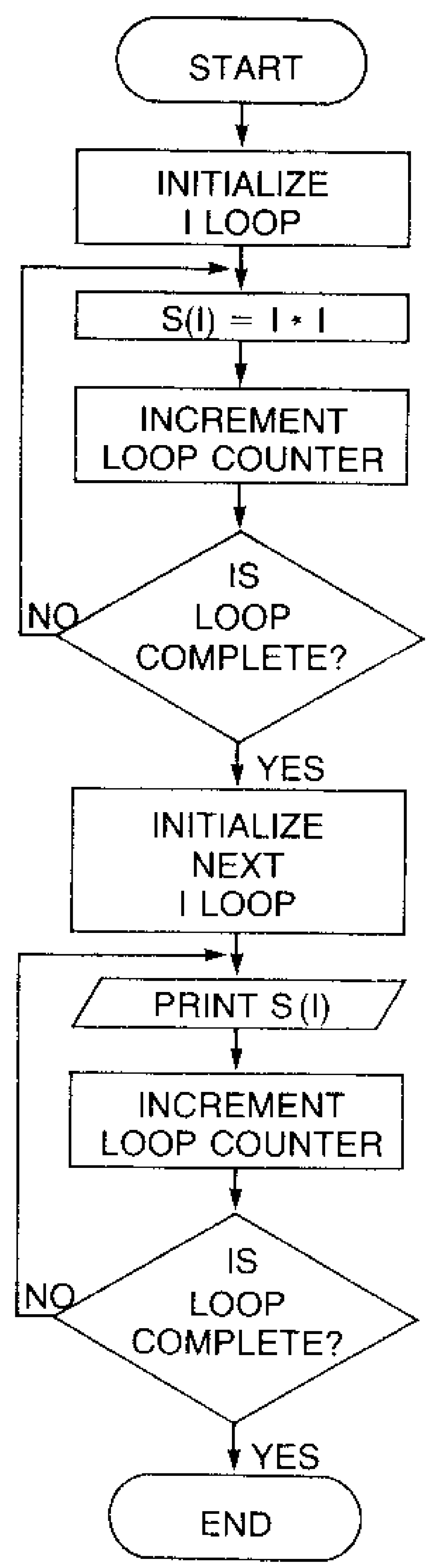
Example:

```
•10 OPTION BASE 1
 20 DIM K(3,5)
 30 FOR I=1 TO 3
 40 FOR J=1 TO 5
 50 DISP "ROW";I;"COLUMN";J
•60 INPUT K(I,J)
 70 NEXT J
 80 NEXT I
 90 END
```

Lower bound of array is 1.
Dimension 3 by 5 array K.
Nested FOR-NEXT loops.
First input all elements of row 1, then all elements of row 2, etc.

There are many ways to assign array elements values within the program. The following program uses the loop counter to produce a list of squares of consecutive integers from 1 to 15.

```
 10 OPTION BASE 1
 20 DIM S(15)
 30 FOR I=1 TO 15
•40 S(I)=I*I
 50 NEXT I
 60 FOR I=1 TO 15
 70 PRINT "S(";I;") =";S(I)
 80 NEXT I
 90 END

RUN
S( 1 ) = 1
S( 2 ) = 4
S( 3 ) = 9
S( 4 ) = 16
S( 5 ) = 25
S( 6 ) = 36
S( 7 ) = 49
S( 8 ) = 64
S( 9 ) = 81
S( 10 ) = 100
S( 11 ) = 121
S( 12 ) = 144
S( 13 ) = 169
S( 14 ) = 196
S( 15 ) = 225
```



Initializing Variables

It's good programming practice to initialize (set) variables to their starting values in a program before you use them. All numeric variables are initialized to undefined values by RUN or INIT. Thus, if you access the variable before it is defined, an error will occur.

As long as you assign the variable a value before it is accessed, you will not err. For instance, the following programs on the right cause warning messages (with DEFAULT ON) or error messages (with DEFAULT OFF) to occur. With DEFAULT ON, the default value of 0 is assigned to uninitialized variables, and the specified computations or programming operations will be performed. But a warning message is displayed to alert you to the error.

Correct

```
•10 T=0
 20 INPUT W
 30 T=T+W
 40 DISP T;W
 50 END
 RUN
 ?
 5
  5   5
```

Incorrect

```
10 INPUT W
20 T=T+W
30 DISP T;W
40 END
```

Error here; using T when it has not yet been assigned a value.

```
RUN
?
5
Warning 7 on line 20 : NULL DATA
 5   5
```

Correct

```
 10 DIM A(6,4)
•20 A(3,3)= 3
 30 DISP A(3,3)+3*2
 40 END

 RUN
  9
```

Incorrect

```
10 DIM A(6,4)
20 DISP A(3,3)+3*2
30 END
```

Error occurred here; using a variable that has not yet been assigned a value.

```
RUN
Warning 7 on line 20 : NULL DATA
A
 6
```

If you don't plan to assign values to all array elements in a program but want to be able to access any of them, you can easily initialize them using FOR-NEXT loops. For example, suppose we want to initialize all elements of array A to 0:

```
 10 DIM A(6,4)
 20 FOR I=0 TO 6
 30 FOR J=0 TO 4
•40 A(I,J)=0
 50 NEXT J
 60 NEXT I
 :
```

The READ and DATA Statements

Many programs require you to enter large numbers of data items into the computer. You can accomplish this with the INPUT or LET statements, though it may be cumbersome to do so. If you had used an INPUT statement and decided to run the program with the same values at a later date, you would have to reenter all of the data once again. BASIC programming language provides a more convenient means of assigning values to variables in these instances—by using the READ and DATA statements.

The READ and DATA statements work together to assign values to variables within a program.

READ variable name$_1$ [, variable name$_2$ ...]
DATA constant or string [, constant or string ...]

The READ statement specifies the variables whose values are to be assigned from within the program. The variables in a READ statement may be simple variables, subscripted variables, or string variables, and they must be separated by commas.

The DATA statement contains a list of the numbers or character strings that will be assigned to the variables in the READ statement. The numbers or strings must be separated by commas. Each DATA item must correspond to the appropriate variable of the same type in a READ statement.

```
• 10 READ N$,A(1),C
  20 IF C>3 THEN PRINT N$;A(1);C
• 30 DATA "NAME", 43,6
  40 END
  RUN
  NAME 43  6
```

These statements cause "NAME" to be assigned to N$, 43 to A(1), and 6 to C.

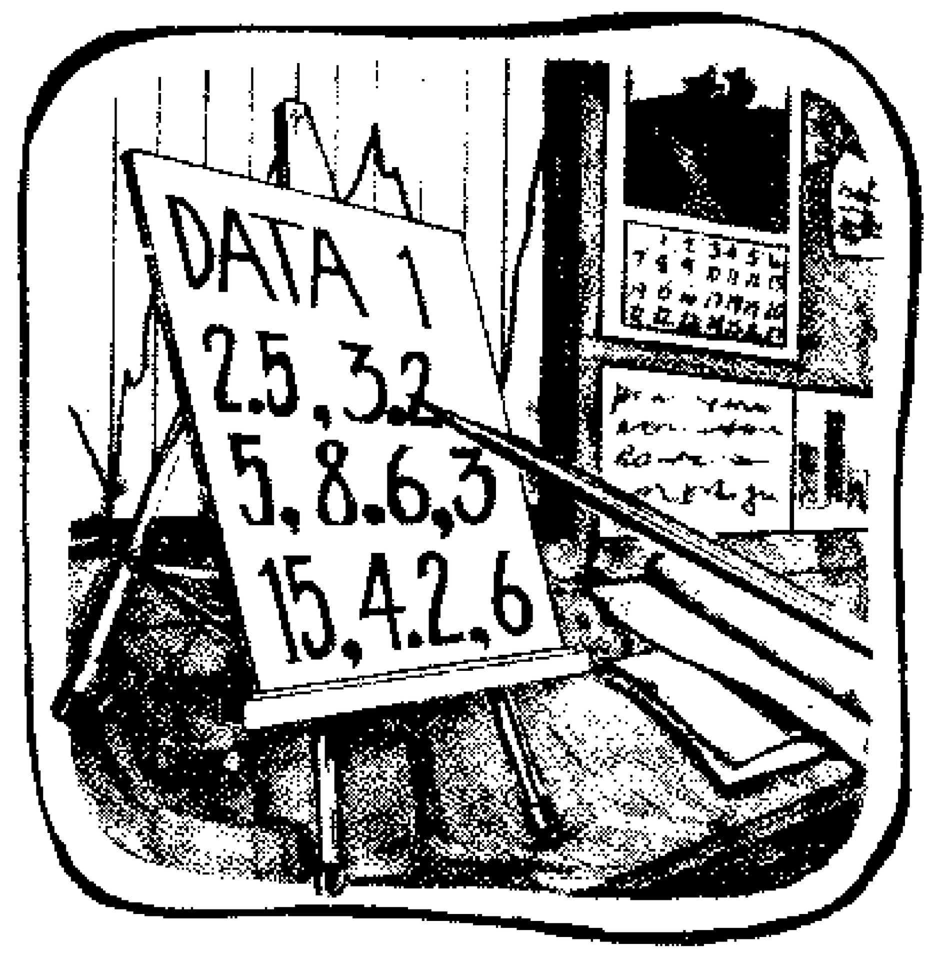
In DATA statements, literal text may be quoted *or* unquoted.

The DATA statement is declaratory and is simply ignored in a program if there is no corresponding READ statement. Therefore, a DATA statement need not correspond exactly with the READ statement. Your DATA statements can contain more items than accessed by the READ statement, and they can be positioned anywhere in a program except after THEN or ELSE in an IF... THEN statement. The important point is that the order of DATA statements within a program determines the order of their use.

For example, load the following program and run it:

```
• 10 DATA 24,8.3,17,19,3.2,58
  20 FOR I=1 TO 4
• 30 READ X
  40 PRINT X; "SQUARED=";X^2
  50 NEXT I
  60 END


  RUN
  24 SQUARED= 576
  8.3 SQUARED= 68.89
  17 SQUARED= 289
  19 SQUARED= 361
```

Extra data items are ignored.



The system uses an internal mechanism, called a "pointer," to locate the data element that is to be read. The left-most element of the lowest-numbered `DATA` statement is read first. After this element is read, the data pointer repositions itself one element to the right and continues to do so each time another data item is read.

After reading the last element in a `DATA` statement, the data pointer locates the next higher-numbered `DATA` statement (if any) and repositions itself at the first element in that statement. But if there are no higher-numbered `DATA` statements, the data pointer remains at the end of the last `DATA` statement; any effort to read additional data will cause a `NO DATA` message to be displayed.

For example:

```
AUTO 10
•10 READ N
 20 FOR P=1 TO N
 30 READ D,D1
 40 DISP D^2-D1
 50 NEXT P
•60 DATA 4
•70 DATA 9,1,8,4,7,9
 80 END
```

Assigns 4 to N.

First, assigns 9 to D and 1 to D1; then, 8 to D and 4 to D1; finally 7 to D and 9 to D1.

```
RUN
80
60
40
Error 34 on line 30 : NO DATA
```

Since N=4, program tries to read more values for D and D1, but finds no more data available.

The `DATA` statement in the last program can be entered in a variety of ways. For example, the following representations are equivalent:

Note: We do not change the order of the items themselves.

```
60 DATA 4,9,1,8,4,7,9
```

or

```
60 DATA 4,9
62 DATA 1,8
64 DATA 4,7
66 DATA 9
```

or

```
65 DATA 4,9,1
68 DATA 8,4,7,9
```

Even though the data items can be entered in one or several `DATA` statements, as shown above, the order in which they appear must correspond exactly with the order in which you access them.

The READ and DATA statements are often used to assign values to array elements.

Example:

```
 10 FOR I=1 TO 5
 20 FOR J=1 TO 5
•30 READ A(I,J)
 40 NEXT J
 50 NEXT I
•60 DATA 1,2,3,4,5,2,4,6,8,10,3,6
,9,12,15,4,8,12,16,20,5,10,15,20
,25
 70 FOR I=1 TO 5
 80 FOR J=1 TO 5
 90 PRINT A(I,J);
 100 NEXT J
 110 PRINT
 120 NEXT I
 130 END

 RUN
 1  2  3  4  5
 2  4  6  8  10
 3  6  9  12  15
 4  8  12  16  20
 5  10  15  20  25
```

Notice that you must READ each array element one at a time.

The semicolon causes printed items to be retained in the print buffer.
The extra print statement forces a print after each row.

You can see a number of things about READ and DATA from the examples above. To recap:

- It doesn't matter where the DATA statement is, in relation to the READ statement, as long as the data items correspond to the variables in the READ statement in order and in type.
- More than one READ statement can access a DATA statement. As each READ is executed, the DATA pointer moves to the next data item. There must be at least as many items in the set of DATA statements as there are variables in the READ statements. Extra data items are ignored.
- The items in a data list must be either numbers or strings; variables and formulas are not allowed. A data item accessed by a string variable in a READ statement may be quoted or unquoted in the DATA statement.
- Variable assignments made with READ and DATA statements are part of a program, contrasted with variable assignments made with the INPUT statement. Thus, the data is stored with the program and will remain with the program until the DATA statement, itself, is changed.

Rereading Data: The RESTORE Statement

Up to this point we have been able to access DATA items only once in a program. Once the data pointer moves past the last data item in the last DATA statement, an additional READ statement causes "Error : NO DATA". Of course, if the data items have been assigned to variables, you can use the same values again by using the variable name.

But certain programs may require some, if not all, of the data to be read more than once. BASIC provides the RESTORE statement just for this purpose:

RESTORE [statement number]

The RESTORE statement resets the data pointer to the first item of the specified statement (or the first item of the lowest-numbered DATA statement in the program if no statement is specified) each time it occurs within a program.

For example:

```
10 READ A,B,C$
20 IF C$#"N" THEN 50
30 RESTORE
40 GOTO 10
50 PRINT A,B
60 GOTO 10
70 DATA 5,10,"Y",15,20,"Y",3,9,"
   N"
80 END

RUN
5                   10
15                  20
5                   10
15                  20
5                   10
15                  20
5                   10
15                  20
```

PAUSE

As you can see, the data pointer is continually reset to 5 in the DATA statement each time "N" is read and RESTORE is executed.

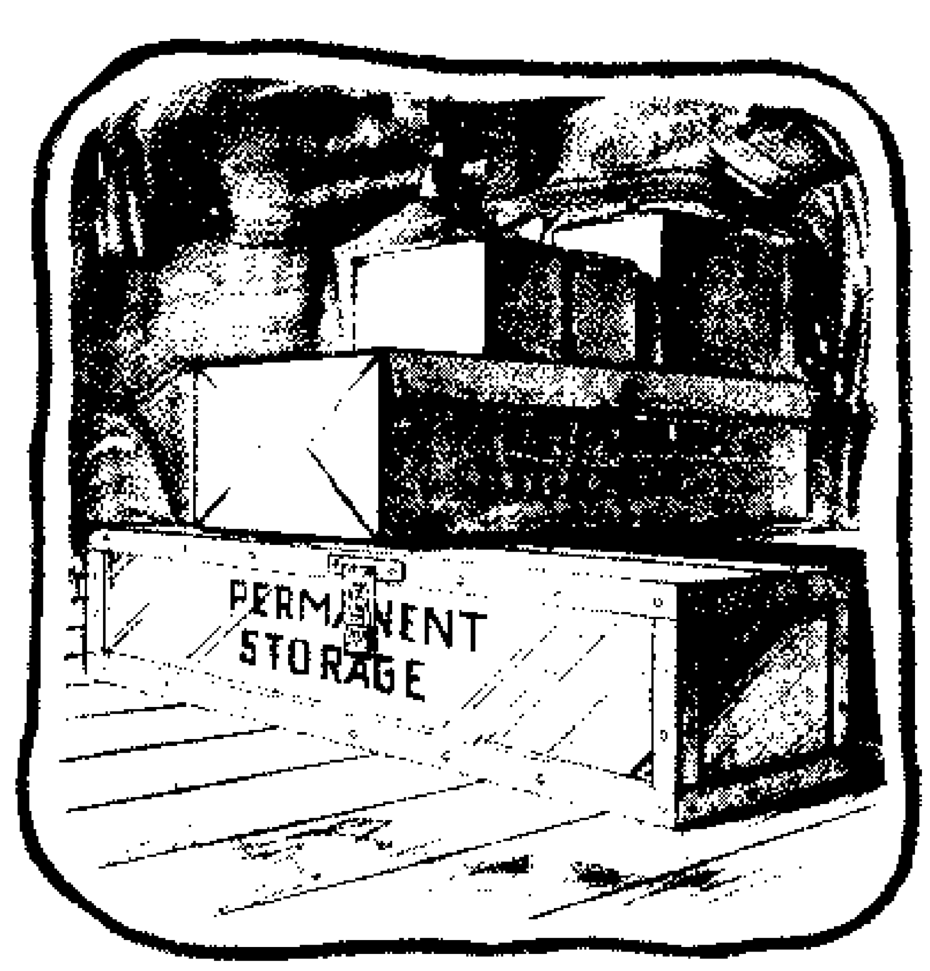
System Memory and Variable Storage

Memory

The HP-85 uses two types of memory: **read/write memory** (or random access memory—RAM) and **read only memory** (ROM). Read/write memory is used to store programs and data. When you store a program or data, you "write" into memory. When you access a line of your program or data element, you "read" from memory, thus the term read/write. Read/write memory is temporary; it can be changed or erased.

Programs and data in read/write memory can be saved for future use by recording the information on a tape cartridge or other storage medium.

Read only memory differs in that it is permanent. When the machine is turned off, the contents of the read/write memory are lost, whereas the read only memory is unaffected. ROM modules can be plugged into the slots in the back of the machine, making it possible to expand the language and capabilities. A small amount of read/write memory is used by some plug-in ROMs. This area is called "working storage."



Storing Variables

Byte is computer language for a "memory location" composed of eight *bits* (*b*inary dig*its*). It is the basic unit of information to or from memory. A kilobyte is a unit of 1,024 bytes and is abbreviated as "K".

The HP-85 has 16K bytes (or 16,384 bytes) of read/write memory; 14,576 are available for your use. The memory can be expanded to 32K bytes of which 30,704 are available for your use with the memory module; refer to appendix A.

Use the following tables to determine the number of bytes that variables need in order to be stored in system memory. (Do not confuse storing in system memory with storing on a tape cartridge. Tape cartridge storage will be discussed in section 11.)

Simple Variables	Bytes of Memory
Full precision	10 bytes
Short precision	6 bytes
Integer	5 bytes
String	8 bytes + 1 byte per character

Array Variables	Bytes of Memory
Full precision	8 bytes + 8 bytes per element
Short precision	8 bytes + 4 bytes per element
Integer	8 bytes + 3 bytes per element

You have already noticed that at the end of every program listing, the HP-85 displays the number of bytes (or memory locations) remaining in system memory. Press INIT or execute the INIT (*initialize*) command before LIST or PLIST so that the memory displayed will include the memory required for allocated variables.

If you do not wish to LIST the entire program to recall the memory, type LIST and then a statement number larger than any in the current program. For instance, you could execute LIST 9999 to display the number of bytes left.

You need not have a program in memory to execute LIST. If there is no program, the system merely outputs the number of bytes available.

Conserving Memory

Large programs that involve large amounts of data sometimes need more memory than is available for use. You can conserve memory by:

1. Limiting the use of REM statements and comments in a program. This limits program readability and documentation, but it does conserve memory.
2. Using SHORT and INTEGER precision array variables, whenever possible or convenient, rather than full precision. This is a very good way to conserve memory in a program that has a lot of data and is most useful when dealing with large arrays.
3. A third way to conserve memory is to break a program down into several sections and STORE each s into a different file. Then each section of the program can be brought into memory, one at a time, usi CHAIN statement. (Refer to section 11.)

4. Combine statements using "@". This reduces program readability, but it does conserve memory by three bytes per line. For example:

```
10 BEEP @ BEEP
```

is seven bytes of information while

```
10 BEEP
20 BEEP
```

is 10 bytes of information.

Problems

8.1 Here is your chance to invent some new words. Write a program that accepts a base string and a first-letter string, and then prints the "words" formed by combining each of the first letters with the base string, but omitting those that would begin with a double letter.

8.2 One light-year is the distance light travels in one year—approximately 9 trillion kilometers. The distances (in light-years) of the 27 stars within 15 light-years of our solar system are listed below. Write a program that will group these distances into intervals of 1 light-year (0-1 through 14-15) and determine the number of stars in each interval. After printing these results, the program should request an interval number, 1 through 15, and print the distances for the stars in that interval. Use an INTEGER array for accumulating the interval distributions. Use a simple SHORT variable for the actual distances and READ them one at a time. A RESTORE statement is necessary for the second part of the program.

STAR DISTANCES (light-years)			
4.3	10.3	11.5	12.8
5.9	10.7	11.6	13.1
7.6	10.8	11.7	13.1
8.1	10.8	11.9	13.9
8.6	11.2	12.2	14.2
8.9	11.2	12.5	14.5
9.4	11.4	12.7	

8.3 The world record, set in 1970, for the 30-kilometer run is 1:31:30.4 (1 hour, 31 minutes, 30.4 seconds) and is held by Jim Adler of Britain. In 1974, Bernd Kannenberg of West Germany set a world record of 2:12:58.0 for the 30-kilometer walk. Write a program that accepts an individual's time for a 30-kilometer course and calculates the average speed according to

$$\text{Speed (m/s)} = \frac{30{,}000\ \text{(m)}}{\text{Time (s)}}$$

The time is to be specified in *hours:minutes:seconds* format (including colons). Use the POS function to locate the colons, and the VAL function to extract the numerical values from the string. Also, use the program to calculate the speed of Sergei Saveliev of the USSR, who set a world record of 1:30:29.38 for the 30-kilometer Nordic ski event in 1976, and for Clem Turvy on his motorcycle, covering 30 kilometers in 26:44 (26 minutes, 44 seconds).

8.4 Although a string variable may not be declared to be an array, it is possible to use substrings of a string variable to achieve the effect of a "string array." For example, if the words representing the numbers 0 through 9 are strung together with proper spacing, any one word is readily accessible by determining the first and last substring specifiers corresponding to the word (similar to the subscript of an array element). Using this concept and concatenation, write a program that counts from 0 to 99 in this way:

```
ZERO
ONE
:
NINE
ONE  ZERO
ONE  ONE
:
NINE NINE
```

8.5 Farmer Flem Snopes wants to install irrigation sprinklers in his three strawberry patches. The table below gives coverage diameters for a particular sprinkler design at various water pressures and nozzle options. Write a program based on this table that asks for the width of the irrigated strip (which determines the minimum coverage diameter) and the available water pressure at that location, and then specifies the appropriate nozzle option. Use the program to find the nozzle options for Snopes' east strawberry patch (150 feet wide, 75 psi pressure), his southeast patch (140 feet wide, 75 psi pressure), and his far-north patch (140 feet wide, 60 psi pressure).

Coverage Diameter (feet)

Nozzle Option	A	B	C	D
Water Pressure (psi)				
60	124	133	138	142
65	126	136	141	146
70	129	139	144	149
75	132	142	147	152
80	134	145	150	155

More Branching

There's much more to branching operations on the HP-85 than IF ... THEN and GOTO. The system enables you to define your own functions and use them in programs, just as you use the built-in functions. For longer program segments or routines that are often repeated within a program, the BASIC language provides subroutines that can be accessed any number of times within a program. In addition, the system contains three timers that can interrupt a program in the time intervals of your choice. Last, but not least, we'll discuss the special function keys—how to define them so that when pressed, they immediately cause special branching in a program.

Defining a Function

If a numeric or string operation has to be evaluated several times, it is convenient to define it as a function. With the DEF FN *(define function)* statement, you can define your own functions within a program and reference them in exactly the same manner that you reference the system's built-in functions. A function must be defined in the same program that references the function. The definition can appear anywhere in the program, before or after the function is referenced.

DEF FN numeric variable name [(parameter)] [= numeric expression]

DEF FN string variable name [(parameter)] [= string expression]

Once a function is defined, it can be used by referring to the function name. A numeric function name must consist of the letter FN followed by a numeric variable name. A string function name is a numeric function name followed by a dollar sign, $. If the function requires an argument, then it must appear immediately after the function name, enclosed within parentheses. The parameter may be any simple numeric or string variable name. Array names are not allowed. The length of a string argument passed between a function and the main program defaults to 18 characters. But you *can* allocate a larger string in the function definition. Refer to page 151.

Single-Line Functions

The simplest form of a function definition is the single-line function. The function is defined in one DEF FN statement with an equals sign separating the function name from the expression assigned to the function.

For example, the following program defines FNX2 as the X^2 function and then uses the function to evaluate 8^2.

```
 10 REM *X SQUARED
•20 DEF FNX2(N)=N*N          Defines function FNX2.
 30 DISP FNX2(8)             Displays FNX2(8).
 40 END

 RUN
  64
```

The parameter, N, in statement 20 is a dummy variable used only in the definition. It is replaced by the actual variable or expression when used to evaluate the function. In this case, N is replaced by 8.

All user-defined functions may have, at most, one argument. The function is evaluated using that argument to return, at most, one value at a time.

But a function need not have an argument. (Recall the `PI`, `EPS`, and `INF` built-in functions.)

Examples:

```
10 DEF FNE$="10-SECONDS"
20 DISP FNE$
30 END
RUN
10-SECONDS
```

```
10 REM *Planck's constant
20 DEF FNH=6.625E-27
30 PRINT FNH;FNH^2
40 END
RUN
6.625E-27   4.3890625E-53
```

A function definition cannot be recursive; in other words, you may not use the function that you are defining in the expression that defines the function or in any user-defined function referenced by that expression. But you may use any other user-defined function that has been fully defined elsewhere in the program, and of course, you can use any of the built-in functions in the definition.

Example: Write a program that defines function `FNR` to round any given number to the hundredths place. Then use `FNR` to display the square roots of 1, 1.5, 2, 2.5, ... , 10.

```
 10 REM *ROUND TO 2 DECIMAL PLAC
    ES
 20 FOR I=1 TO 10 STEP .5
 30 DISP I,FNR(SQR(I))
 40 NEXT I
•50 DEF FNR(D2) = INT(D2*100+.5)
    /100
 60 END
```

Notice the use of non-integer steps.

Defines rounding function.

```
RUN
 1                   1
 1.5                 1.22
 2                   1.41
 2.5                 1.58
 3                   1.73
 3.5                 1.87
 4                   2
 4.5                 2.12
 5                   2.24
 5.5                 2.35
 6                   2.45
 6.5                 2.55
 7                   2.65
 7.5                 2.74
 8                   2.83
 8.5                 2.92
 9                   3
 9.5                 3.08
 10                  3.16
```

Displays square roots of number in left column, rounded to hundredths place.

A function definition is a declaratory statement and may be placed anywhere in the program. It merely defines the function, and is ignored by the program unless it is referenced elsewhere by the function name.

See problems 9.1 through 9.3 at the end of this section for more examples of single-line functions.

Multiple-Line Functions

Often, a single line is not enough to define a function, especially if the function contains lengthy computations or branching operations. Multiple-line functions work much like single-line functions in that the function can contain at most one argument and returns one value. Again, the function definition may be placed anywhere within the program since, as a block of statements, it is non-executable unless it is referenced by the function name. But more sophisticated functions can be defined since a function can be described in a series of statements.

There are three basic parts to the multiple-line function definition:

1. The first statement is the DEF FN statement. It is the only DEF FN statement that may occur within the function definition.
2. The last statement is the FN END *(function end)* statement.
3. At least one of the statements in the function definition should assign the function name a value.

Unlike single-line functions, the function definition is not included in the DEF FN statement. Only the function name and argument (if any) must be declared.

The FN END *(function end)* statement defines the end of a multiple-line function. Its syntax is simply:

```
FN END
```

Any number of statements can be included between the DEF FN and FN END statements. But one of these statements should assign the final value of the function to the function name.

For example, this program defines a function that converts an integer with a decimal base to its octal equivalent.

• 10 DEF FNO(D)	Defines beginning of multiple line function.
20 D=IP(D)	Throw away the fractional part of the number to avoid an error.
30 N8=0	Initializes variables N8 and I.
40 I=1	
50 Q=IP(D/8)	
60 N8=N8+(D-Q*8)*I	Converts decimal value to octal equivalent.
70 D=Q	
80 I=I*10	
90 IF D#0 THEN 50	Works for both positive and negative integers.
•100 FNO=N8	Assigns function name a value.
•110 FN END	Function end.

The dots by statements 10, 100, and 110 indicate the essential parts of a multiple-line function.

Again, the program segment above only *defines* the function. In order to *evaluate* the function, you must reference it in another part of the same program, replacing the parameter D with the desired expression.

For instance, add the following statements to the program segment above.

```
  1 PRINT "DECIMAL","OCTAL"
  2 FOR J=128 TO 256
  3 PRINT J,FNO(J)
  4 NEXT J
120 END
```

RUN

```
DECIMAL                   OCTAL
 128                       200
 129                       201
 130                       202
 131                       203
 132                       204
 133                       205
 134                       206
 135                       207
 136                       210
 137                       211
 138                       212
 139                       213
 140                       214
 141                       215
 142                       216
 143                       217
 144                       220
```

STEP PAUSE

Notice that we used the variable J as our loop counter in program statements 2 through 4. What if we had used the variable I in both our main program and in the function definition?

```
  1 PRINT "DECIMAL","OCTAL"
  2 FOR I=128 TO 256
  3 PRINT I,FNO(I)
  4 NEXT I
 10 DEF FNO(D)
 20 D=IP(D)
 30 N8=0
 40 I=1
 50 Q=IP(D/8)
 60 N8=N8+(D-Q*8)*I
 70 D=Q
 80 I=I*10
 90 IF D#0 THEN 50
100 FNO=N8
110 FN END
120 END
```

This program would only generate the first value of FNO(I) because the value of I is changed in the function definition.

Note that variable D in the main program would not be similarly affected. For instance, if all of the I variables were changed to D's in statements 1 through 4, the program would work.

The program would not compute all of the values assigned by the loop counter.

```
DECIMAL                   OCTAL
 128                       200
```

The point of our discussion, here, is that if the value of a variable is changed in a function definition, then it is also changed in the main part of the program.

Let's look at two examples of multiple-line functions using string variables.

Example: Write a program that formats a number with a comma in place of the decimal point. If the number is an integer, supply two zeros to the right of the comma. Consider only numbers with absolute values that are greater than 1×10^{-11} and less than 1×10^{11}.

```
  10 REM *REPLACE POINT WITH COMM
     A
•  20 DEF FNE$(N)
  30 IF FP(N)=0 THEN F$="000" ELS
     E F$=VAL$(ABS(FP(N)))
  40 I$=VAL$(IP(N))
•  50 FNE$=I$&","&F$[2]
•  60 IF ABS(N)<.00000000001 OR AB
     S(N)>100000000000 THEN FNE$=
     "OUT OF RANGE"
•  70 FN END
  80 INPUT D
  90 DISP D,FNE$(D)
 100 GOTO 80
 110 END
```

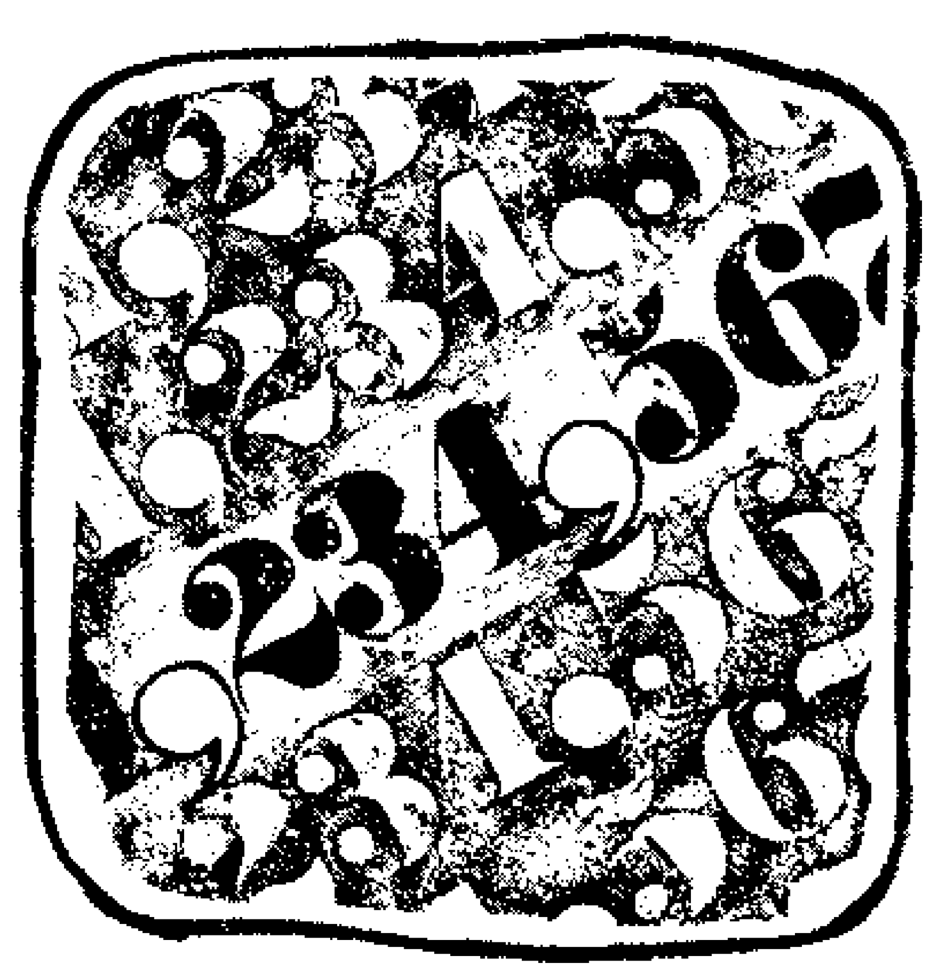
Function definition.
Checks for out-of-range numbers.

RUN

```
?
1234.1234
 1234.1234                1234,1234
?
56.85
 56.85                    56,85
?
14
 14                       14,00
?
689234.156
 689234.156               689234,156
```

PAUSE

Example: Now write a program that formats a number with commas every three digits to the left of the decimal point, using a multiple-line string function to insert the commas. Consider only numbers with absolute values less than or equal to 1×10^{11} and greater than 1×10^{-11}.



```
   10 REM *INSERT COMMAS
 • 20 DEF FNC$(N)
   30 B=0 @ N$="" @ M=1000000000 @
        N1=N
   40 IF ABS(N)<=100000000000 AND
      ABS(N)>.00000000001 THEN 70
 • 50 FNC$="OUT OF RANGE"
   60 GOTO 160
   70 FOR I=1 TO 3
   80 IF N1<M THEN 120
   90 B=IP(N1/M)
  100 N1=FP(N1/M)*M
  110 N$=N$&VAL$(B)&","
  120 M=M/1000
  130 NEXT I
  140 N$=N$&VAL$(N1)
 •150 FNC$=N$
 •160 FN END
  170 INPUT X
  180 DISP X,FNC$(X)
  190 GOTO 170
  200 END
```

Function definition.

```
RUN
?
1234
 1234                    1,234
?
1234567
 1234567                 1,234,567
?
12345678912
 12345678912
12,345,678,912
?
1234.567
 1234.567                1,234.567
?
123456789.12
 123456789.12
123,456,789.12
?
```

PAUSE

Multiple-line functions are not recursive. For example, the following attempt to define a factorial function would generate an error message.

```
10 DEF FNF(X)
20 IF X=0 THEN FNF=1 ELSE FNF=X
   *FNF(X-1)
30 FN END
40 INPUT T
50 PRINT T;FNF(T)
60 GOTO 40

RUN
?
3
Error 42 on line 20 : RECURSIVE
FN CALL
```

The error occurs in attempting to use the function name in the function definition.

As we mentioned earlier, the length of a string argument passed between a function and the main program defaults to 18 characters. You can specify a larger string in the function definition by enclosing the length within brackets following the string argument. For instance:

`DEF FNS$(A$[75])`	Allocates a string argument of 75 characters.

You cannot use the `DIM` statement to dimension the string argument since it is considered a "dummy" variable. Therefore, you must allocate space for the argument within the `DEF FN` statement itself. When you do this, the system considers the entire `DEF FN` statement, including the allocated variable, as part of the program line. Thus, the maximum length of the string argument in a multiple-line function is approximately 240 characters and the maximum length of the string argument in a single line function is dependent on the complexity of the expression that defines the function. If the argument is too large, the system will display `Error 85 : EXPR TOO BIG`. If this happens, decrease the length of the string argument until the system accepts the statement.

Note that although a string function may accept a string argument larger than the default length, the resulting string passed from the function to the main program can be no longer than 18 characters.

Refer to problems 9.4 through 9.6 at the end of this section for more examples of multiple-line functions.

Subroutines

Often, the same sequence of statements is executed more than once within a program. By using a subroutine you can key in the group of statements only once and then access the statements from different places within the program. If you group all of the often-used routines at the end of your program, you can make the program easier to follow and understand.

Subroutines are similar to functions in that they can be referenced from other parts of the program. But a subroutine is not given a name; it is referenced by a `GOSUB` statement and the beginning statement number of the routine.

`GOSUB` statement number

The `GOSUB` statement transfers program control to the subroutine you wish to execute. The statement number must be that of the first statement of the subroutine.

A subroutine can begin with any statement except `NEXT`. For example, the subroutine might begin with `REM`, `LET`, `IF... THEN`, `FOR`, etc. The last statement of a subroutine must be a `RETURN` statement.

`RETURN`

There may be more than one `RETURN` statement within a subroutine. As soon as a `RETURN` statement is encountered, program control is transferred to the statement following the particular `GOSUB` that referenced the routine.

Arguments or parameters are not used to pass values from the subroutine to the main program. As with functions, all variables used in subroutines are global variables, in other words, all program variables are accessible in both functions and subroutines. If the value of the variable is changed within a subroutine, it is also changed in the main program. Unlike user-defined functions, subroutines cannot appear in a numeric or string expression. And, character strings "passed" between a subroutine and the main program can be as long as the length dimensioned in a `DIM` statement.

For example:

```
   10 DISP "ENTER NUMBER"
   20 INPUT N
   30 IF N<0 THEN 10
 • 40 GOSUB 100
  .
  .
  .
   90 GOTO 160
  100 REM *SUM FROM 1 TO N
  110 S=(N*(N+1))/2
  120 PRINT "SUM=";S
  .
  .
  .
  150 RETURN
  160 N=N*2
 •170 GOSUB 100
  .
  .
  .
  300 END
```

Subroutine.

When the program executes statement 40, program control is immediately transferred to statement 100. When a RETURN statement is encountered, control is transferred to the line following 40. Statement 170 also transfers control to statement 100. In this case, RETURN transfers program control to the line following 170.

Subroutines may be nested, that is, a second subroutine can be entered before the RETURN statement of the first is executed.

For example:

```
    10 DISP "ENTER NUMBER"
    20 INPUT N
    30 IF N<0 THEN 10
 •  40 GOSUB 1000
   .
   .
   .
    90 STOP
  1000 REM *SUM FROM 1 TO N
  1010 S=(N*(N+1))/2
  1020 PRINT "SUM =";S
  1030 DISP "SUM OF SQUARES (Y/N)"
  1040 INPUT A$
 •1050 IF A$ = "Y" THEN GOSUB 2000
   .
   .
   .
  1200 RETURN
  2000 REM *SUM SQUARES OF INTEGER
       S FROM 1 TO N
  2010 S2=(N*(N+1)*(2*N+1))/6
  2020 PRINT "SUM OF SQUARES=";S2
   .
   .
   .
  2090 RETURN
```

Nested subroutine.

The subroutine at line 2000 is nested within the one at line 1000. The RETURN statement on line 2090 returns to the line following 1050 in the first subroutine. The RETURN statement at 1200 returns to the line following statement 40.

Subroutines can be nested as deeply as available memory allows (up to 255 levels of nesting). When a RETURN is executed, control returns to the subroutine that was entered most recently.

See problem 9.7 to write a complete program that uses subroutines.

The Computed GOSUB Statement

The ON... GOSUB *(computed GOSUB)* statement enables you to access any of one or more subroutines based on the value of a numeric expression. It operates exactly like an ON... GOTO statement except that instead of transferring program control to one statement, ON... GOSUB transfers control to the first statement of a subroutine. The RETURN statement of the subroutine returns program execution to the statement following the ON... GOSUB statement that referenced it. The ON... GOSUB statement is programmable only; it can't be executed from the keyboard.

ON numeric expression GOSUB statement number list

The numeric expression is evaluated and *rounded* to an integer. A value of 1 causes the subroutine at the first statement number in the list to be accessed; a value of 2 causes the subroutine at the second statement number in the list to be accessed, and so on.

All RETURN statements in the subroutines accessed transfer program control back to the end of the statement number list of the ON... GOSUB statement.

For example:

```
  10 FOR X=1 TO 3
• 20 ON X GOSUB 200,300,400  <----  This statement means:
                                     If X=1, then GOSUB 200.
  30 NEXT X                          If X=2, then GOSUB 300.
                                     If X=3, then GOSUB 400.
  .                                  Program control reaches statement 40
  .                                  when the FOR-NEXT loop is completed.
  .                                  RETURN in each subroutine transfers
 100 STOP                            control to statement 30.
 200 PRINT X;SIN (X)       }  <----  Subroutine 200.
 290 RETURN
 300 PRINT X;X^2;COS(X)    }  <----  Subroutine 300.
 390 RETURN
 400 PRINT X;X^3;TAN(X)    }  <----  Subroutine 400.
 490 RETURN
```

If the value of the numeric expression is less than one or greater than the number of statement numbers in the list, an error occurs.

Problem 9.8 provides another example of the use of the ON... GOSUB statement.

Branching Using Special Function Keys

You have seen some of the many uses of the special function keys from running the programs in the Standard Pac. The eight special function keys, (k1) through (k4) (unshifted), and (k5) through (k8) (shifted), can be used to interrupt a running program and cause branching.

This interrupt capability is declared with an `ON KEY#` statement. The `ON KEY#` statement specifies the branching operation that will occur when the related key is pressed.

`ON KEY#` key number [, key label] `GOTO` statement number
`ON KEY#` key number [, key label] `GOSUB` statement number

The key number must be an integer from 1 through 8. The "key label" is a string expression which is truncated to the first eight characters. When a user-defined key is pressed during a program run, and an `ON KEY#` statement has been declared for it, the specified branching occurs. With `ON KEY# . . . GOSUB`, *program control returns to the next executable statement.*

If a program is not running, pressing a user-defined key does nothing.

KEY LABEL

The `KEY LABEL` statement is used to recall key labels for the user-defined keys to the display. The statement is simply:

`KEY LABEL`

As you can see from the `ON KEY#` statement syntax, you can optionally specify a key label in the program definition of a key. Once defined and labeled in a program, the `KEY LABEL` statement causes the labels to appear on the lower three lines of the display.

All eight user-defined keys can have labels defined and displayed: each one appears in a unique location on the display, situated directly above the corresponding special function keys on the keyboard.

The (KEY LABEL) key recalls all current labels, at any time, and displays them on the bottom three lines of the display. It performs the same operation that the `KEY LABEL` statement does in a program.

Both the (KEY LABEL) and the `KEY LABEL` statement also move the cursor to the home position on the display. Thus a full 13 lines may be entered or displayed before the key labels are over-written.

_ (Cursor position after `KEY LABEL`)

This is a sample display with seven of the eight keys labeled, immediately after `KEY LABEL` has been executed.

```
--------------------------------------------------------
ADD       STORE               EXIT
INIT-A    INPUT     COPY-A    CHANGE
```

Perhaps the following short program can best illustrate the ease with which function keys can be defined and the rapidity with which they are executed when pressed while the program is running.

```
10 ON KEY# 1,"MID C" GOSUB 100
20 ON KEY# 2,"D" GOSUB 200
30 ON KEY# 3,"E" GOSUB 300
40 ON KEY# 4,"F" GOSUB 400
50 ON KEY# 5,"G" GOSUB 500
60 ON KEY# 6,"A" GOSUB 600
70 ON KEY# 7,"B" GOSUB 700
80 ON KEY# 8,"C" GOSUB 800
90 CLEAR @ KEY LABEL
96 DISP "KEY OF C MAJOR"
97 DISP "PLAY MELODIES BY PRESS
   ING THE USER-DEFINED KEYS"
98 GOTO 98
100 BEEP 201,100
110 RETURN
200 BEEP 178,100
210 RETURN
300 BEEP 157,100
310 RETURN
400 BEEP 147,100
410 RETURN
500 BEEP 130,100
510 RETURN
600 BEEP 114,100
610 RETURN
700 BEEP 101,100
710 RETURN
800 BEEP 94,100
810 RETURN
1000 END
```

As soon as you press (RUN), the display is cleared and the key labels are recalled:

```
KEY OF C MAJOR
PLAY MELODIES BY PRESSING THE US
ER-DEFINED KEYS
_









--------------------------------
G        A        B        C
MID C    D        E        F
```

Play a few tunes with the special function keys. The program quickly illustrates that each press of a special function key causes *one* execution of the GOTO/GOSUB as defined by the ON KEY # statement, and that one key interrupts another. When a defined key is pressed during a running program, the current program line is completed before the specified branching occurs.

Notice statement 98 in the Key of C program:

```
98 GOTO 98
```

Since ON KEY# statements are only active when a program is running, it is often necessary to have a place in the program that does nothing but idle, waiting for a keystroke. We cannot use a STOP or END statement to separate the key definitions from the subroutines; program execution would halt as soon as either statement was encountered. Thus, a GOTO statement that "goes to" itself keeps the ON KEY# declaratives active in a particular part of a program.

ON KEY# declaratives are temporarily deactivated while a program is waiting for a response to an INPUT statement. Pressing them on input will cause their related keycodes to appear on the input line. Key definitions are also deactivated after PAUSE is executed. They resume functioning with RUN or CONT. If another program is "chained" to the program with the ON KEY# statements, the key definitions will no longer be active. (Refer to the CHAIN command in section 11.)

Cancelling Key Assignments

The ON KEY# declarative holds for a key until another declarative for the same key, (SCRATCH), or OFF KEY# is executed.

OFF KEY# key number

The OFF KEY# statement cancels the definition and branching operation of the specified key.

Problem 9.9 provides another example of ON KEY# statements. Refer to any of the Standard Pac programs for more examples.

The Timers

Along with the SETTIME statement and the time functions, the HP-85 provides three individual timers that may be set to interrupt a program at the specified time interval and cause the specified branching to occur. Interrupt intervals for the timers are declared with ON TIMER# statements. The ON TIMER# statement must be declared within a program.

ON TIMER# timer number , milliseconds GOTO statement number
ON TIMER# timer number , milliseconds GOSUB statement number

The timer number must be either 1, 2, or 3. The number of milliseconds must be a value less than |99999999| and greater than |.5|. The sign of the milliseconds parameter is ignored. Zero and numbers outside the given range interrupt immediately and then wait 99999999 milliseconds before the next interrupt.

When the interrupt occurs, the specified branching occurs within a program.

For example, timer #1 interrupts the program every 15 minutes to go to statement 50 in the following program (as long as the program is running).

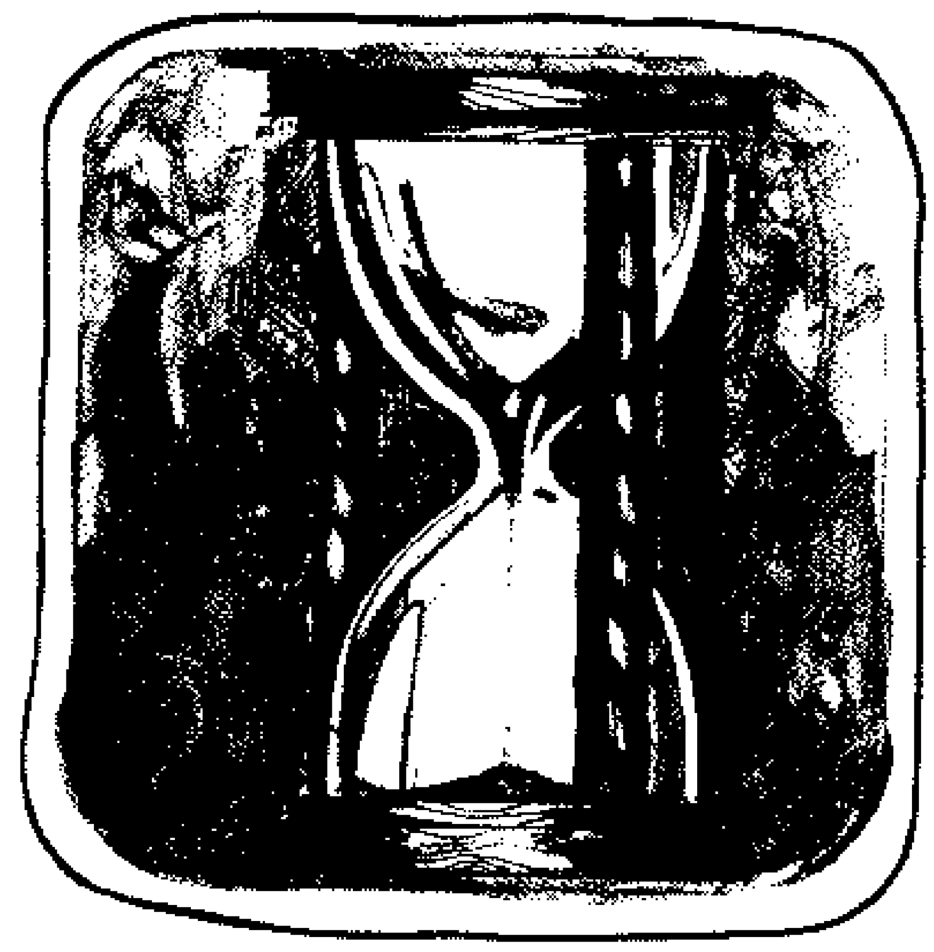
```
10 ON TIMER#1,900000 GOTO 50
:
:
50 BEEP 157,50 @ BEEP 201,50
60 BEEP 178,50 @ BEEP 272,75
70 WAIT 100
80 BEEP 272,50 @ BEEP 178,50
90 BEEP 157,50 @ BEEP 201,100
```

The timers continue to interrupt the system after a program is halted, but the interrupt does not cause the specified branching. The timers are deactivated when you edit the program, when SCRATCH or RESET is pressed, or when the `OFF TIMER#` statement is declared.

`OFF TIMER#` timer number

The `OFF TIMER#` statement deactivates the corresponding `ON TIMER#` statement. No further interrupts will occur from the specified timer until it is reactivated.

Example: Suppose you have written a lengthy program which is actually composed of five separate tests. Set up timers to wait for an input response from the user. If there is no response within 20 seconds, go to the next segment of the program.



```
  10 PRINT "SECTION 1.1"
  20 PRINT
  30 DISP "IF THIS TEST IS TO BE
     RUN, TYPE YES"
• 40 ON TIMER# 1, 20000 GOTO 800
  50 INPUT Z9$
• 60 OFF TIMER# 1
  70 IF Z9$#"YES" THEN 810
  80 PRINT
  90 PRINT "BEGIN TEST NOW"
 100 PRINT
 .
 .
 .
•800 OFF TIMER# 1
 810 PRINT "SECTION 1.2"
 820 PRINT
 830 DISP "IF THIS TEST IS TO BE
     RUN, TYPE YES"
•840 ON TIMER# 1, 20000 GOTO 1600
 850 INPUT Y8$
•860 OFF TIMER# 1
 870 IF Y8$#"YES" THEN 1610
 880 PRINT
 890 PRINT "BEGIN TEST NOW"
 .
 .
 .
```

If the user types "YES" within 20 seconds, the test will be executed. If there is no response to line 50 within 20 seconds, the program branches to statement 800.

Reset timer for second test.

And so on.

The fact that timer's continue to interrupt even after the program is halted is important. Errors may occur if the timers are interrupting so fast that the system (program) cannot get anything done. Try this:

```
 10 ON TIMER# 1, 1 GOSUB 100
 20 ON TIMER# 2, 1 GOSUB 100
 30 ON TIMER# 3, 1 GOSUB 100
 40 STOP
100 DISP "SUB"
110 RETURN
120 END
```

First press (RUN), then press (LIST). When you press (RUN) the first timer tries to go to statement 100 but gets interrupted by the second timer and the second timer gets interrupted by the third, etc. Thus statement 100 may never be executed or the system will give you an error message. You'll find that the system will list the program very slowly since it is being interrupted continually. Execute (SCRATCH), (RESET), or the `OFF TIMER#` statement to halt the timers.

Refer to the Standard Pac for more examples using the timers, especially the Timer Program.

Problems

9.1.a. Define a single-line function that rounds a number at the decimal point. Evaluate the function from –5 to 5 in intervals of 0.3.

b. Define another single-line function that rounds a given number to the thousandths decimal place. Evaluate this function from 1 to 10 in intervals of 0.5.

9.2.a. Define a single-line function to compute the area of a circle given the radius of the circle according to the formula $A = \pi r^2$. Evaluate this function for integer values of 350 to 360.

b. Use a rounding function to display the areas of the circles with the above radii, rounded to the second digit past the decimal place.

9.3 Define a function that computes the length of the hypotenuse of a right triangle given the lengths of the two sides. Evaluate the function with one side equal to 5 while the other has values of 4,3,6,7, and 9.

9.4.a. Define a multiple-line function that converts a number with an octal base to its decimal equivalent. Test your program with the values obtained from the opposite conversion in the program on page 148.

b. What if, in the octal to decimal conversion, the original number has an illegal digit, i.e., a digit greater than or equal to 8? How would you check for an illegal digit and what value would you return for the function?

9.5 Define a multiple-line function to compute the factorial of a non-negative number. Use the function to compute the number of ways that eight books can be arranged on one shelf.

(Method: $P_8^8 = 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$)

What if, instead of eight different books, you have only four different books for each of which there are two copies? Determine the number of distinguishable arrangements on one shelf.

The number of arrangements $= \dfrac{8!}{(2!)^4}$

9.6 Define a multiple-line function to round a numeric value to the hundredths place, and add either a `$` or a `£` before the number. If the fractional part of the number is zero, fill the fractional part of the final number with zeros.

9.7. Write a program that will make it easy for you to manipulate tables of data. First dimension and initialize the elements of an array, then input values for the array elements. Include in your program three subroutines to accomplish the following tasks. Use the subroutines to print or display the sum of the rows and columns of your data table:

1. Write one subroutine to display or print the array.
2. Write a second subroutine that enables you to change a particular array element.
3. Write a third subroutine that finds the sum of each row, the sum of each column and the total sum.

Test your program by finding the row sums, column sums, and total sum of the data in the following table.

12.59	13.69	14.78	?
11.43	22.56	43.78	?
13.52	12.78	14.98	?
?	?	?	

Before you find the row sums, column sums, and total sum of the data in the table, change the value in row 1, column 3 (14.78) to 14.67.

9.8. On his frequent transatlantic missions, chief detective Sylvester S. Py must send encoded messages to the home office. Prior to each mission he supplies the home office with his encoding number. They, in turn, give Sylvester the number they'll use to encode messages sent to him.

Write a BASIC program that uses two subroutines, one to encode messages, the other to decode messages. Use a computed `GOSUB` statement to determine which subroutine is to be accessed. Let the code number be a seed for a sequence of random numbers that encodes the message. (This enables you to use the same random number seed to decode the message.) Use only capital letters in input, coding, or decoding operations. Allow the user to enter one word at a time; you supply the spaces between words.

Suppose Sylvester wants to send the following message to the home office, using his code number (random number seed) .123.

"GET ME TO THE BANK ON TIME"

Run the program to find the encoded message. Run it again, using the same code number to decode the message. Then decode the following message, recently received by Sylvester, using the home office code number .3579.

"NNLSNUNVS IGPXR RQP BVE"

Hint: Use an encoding function like `C$=C$&CHR$(65+(NUM(I$[I,I])+INT(26*RND))MOD 26)` for the length of each word, `I$`.

9.9. Write a "standard pac" program by modifying the row sum and column sum program you wrote for problem 9.7 (sample solution in appendix F) so that the subroutines are performed at the touch of a special function key. Define the special function keys as follows:

```
ON KEY #1, "INIT" GOSUB      ———→ Initialize array elements.
ON KEY #2, "INPUT" GOSUB     ———→ Input values into array.
ON KEY #3, "COPY-A" GOSUB    ———→ Display or print current array.
ON KEY #4, "CHANGE" GOSUB    ———→ Change a particular array element.
ON KEY #5, "SUM" GOSUB       ———→ Sum the rows, columns, and find total sum
                                  of array.
```

We'll leave (k6) through (k8) for you to define. Additional subroutines might be "ADD," add a row or column to the array; "HELP," display the key definitions; "DELETE," delete a row or column from the array; or "AVG," find the average of the values in a particular row or column.

Run the program to sum the rows and columns of some tables of your own.

Printer and Display Formatting

You have seen that the use of commas, semicolons, and quoted text provide limited control of the format of printed or displayed information. Three statements, PRINT USING, DISP USING, and IMAGE, provide the capability of generating printed or displayed output with complete control of the format. The syntax of the statements have two different forms. First, we'll discuss PRINT USING and DISP USING with IMAGE. Later, we'll show that you can specify the format and the information to be formatted in the same statement. Other topics included in this section are:

- Using the TAB function.
- Redefining the printer and the display with the PRINTER IS and CRT IS statements.

Using IMAGE

The IMAGE statement specifies the format by which numbers and strings in the PRINT USING or DISP USING statements will be printed.

PRINT USING statement number [; print using list]
DISP USING statement number [; disp using list]
IMAGE format string

The statement number must refer to an IMAGE statement. The print and disp using lists may be comprised of simple and subscripted variable names, numeric expressions, and string expressions. Functions (including user-defined functions) may be included in the print or disp using list, but if a multiple-line function contains PRINT or DISP statements it may distort the output format. The items in the list are separated by commas or semicolons. However, the commas and semicolons do not affect the format as they do in the PRINT or DISP statements; they merely separate the items in the list. The output is totally controlled by the format string of the IMAGE statement. The format string is a list of field specifiers separated by delimiters. Each field specifier is comprised of special symbols that determine the format of a single item in the print or disp using list. The symbols specify the number of digits, the placement of a comma, decimal point, or blanks—virtually anything having to do with numeric and string output and carriage control.

Each item in the print or display list must correspond to an appropriate numeric or string field specifier.

Delimiters

Two delimiters are used to separate field specifiers:

, A comma is used only to separate two specifiers.

/ A slash can also be used to separate two specifiers, but its main function is to perform a carriage return and line feed (CR-LF).

The slash, /, can be used as a field specifier by itself; that is, it can be separated from other specifiers by a comma. But only the slash delimiter, /, can be directly replicated (see page 166).

```
450 PRINT USING 460
460 IMAGE "COST",3/,"DISCOUNT"
```

3/ is equivalent to ///.

```
COST



DISCOUNT
```

Prints "COST" and performs 1st CR-LF.
Performs 2nd CR-LF.
Performs 3rd CR-LF.
Prints "DISCOUNT."

The symbols 3/ indicate that three carriage returns and line feeds are to be performed between printing COST and DISCOUNT. Thus, two blank lines are output.

However, the following image statement would output three blank lines before printing COST.

```
460 IMAGE 3/,"COST"



COST
```

Performs 1st CR-LF.
Performs 2nd CR-LF.
Performs 3rd CR-LF.
Prints "COST."

If *n* / is at the beginning of an image format string, *n* blank lines are output.
If *n* / follows a field specifier in an image format string, *n*–1 blank lines are output.

Blank Spaces

X Specifies a blank space.

A number preceding X specifies the number of blanks; for instance, 4X means four blanks. (XXXX also specifies four blanks.)

String Specification

Text can be specified in two ways:

" " Text enclosed within quotation marks is printed or displayed exactly as it is quoted. You may specify quoted literals (strings) in either the print or display list or in the IMAGE statement.

For example:

```
40 IMAGE "**",4X,"Results",4X,"
  **"
50 PRINT USING 40
**    Results    **
```

A **Specifies a single string character. A number preceding A specifies the number of characters. The length of a string specifier is determined by the number of As that are specified between delimiters; this corresponds to one item in the print/disp using list. When using the A string specifier, all text is left-justified.**

The above example could also have been written:

```
 90 A$="Results"
100 IMAGE "**",4X,7A,4X,"**"
110 PRINT USING 100 ; A$
**    Results    **
```

7A specifies a field comprised of seven characters. 4X specifies a field comprised of four blanks.

Or like this:

```
130 A$="Results"
140 IMAGE AA,4X,7A,4X,AA
150 PRINT USING 140 ; "**",A$,"*
*"
```

AA can also be represented as 2A.

If the string item in the print/disp using list is longer then the number of characters specified, the string is truncated. For example:

```
180 PRINT USING 190; "RESIDENCE"
190 IMAGE 6A
RESIDE
```

If the item is shorter, the rest of the field is filled with blanks.

Numeric Specification

A variety of symbols can be used to specify numbers: digit symbols, sign symbols, radix symbols, separator symbols, and an exponent symbol.

Digit Symbols

D Specifies a digit position. A number preceding D specifies the number of digit positions. If the number of Ds to the left of the decimal point or radix specify a field larger than the numeric item, then the item is right-justified in the field and leading zeros are replaced with spaces. If the number of Ds to the right of the decimal point or radix specify a field larger than the numeric item, then the item is left-justified in the field with trailing zeros. If the fractional part of the numeric item is larger than the number of Ds to the right of the decimal point or radix, then the item is rounded to fit the specified field. D is the only digit symbol that can be used to specify digits to the right of a decimal point or radix. For example:

```
210 PRINT USING 280 ; 250,25.50
280 IMAGE 5D,2X,DD.DD
  250  25.50
```

Z Specifies a digit position—leading zeros are replaced with zeros as a fill character. You cannot use a Z to the right of a radix symbol. Again, a number preceding Z specifies the number of digit positions. For example:

```
290 PRINT USING 310 ; 256,321
310 IMAGE 5Z,2X,ZZZZZ
00256  00321
```

* An asterisk also specifies a digit position, but leading zeros are replaced with asterisks as a fill character. You cannot use an * to the right of a radix symbol. A number preceding * specifies the number of asterisks. For example:

```
340 IMAGE 5*,2X,5Z,2X,5D
350 PRINT USING 340 ; 99,77,55
***99  00077     55
```

As you can see, any digit symbol, *, Z, or D, can be used to specify the integer portion of any number. But, you cannot mix the symbols in the manner shown below, in the first IMAGE statement. For instance, if D is used to specify a digit position of a number, all of the number must be specified with D's, except that the digit symbol specifying the one's place can be a Z regardless of the other symbols. For example:

```
360 PRINT USING 370 ; 357,972
370 IMAGE DDZZ,2X,D*ZZZ
```

The IMAGE statement contains an invalid image and would cause an Error message to appear. However, the following image is valid:

```
370 IMAGE DDDZ,2X,*****Z
 357   ***972
```

Radix Symbols

A radix indicator is the symbol that separates the integer part of a number from the fractional part. In the United States, this is customarily the decimal point, as in 34.7. In Europe, this is frequently the comma as in 34,7. One radix symbol at most can appear in a numeric specifier. Only the symbol D can be used to specify a digit to the right of the radix indicator.

. Specifies a decimal point in that position.

R Specifies a comma radix indicator in that position.

Here are some examples:

```
440 PRINT USING 450 ; 473.1,25.3
    92,76.5
450 IMAGE DDD.DD,2X,**Z.DDD,2X,Z
    ZZRDD
473.10  *25.392  076,50

490 IMAGE DDZ.DDD,4X,3Z.3D,4X,Z.
    DD
500 PRINT USING 490 ; .756,99.99
    ,.879
  0.756     099.990     0.88
```

Note that .879 is rounded to 0.88 since the image specified only two digits to the right of the radix.

Sign Symbols

Two sign symbols control the output of the sign characters + and −. Only one sign symbol at most can appear in a numeric specifier. When no sign symbol is specified, any minus sign occupies a digit position.

S Specifies output of a sign: + if the number is positive, − if the number is negative.

M Specifies output of a sign: − if the number is negative, a blank if it is positive.

For example:

```
502 PRINT USING 504 ; -47.2,-.51
    ,33.5,38.12
504 IMAGE MDD.DD,2X,SZZ.DD,2X,SZ
    Z.DD,2X,MZZ.DD
-47.20  -00.51  +33.50  38.12
```

The sign "floats" with the number; for example:

```
506 PRINT USING 508 ; -5,6,-.07
508 IMAGE SDDD.D,S3D.D,M2D.DD
  -5.0  +6.0  -.07
```

In the examples above, the sign appears immediately to the left of the number. If you use a Z or * symbol in your format, the minus sign will appear to the left of any leading zeros or asterisks.

Digit Separator Symbols

Digit separators are used to break large numbers into groups of digits (generally three digits per group) for greater readability. In the United States the comma is customarily used; in Europe, the period is commonly used.

C Specifies a comma as a separator in the specified position.

P Specifies a period as a separator in the specified position.

The digit separator symbol is output only if a digit in that item has already been output; the separator must appear between two digits. When leading zeros are generated by the Z symbol, they are considered digits and will contain separators. An IMAGE format consisting of leading asterisks may contain separators. But if numbers are not output on both sides of the separator, the separator will be replaced with an asterisk.

```
512 PRINT USING 515 ; 25613.92,2
    7.96,71.5
515 IMAGE 3DC3D.DD,2X,ZC3Z.DD,2X
    ,3DC3D.DD
 25,613.92  0,027.96      71.50

517 DISP USING 520 ; 99,9999.99
520 IMAGE DDD,2X,DCDDD.DD
 99  9,999.99
```

Exponent Symbol

E Specifies that the numeric field that contains E is to be output in scientific notation. E causes the output of an E, the sign of the exponent (+ or −), and a three-digit exponent. At least one digit symbol must precede the E symbol.

For example:

```
530 DISP USING 533 ; 157.24
533 IMAGE D.DDDE
1.572E+002

535 PRINT USING 538 ; 5.762
538 IMAGE DDD.DDE
576.20E-002
```

Compacted Field Specifier

A single symbol, K, is used to define an entire field for either a number or a string of characters. If the corresponding print/disp using item is a string, the entire string is output. If it is a numeric, it is output in standard number format (see page 47), except that K outputs no leading or trailing blanks. For example:

```
80 PRINT USING 90 ; "ABC",415,"
   DEF",.01
90 IMAGE K,2X,K,K,K
ABC  415DEF.01
```

Replication

Many of the symbols used to make up image specifiers can be repeated to specify multiple symbols by placing an integer in the range 1 through 9999 in front of the symbol. You have already seen some examples; the following IMAGE statements, for instance, all specify the same image:

```
540 IMAGE DDD.DD
545 IMAGE D2D.2D
550 IMAGE 3D.DD
555 IMAGE 3D.2D
```

These symbols can be replicated: D, Z, X, *, ( ), A, and /.

In addition to symbol replication an entire specifier or group of specifiers can be replicated by enclosing it in parentheses and placing an integer in the range 1 through 9999 before the parentheses. For example:

```
40 IMAGE DD.D,6(DDD.DD)
50 IMAGE 4Z.D,4(2X,7*Z.D,2(2X,D
   ))
```

So, specifying 3(DD) is the same as specifying DD,DD,DD.

In this manner, K can be repeated:

```
60 IMAGE 4(K)
```

Same as specifying K,K,K,K.

Up to 128 levels of nested parentheses can be used for replication.

Reusing the IMAGE Format String

A format string is reused from the beginning if it is exhausted before the print using list. For example:

```
150 PRINT USING 155 ; 25.71,99.9
    ,14.23
155 IMAGE DDD.DD
 25.71 99.90 14.23
```

Field Overflow

If a numeric item requires more digits than the field specifier provides, an overflow condition occurs. When this happens, a warning message is displayed and the program continues. For example:

```
160 PRINT USING 165 ; 25.9,336.7
    1,12.1,-14.3
165 IMAGE 4(DD.DD)
Warning 2 on line 160 : OVERFLOW
```

Both numbers 336.71 and -14.3, with an image of `DD.DD`, create an overflow condition. Remember that a minus sign not explicitly specified with `S` or `M` requires a digit position.

Formatting in PRINT/DISP USING Statements

There is another form that a `PRINT USING` or `DISP USING` statement may have, which enables you to specify the image string and the print/disp using list in the same statement:

`PRINT USING` image format string [`;` print using list]
`DISP USING` image format string [`;` disp using list]

Instead of specifying the `IMAGE` statement number, you can include the image format string, enclosed within quotation marks in the `PRINT/DISP USING` statements before you specify the print/disp using list. The image format string may be a string enclosed within quotation marks, a string variable, or any string expression that specifies the format.

Examples:

```
 10 PRINT USING "4D.DD,2X,**Z.DD
    D"; 1473,25.39
1473.00  *25.390

 20 PRINT USING "3D.2D" ; 310.12
    ,56,42.5
310.12 56.00 42.50

 30 DIM F$[19]
 40 F$="3DC3D.2D,2X,2C3Z.2D"
 50 DISP USING F$ ; 25613.92,27.
    96
 25,613.92  0,027.96
```

Remember to dimension the string if it is longer than 18 characters.

You *cannot* use quotation marks to specify literal text within an image format string in a `PRINT/DISP USING` statement since quotation marks are used to define the string.

For instance, the following is not allowed and would cause an `Error 84 : EXCESS CHARS` message to appear if you try to enter the statement:

```
50 PRINT USING "NAME",2X,10A,
   "AGE",3D" ; "CHARLES",43
```

The statement is not recognized after the second quotation mark.

An image format string for statement 50 could be specified in either of these ways:

```
50 PRINT USING "4A,2X,10A,3A,3D
   "; "NAME","CHARLES","AGE",43
NAME   CHARLES    AGE 43
```

Or:

```
50 PRINT USING 60 ; "CHARLES",43
60 IMAGE "NAME",2X,10A,"AGE",3D
NAME   CHARLES    AGE 43
```

You can use quoted literals in an IMAGE statement since the quotation marks do not define the complete image format string as they do in the PRINT/DISP USING statement.

Here is a summary table of image symbols and their uses:

Image Symbol	Symbol Replication	Purpose	Comments
X	Yes	Blank	Can go anywhere
" "	No	Text	Can go anywhere
D	Yes	Digit	Fill=blanks
Z	Yes	Digit	Fill=zeros
*	Yes	Digit	Fill=asterisks
S	No	Sign	"+" or "−"
M	No	Sign	blank or "−"
E	No	Scientific notation	Format=ESDDD
.	No	Radix	Output "."
C	No	Comma	Conditional number separator
R	No	Radix	Output ","
P	No	Decimal point	Conditional number separator
A	Yes	Characters	Strings
( )	Yes	Replicate	For specifiers, not symbols
K	No	Compact	Strings or numerics
,	No	Delimiter	
/	Yes	Delimiter	Output CR-LF

The main factor that must be taken into account with formatted output is the display or printer width. Especially when dealing with numeric output, formatting should be designed so that a line of characters does not exceed the number of characters per line (32 characters per line on the HP-85 printer or display).

The TAB Function

The TAB function is used with the PRINT and DISP statements to print or display information at specified character positions. The main consideration with TAB is the length of a line on the printer or display.

TAB (character position)

The character position may be a number as large as 32767, but you really have 1 through 32 characters positions on either the display or the printer. When the character position specified is greater than the number of columns, it is reduced MOD32.

Example: The following program prints the heading for the variables *X*, *Y*, and *Z*.

```
10 INPUT X,Y,Z
20 PRINT "AVERAGE";TAB(15);"MEA
   N";TAB(26);"MEDIAN"
30 PRINT X;TAB(15);Y;TAB(26);Z
40 END
```

The first heading, AVERAGE, starts at character position 0; the heading, MEAN, starts at character position 15; and the heading, MEDIAN, starts at character position 26. Then in statement 30, the variables are printed under the three headings. If your X, Y, and Z values were input as 11.23, 11, and 11.4, respectively, the printout would be:

```
AVERAGE        MEAN       MEDIAN
 11.23          11         11.4
```

Remember that a comma in a printer or display list outputs the next item in the next print or display zone. Thus, all print or display items used with TAB must be separated by semicolons. The TAB function cannot be used with the PRINT USING, DISP USING, or IMAGE statements.

Redefining the Printer and the Display

The PRINTER IS and the CRT IS statements are used to "redefine" the printer and the CRT. Although the statements are most often used with peripherals, you can tell the HP-85 that the display is the printer (CRT IS 2); and all display messages from DISP, DISP USING, LIST, Errors, and Warnings will be printed rather than displayed. Or, you can define the printer as the display (PRINTER IS 1); all information from PRINT, PRINT USING, PLIST, and TRACE statements will be displayed rather than printed. The PRINTER IS and CRT IS statements are programmable.

PRINTER IS output code

CRT IS output code

Code	Device
1	CRT
2	PRINTER

For instance, execute:

```
PRINTER IS 1
```

Redefines the printer to be the CRT (display); all PRINT statements will be displayed rather than printed.

```
10  PRINT
20  PRINT "**************"
30  PRINT "* NOW CRT IS *"
40  PRINT "*   PRINTER  *"
50  PRINT "**************"
60  GOTO  10
```

Now run the program, the message will be "printed," repeatedly, on the CRT display until you press PAUSE to stop the program.

RUN

```
*    PRINTER   *
**************

**************
* NOW CRT IS *
*    PRINTER   *
**************

**************
* NOW CRT IS *
*    PRINTER   *
**************

**************
* NOW CRT IS *
```

This message will be continuously "scrolled" on the display until you press PAUSE.

PAUSE

After pressing PAUSE, press LIST. Your program will be listed on the display. You can return the system to normal output mode by typing `PRINTER IS 2`, or pressing RESET.

The same can be done with `CRT IS 2` to redefine the CRT. Once `CRT IS 2` is executed, all messages that are normally displayed on the CRT are output to the printer.

For instance:

```
CRT IS 2
DISP "SQR(86)=";SQR(86)
```

These statements cause the following to be printed:

```
SQR(86)= 9.2736184955
```

Again, return the system to normal output mode by executing `CRT IS 1`, or by pressing RESET. The `PRINT ALL` and `COPY` statements are unaffected by the `PRINTER IS` and `CRT IS` statements: `PRINT ALL` and `COPY` always transfer the information from the display to the printer.

Problems

10.1 While considering the variations of social and economic factors among nations of the world, you decide to use the populations, areas, and annual gross national products (GNPs) of various nations to determine their population densities (by dividing the population by the area) and per capita GNPs (by dividing the

GNP by the population). You would like the results to be summarized for each nation. Write a program that requests the name, population, area (in square kilometers), and GNP (in U.S. dollars) of each nation, and prints a summary for each nation according to the following format:

```
  POPULATION      AREA    POP DENS
       ANNUAL GNP       GNP/PERS
--------------------------------
name of nation
 nnn,nnn,nnn   n,nnn,nnn   n,nnn.n
   $n,nnn,nnn,nnn,nnn   $nn,nnn
                :
```

The information below is available for 1977.

Nation	Population	Area (sq km)	Annual GNP (US $)
China	865,193,550	9,560,980	223,000,000,000
United States	216,817,000	9,363,123	1,781,400,000,000
Canada	23,469,142	9,976,139	195,785,000,000
Singapore	2,322,576	581	5,885,600,000
Mongolia	1,531,940	1,565,000	547,000,000
Qatar	97,792	11,000	4,044,000,000

10.2 In her studies of natural phenomena, physicist Shirley Bright encounters the extremes of length measurement—from the wavelengths of radiation (measured in angstroms) to intergalactic distances (measured in light-years). In order to relate these extremes to each other, Ms. Bright would like to see how a given measurement is expressed in a number of different units, specifically angstroms, meters, and light-years. There are 10^{10} angstroms in a meter and 9.460×10^{15} meters in a light-year. Write a program that converts a measurement (entered as a numerical value and a dimensional unit—A,M, or L) into all three units and prints the three values. An exponential format should be used because of the extremely large and small numbers that are involved. The output should look like:

```
ANGSTROMS    METERS   LIGHT-YEARS
--------------------------------
n.nnnE+nn  n.nnnE+nn  n.nnnE+nn
n.nnnE+nn  n.nnnE+nn  n.nnnE+nn
```

Use this program to express in other units the wave length of light with greatest human visibility (5560 anstroms), the length of the Humber Bridge span in England (1410 meters), the wavelength of certain gamma rays (5.6×10^{-3} angstroms), the approximate diameter of the nucleus of an atom (10^{-14} meters), and the distance to the nearest galaxy (170,000 light-years).

10.3 As an aid in maintaining an accurate record of your checking account, you decide to write a program that takes a sequence of transactions that have occurred over a period of time and prints the status of your account after each transaction. The program is to be initialized by entering the current date and the balance in your account at the beginning of the period. Each deposit is entered as D, *amount*. Each check is entered as C, *amount*. The bank charges 22 cents for processing a check if your balance at that time is less than $275; there is no charge if your balance is at least $275. If a check (plus check charge) will overdraw your account, print a negative balance and a special warning giving the amount of the overdraft. Your account summary should have this format:

```
SUMMARY FOR date
 CHECKS   CHG DEPOSITS   BALANCE
---------------------------------
                         n,nnn.nn
n,nnn.nn   .nn           n,nnn.nn
               n,nnn.nn  n,nnn.nn
n,nnn.nn   .nn           n,nnn.nn
n,nnn.nn   .nn           n,nnn.nn
```

10.4 A regular polygon with n sides inscribed in a circle of diameter d has a perimeter p which is given by

$$p = (d)\,(n)\,\sin\left(\frac{\pi}{n}\right).$$

As the number of sides of the polygon is increased, the polygon more closely resembles a circle, and the ratio of its perimeter to the diameter, p/d, becomes closer to the constant π (which is the ratio of circumference to diameter for a circle). Write a program that lists the perimeter p and the ratio p/d for a series of polygons with n=3,4,5, and so on. Let the diameter d equal 35 units. Have the two columns start at character positions 3 and 19.

10.5 The `TAB` statement may be used to create a graph by varying the character position for each line of output. For example, the data below represents the average weight of a female during her first 18 years. Write a program to produce a printed plot of this data. Each of the 19 years can correspond to a printed line; the position of a printed symbol (such as `*`) can correspond to the weight.

The range of weights, plus the allowance of two spaces to print the age, suggests that the "*" should be printed at the position determined by TAB(3+W/2). It is also helpful to print a "+" at the position of every 10 units of weight across the top of that plot, and a " . " at the position corresponding to zero weight on each line. The plot should resemble the following:

```
   0    10   20   30   40   50
   +----+----+----+----+----+
0 . *
1 .    *
2 .     *
3 .      *
              :
```

Age (years)	Weight (kilograms)
0	3.2
1	9.5
2	11.9
3	13.9
4	15.7
5	17.6
6	19.1
7	21.9
8	24.8
9	28.1
10	32.4
11	37.1
12	41.5
13	46.2
14	50.5
15	53.8
16	55.7
17	56.7
18	56.7

Using Tape Cartridges

With your HP-85's built-in tape drive, storing and retrieving programs and data on tape cartridges is convenient and easy. The Getting Started section introduced you to storing programs on magnetic tape. In addition, you can:

- Create data files.
- ''Print'' whole arrays onto the tape with only one program statement—and ''read'' them from the tape just as easily.
- Store an ''Autostart'' program that is automatically loaded and executed at power on.
- Run a large program by storing it in segments on the tape and bringing the parts into computer memory one at a time.
- ''Secure'' program and data files.

The Tape Directory

The tape directory is automatically set up by the computer, providing you with an easily accessible ''table of contents'' of recorded programs and data files. The directory can hold the names of, at most, 42 files. At your request, it directs the system to the exact tape location of recorded programs and data. You need to set up the tape directory only the first time you use a new tape with the HP-85 or whenever you wish to set up a new directory on an old tape whose contents you no longer want.

CAUTION

Do not attempt to remove the cartridge while the tape is in motion or while the tape drive light is on. Damage to the tape and its contents may result.

Directory Set-Up

To set up the tape directory, make sure that the RECORD slide tab is in the right-most position and then initialize the tape with the ERASETAPE command. The ERASETAPE command renders all previously recorded information on the tape inaccessible.

```
ERASETAPE
```

You must initialize any tape being used for the first time and any recorded tape that is to be erased for re-use. If you execute CAT on a tape and a READ or SEARCH error appears in the display, the tape probably needs to be initialized. (For recurring READ errors with a tape that has been initialized, see the Tape Care and Tape Life sections of appendix B.) *When a tape is initialized, all previously recorded information is erased.*

Cataloging

The CAT (catalog) command outputs a listing of file names, file types, and physical specifications, enabling you to review the directory contents and to determine the amount of available space remaining on a tape.

```
CAT
```

Place the Standard Pac cartridge into the system's tape transport and execute the `CAT` command. Here is the output you should see:

```
NAME    TYPE    BYTES   RECS FILE
MOVING  PROG     256     40    1
AMORT   PROG     256     18    2
POLY    PROG     256     29    3
SIMUL   PROG     256     47    4
ROOTS   PROG     256     19    5
CURVE   PROG     256     55    6
FPLOT   PROG     256     22    7
DPLOT   PROG     256     43    8
HISTO   PROG     256     36    9
TEACH   PROG     256     27   10
CALEND  PROG     256     22   11
BIORHY  PROG     256     21   12
TIMER   PROG     256     30   13
COMPZR  PROG     256     56   14
SKI     PROG     256     20   15
MUSIC   DATA     256     44   16
```

Each line of the directory output describes one file in the following manner:

`NAME` The name given to the file when a program or data is first stored on tape.

`TYPE` The contents of the file, i.e., `PROG` (program), `DATA`, `NULL` (more on `NULL` later), or `BPGM` (binary program).

`BYTES` The number of bytes per logical record.

`RECS` The number of defined records in the file.

`FILE` The file number assigned by the computer.

Each of the above will be described in the following pages.

Recording and Retrieving Programs

The STORE Command

After a program has been entered into computer memory, executing the `STORE` command attaches a name that you specify to the program, creates a program file, and stores the program on tape in the computer's unique internal language under the name specified.

`STORE` program name

Although the program name is most often specified as a quoted string of at most six characters (e.g., `STORE "PRGM1"`), the name can be any string expression except the null string (e.g., `STORE A$&CHR$(8)`). Longer names are truncated to six characters. Any combination of characters *except* the null string and quotes within quotes (''PRGM''A'''') can be used. Remember that spaces are also characters. Because you have the choice of a wide range of character combinations in program names, you may find it advantageous to choose program names that serve to identify the application of the program. For example, a program written to statistically analyze the performance of a business might be named `STAT`, while a program designed for inventory control might be named `INVENT`.

Place a tape cartridge in the tape transport and then enter the following program for arranging numbers in ascending order. If the tape has not been previously used, remember to initialize it with an `ERASETAPE` command. (If you do not wish to do the example program, just key in the first and last program statements and go to page 178.)

```
 10 REM *THIS PROGRAM WILL LIST
    NUMBERS IN INCREASING ORDER
 20 OPTION BASE 1
 30 DIM L(100)
 40 DISP "THIS PROGRAM WILL SORT
     FROM 2 TO 100 NUMBERS IN AS
    CENDING ORDER"
 50 DISP
 60 DISP "HOW MANY NUMBERS ARE T
    O BE SORTED";
 70 INPUT A
 80 A=IP(A)
 90 IF A>100 THEN DISP "TOO MANY
     TO SORT" @ GOTO 60
100 IF A<2 THEN DISP "DON'T BE R
    IDICULOUS" @ GOTO 60
110 DISP
120 DISP "TYPE THE LIST OF NUMBE
    RS ONE AT A TIME."
130 FOR I=1 TO A
140 INPUT L(I)
150 NEXT I
```

Inputs array elements (numbers to be sorted).

```
160 DISP
170 DISP "HERE IS THE ORIGINAL L
    IST OF NUMBERS:"
180 FOR J=1 TO A
190 DISP L(J);
200 NEXT J
```

Displays original list.

```
210 DISP
220 FOR K=1 TO A-1
230 FOR M=K+1 TO A
240 IF L(K)<=L(M) THEN 280
250 R=L(K)
260 L(K)=L(M)
270 L(M)=R
280 NEXT M
290 NEXT K
```

Sorts numbers in ascending order.

```
300 DISP
310 DISP "HERE IS THE LIST ARRAN
    GED IN INCREASING ORDER:"
320 FOR I=1 TO A
330 DISP L(I);
340 NEXT I
350 DISP
360 END
```

Displays sorted list.

Test the sort program by inputting the following list of numbers to see that it arranges them in ascending order:

35, 10, 97, 67, 1, 29, 25, 80, 19

We executed the PRINT ALL command before running the program to get the printout below:

```
THIS PROGRAM WILL SORT FROM 2 TO
 100 NUMBERS IN ASCENDING ORDER
HOW MANY NUMBERS ARE TO BE SORTE
D?
9
TYPE THE LIST OF NUMBERS ONE AT
A TIME.
?
35
?
10
?
97
?
67
?
1
?
29
?
25
?
80
?
19
HERE IS THE ORIGINAL LIST OF NUM
BERS:
 35  10  97  67  1  29  25  80
 19
HERE IS THE LIST ARRANGED IN INC
REASING ORDER:
 1  10  19  25  29  35  67  80
 97
```

Now record the sort program on the tape cartridge using the STORE command:

```
STORE "SORT1"
```

The above program is now stored on the tape cartridge under the name SORT1. (It also remains in computer memory.)

To see how the system has handled SORT1, execute CAT.

```
CAT
```

Your display should show SORT1 listed as follows:

```
NAME     TYPE   BYTES   RECS  FILE
 .        .       .       .     .
 .        .       .       .     .
SORT1    PRGM   256      7     *
```

* Any other programs you have recorded on the tape will be listed ahead of SORT1 unless a large enough NULL file exists. Most of the time, the file number of a program will match the program's chronological number in the file. (NULL files are reused if the program fits.)

The LOAD Command

After a program has been placed on tape using a STORE command, it can be retrieved as often as you wish by executing a LOAD command.

LOAD program name

When retrieving a program with LOAD, the program name you use must be the same one assigned to the program when it was recorded with STORE. The program name can be a quoted string or any other string expression that specifies the name. The SORT 1 program you just recorded would be retrieved with the following command:

```
LOAD "SORT1"
```

Executing LOAD automatically scratches any program currently in computer memory before loading the new program. The operation also scratches all current data. The LOAD command, like the STORE command, can be used with any string expression that specifies program name. Just remember that calculator mode variables are scratched when a program is loaded. For instance, to load the SORT1 program using the string variable S$, first assign SORT1 to S$.

```
S$ = "SORT1"
```

Then execute LOAD with the S$ substitution.

```
LOAD S$
```

Now check computer memory by executing LIST.

```
LIST
```

When the program is loaded into computer memory with the LOAD command, computer memory is scratched; S$ will no longer be defined.

The CHAIN Statement

The CHAIN statement is similar to the LOAD command, with four important differences:

- The specified program is executed automatically and immediately as soon as it is loaded into memory.
- As with the LOAD command, CHAIN destroys any program and data in memory. However, any data stored in common is preserved if the chained program has a COM statement.
- The CHAIN operation requires that if the program doing the CHAIN has a COM statement, then the CHAINed program must have a COM statement and the COM statements must agree.
- The CHAIN statement is programmable.

The syntax for CHAIN is simply:

CHAIN program name

The chained program must have been previously stored with the STORE command. Again, the program name must be a string expression that specifies the program file.

Chaining allows programs of unlimited size to be run by breaking up the program into smaller segments. A CHAIN statement in the first segment directs the system to load and run the next segment, preserving variables in common. A CHAIN statement in the second segment directs the system to load and run the next segment, preserving variables in common, and so on. (Refer to COM, page 123.)

Autostart

An automatic start capability is enabled by storing a program by the name Autost:

```
STORE "Autost"
```

You may have noticed that as soon as you turn the power on, the amber tape drive light blinks on and then off again if a tape is in the tape drive. The system is searching for an Autost program. At power on, it is instructed to:

```
LOAD "Autost"
```

If a tape cartridge is present in the tape drive at power on and contains a program named Autost, that program will be loaded and executed automatically.

The autostart routine permits the computer to load and run a supervisory program automatically, which in turn could define special function keys or chain other programs without operator instructions.

Using Data Files

Along with program storage and retrieval, many computer applications involve the storage, retrieval, use, and updating of data file. You have seen how the STORE and LOAD commands control the tape-related operations of programming. Because the nature and use of data sets can vary widely, the HP-85 system features five basic data file operations: creating a file, opening a file, recording data, retrieving data, and closing a file. Later in the section, we will discuss the purging, renaming, and securing of files.

Creating a Data File

All data files are initiated with a CREATE statement.

CREATE file name , number of records [, record length]

The file name must be a quoted string (or any string expression) from one to six characters in length. (The null string is not allowed.) File names serve as identifiers in the NAME column of the CAT output. For this reason it is best to select file names that best describe the contents of the data files they represent.

Records

Each file you create will be made up of one or more records. A record is the smallest addressable unit on a tape. The number of records you specify for a data file should, in most cases, correspond to the obvious subdivisions in your data. Note that all records in a given file will be of the same length.

Among the features included in your HP-85 is the option to control record length. By specifying record lengths to match your data sets, you can optimize your use of space on the tape cartridges. There are two types of data records:

- **Physical Records** are a standard 256 bytes in length and are *automatically* established with the STORE command or whenever [, record length] is omitted from the CREATE statement.

```
CREATE "Data 1",4
```

File Name — No. of Records

The statement above creates a data file named Data 1 containing four records. Because no record length was specified in the CREATE statement, the records are automatically 256 bytes in length. At most, 850 records of 256 bytes each are allowed in any one CREATE statement.

- **Logical Records** can be as small as 4 bytes in length or as large as 32K−1 (32,767) bytes in length, depending on the size you specify in the CREATE statement.

```
CREATE "Data 2",4,100
```

File Name — No. of Records — No. of Bytes per Record

The statement above creates a data file named Data 2 containing four logical records. Because a record length of 100 bytes was specified in the statement, each of the four records is 100 bytes long.

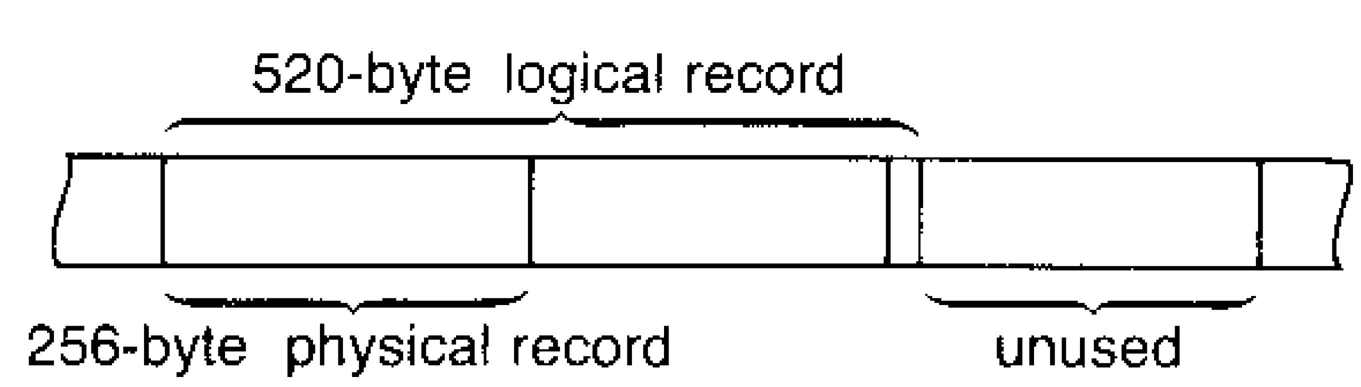
Data Storage

In many cases, logical records will provide the kind of space utilization you want. The following chart description of the number of bytes consumed by numeric and string variables will help you plan logical record sizes for your data files.

Type	Numbers	Strings
Single variable	8 bytes per number	1 byte per character + 3 bytes per string + 3 bytes each time the string crosses into a new defined record.
Array variable	8 bytes X the dimensioned number of elements	None.

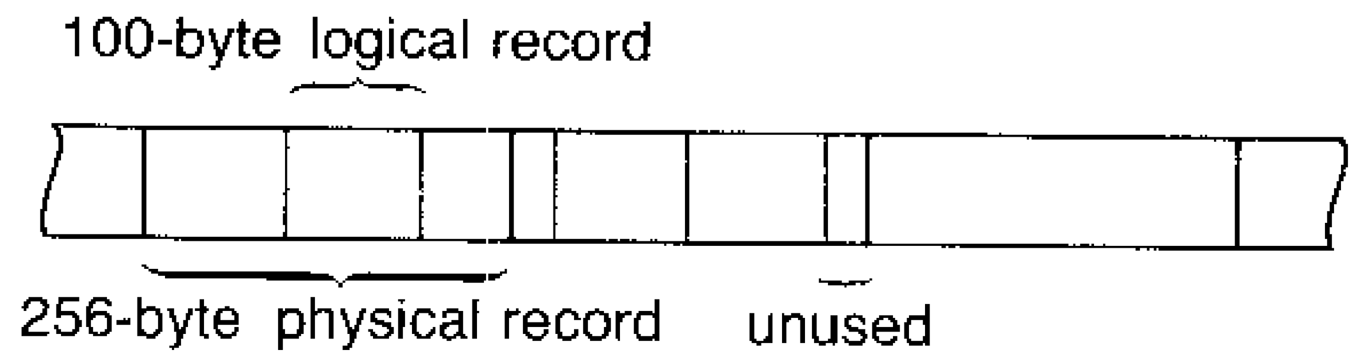
Any *number* of any type (INTEGER, SHORT, or REAL) consumes eight bytes. This means that .1,100, and −49987.532927E499 all consume eight bytes each. All strings consume 1 byte per character.

By summing the number of bytes of storage your data requires, you can tailor your file and define record lengths to suit your needs and minimize waste. However, keep in mind that a file always begins on a new physical record. If you specify a file containing one 520-byte *logical* record (e.g., CREATE "DATA",1,520), three 256-byte physical records are required. So 248 bytes are unused, and therefore, wasted space.



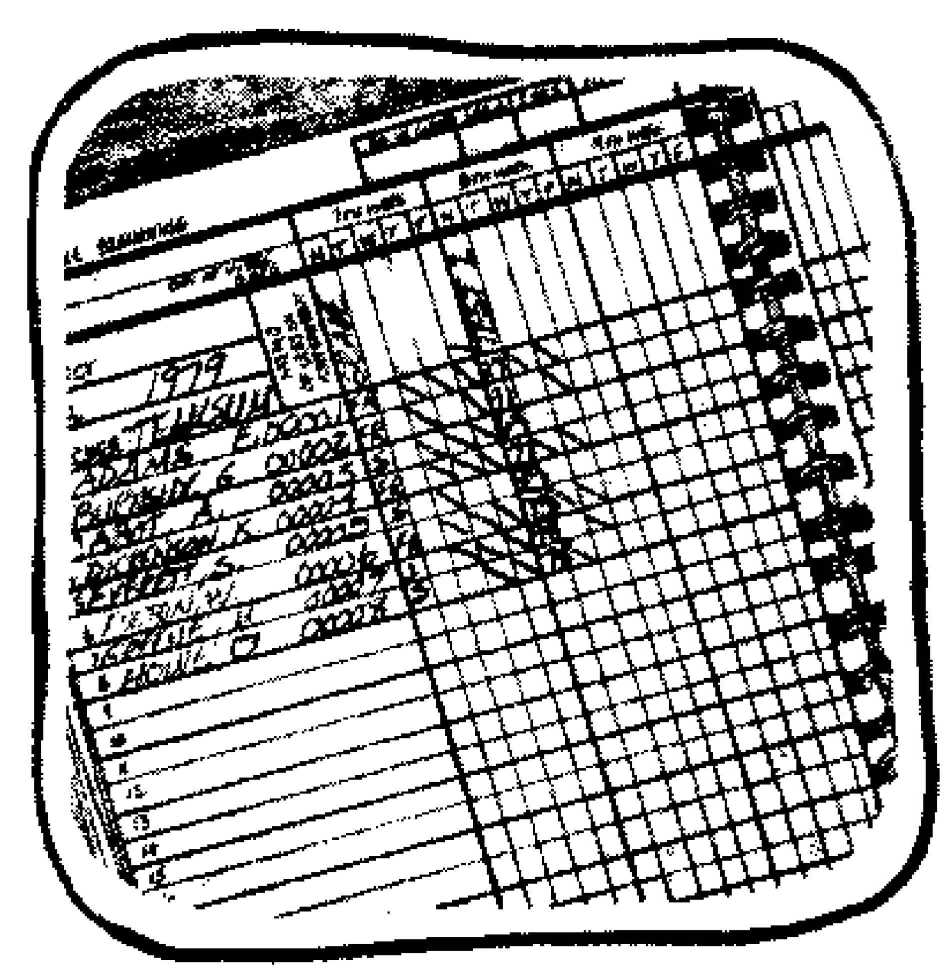
But, whenever you create logical records, the system automatically creates as many logical records as will fit within the physical record space required.

For instance, four 100-byte logical records span two physical records of 256 bytes each. Thus the statement, `CREATE "FILE",4,100`, actually creates five records of 100 bytes each. This way, only 12 bytes are rendered inaccessible rather than 112 bytes.



If a large enough `NULL` file (resulting from a purged file or from a stored program that, after editing, grew too big for its original file) exists on the tape when you create a file, the `NULL` file will be reused. The system will also completely fill the `NULL` file with logical records of length specified in the `CREATE` statement. For instance, if a `NULL` file containing six physical records (1536 bytes) exists on the tape when you execute `CREATE "FILE",4,100`, a total of 15 records of 100 bytes each will be created. To avoid wasting space on the tape, you may wish to store a "DUMMY" program in the `NULL` file until a larger program or file can be stored there.

Example: In a class record book, Professor I. Tehlsum has entered the name, student number, year, and first test score of each of eight computer students. If Tehlsum also wants to make a simple record of the data on an HP data cartridge, how much space will be required?



Name	Student No.	Year	Score
Adams E	00001	Frsh	87
Burnside G	00002	Frsh	91
Cabot A	00003	Soph	63
Dickenson K	00004	Frsh	77
Everett S	00005	Soph	85
Fulton W	00006	Frsh	75
Greene M	00007	Junr	98
Howe O	00008	Soph	78

The student names, numbers, and years each represent separate strings. (Student numbers are stored as strings in order to preserve leading zeros, e.g., `"00001"`.) Each score is a separate numeric variable. How many physical records will Tehlsum's student data require? The professor quickly totals the approximate data space needed as follows:

8 names, allowing 12 characters per name	= 96 bytes
+ 3 bytes for each string.	= 24
8 student numbers of 5 bytes each	= 40
+ 3 bytes for each string.	= 24
8 "year" strings of 4 bytes each	= 32
+ 3 bytes for each string.	= 24
8 numeric variables (scores) of 8 bytes each	= 64
Total bytes	304

The student data requires approximately 304 bytes. As 256 bytes represents the maximum a single physical record can hold, either two physical records of 256 bytes each, or one logical record of 304 bytes will be required.

Place a tape cartridge in your HP-85's tape drive and create a file for Professor Tehlsum's student data.

```
CREATE "CLASS1",1,304
          ↑      ↑  ↑
        File  No. of  Record
        Name  Records Length
```

The result of the above operation is a data file named CLASS1, composed of one logical record of 304 bytes. The directory now contains a record of the data file you have created. Execute CAT to display:

```
CAT
NAME      TYPE     BYTES    RECS FILE
CLASS1    DATA      304        1    1*
```

When you create data files, be sure that the length and number of your physical and logical records suit the storage requirements of your data. If, in the example above, you had specified one record of 250 bytes instead of 304 bytes (CREATE "CLASS1",1,250), an EOF (end-of-file), RANDOM OVF, or RECORD# error would occur if you attempted to record the data. EOF errors result whenever an attempt is made to store more bytes of data than the file has been specified to handle.

Opening a Data File

Even though a data file has been established in the directory with a CREATE statement, it cannot be accessed for data storage until it is "opened" using the ASSIGN# statement.

ASSIGN# buffer number TO file name

The buffer number is a number from 1 through 10. The file name must be a quoted string or any other string expression that specifies the name of a previously created file. To access any data file, whether just created or previously established, the file must be opened by assigning it to a buffer.

Buffers reduce tape wear and increase efficiency by reducing the number of transfers to the tape. They function "behind the scenes" in your HP-85 computer, so you don't have to be too concerned about how they work. However, it may be helpful to know that data you wish to record on tape is first collected in a 256-byte buffer. The buffer is allocated in computer memory when you assign it to a file name; it returns to regular read/write memory (available to the user) when you're through using it for buffering data. The HP-85 can reserve at most 10 data buffers at any time.

PRINT# statements (see page 185) cause data transfers to the buffer (rather than to the tape); each time the buffer's contents reach 256 bytes of data, it automatically records the data onto the tape. You'll know when the buffer records or retrieves data from the tape: the display will blink off and the tape drive light will blink on as the tape is being accessed.

* (CLASS1 will be file 1 if no other files precede it on the tape.)

A buffer that is assigned to a file name is also recorded under these conditions:

- Assigning that buffer number to a different file.
- Executing a PAUSE, (PAUSE), STOP, or END.
- Pressing any key that interrupts program execution.
- Closing the file.
- Random access to another logical record (we'll discuss this shortly).
- Executing a PRINT# statement from the keyboard.

If you wish to have more than one file open at a time, simply add to your instructions a separate ASSIGN statement and buffer number for each additional file. Just remember that assigning a buffer currently in use to a new file "closes" the current file and reassigns the buffer to the new file.

Examples:

```
150 ASSIGN# 1 TO "CLASS1"
160 ASSIGN# 4 TO "Data12"
170 ASSIGN# 10 TO "NUMS"
```

Buffer No. — File Name

Assigns buffer 1 to CLASS1.
Assigns buffer 4 to Data12.
Assigns buffer 10 to NUMS.

Once you have created a file and opened it (by assigning a buffer to it), you can record data in the file or read data from a previously recorded file.

Closing a Data File

When you finish recording data into a file, always close the file with a special ASSIGN statement before proceeding with other operations.

ASSIGN# buffer number TO *

The ASSIGN statement used in closing a file is always identical to the ASSIGN statement used to open the file, *except* the file name is replaced by an asterisk. For example, the ASSIGN statement we will use later to close the student data file will appear as follows:

```
ASSIGN# 1 TO *
```

Buffer Number

Remember from our section on *buffers* that some information is kept in memory instead of being immediately stored on the tape. If a program error causes a halt when some information is in the memory buffer but not yet on tape, that part of the data in *memory only* will be lost unless you *close* the file. When you close a file (ASSIGN# buffer number TO *) all information in the buffer is recorded on the tape and the buffer is released. Since the buffering process is "invisible," it is important to close all files after use, even when errors occur.

Storing and Retrieving Data

A data file opened by an ASSIGN# statement is ready for data access. There are two methods for both data storage and data retrieval operations: **serial file access** and **random file access**. Your choice of access depends on the way in which your data is to be used.

Serial File Access

Serial file access is normally used to record or retrieve masses of data items sequentially, without regard to logical records. Your use of serial file access should be planned for applications requiring this non-selective form of recording and retrieval. Collections of string or numeric data that will be recorded or retrieved sequentially as a complete list are the best candidates for serial file access.

When data is recorded onto the tape serially, it can be longer or shorter than the logical record length. (For example, a long string might span three logical records.) For each data file opened, a file pointer keeps track of the data item currently being accessed. As you store or retrieve data, the pointer *moves serially forward* through the file.

Writing Serial Files

The serial PRINT# statement records values into the specified file from the variables, strings, or numbers in computer memory without regard to record subdivisions.

PRINT# **buffer number** ; **print # list**

The buffer number (1-10) used in the PRINT# statement must be the same buffer number you have assigned to the file with the ASSIGN# statement.

The print# list is a sequential listing of data items identified for writing (recording) on tape. Items in the print# list are separated by commas and can be numbers, variables, strings, or array names.

If nothing has been stored or retrieved, writing begins at the beginning of the file specified. Otherwise, writing begins immediately after the data item most recently stored or retrieved.

Example:

```
110 FOR S=1 TO 8
120 INPUT A$,B$,C$,D
130 PRINT# 1 ; A$,B$,C$,D
140 NEXT S
```

These statements record values for A$, B$, C$, and D onto a tape file through buffer #1. The file pointer remains at the end of the data list until:

- A PRINT# statement is executed, writing data to the same file.
- The file is closed.
- A READ# statement is executed (more on READ# shortly).

Advanced Programming Note: When storing a long *string,* it might be too long to be contained in one logical record. As long as you have already allocated enough records in the file, the string is automatically broken up into as many pieces as logical records that are needed. This adds *three* bytes for each time the string crosses over into another logical record. These bytes compose the string "header," which identifies the parts of the string as first, intermediate, or last, and specifies the length of each part.

But, when insufficient space remains in a nearly full record to completely write all eight bytes of a *number*, the number will be written entirely in the next record. The remaining bytes in the first record are unused, resulting in from one to seven bytes of wasted space. This happens because all numbers require eight bytes of storage space on the tape, and cannot be split between two records.

For example, if a record has only five bytes remaining in which to write a number, the five will be skipped and the number will be written in the first eight bytes of the following record. Note that crossing over into a new record to write a number does not cause a header to be written. Only string data, which can be split between two or more records, require headers.

When you set up a serial file of two or more records, avoid an unexpected space shortage by allowing some extra bytes in the record length. This will compensate for any additional bytes that might be consumed by strings or numbers crossing record boundaries.

The length of the data in the list must be less than or equal to the storage space that remains in the file after the pointer; otherwise, an EOF (end-of-file) error occurs, signaling that you have filled your file.

Example: By following the steps covered in the preceding pages, write a program to create a file for Professor Tehlsum's student data, assign a buffer to the file, and record the information from the table on page 182 onto the tape. (Do not use a `CREATE` statement in the program if you have already created the file.)

```
 10 CREATE "CLASS1",1,304                       Creates the file.
 20 ASSIGN# 1 TO "CLASS1"                       Opens the file.
 30 DISP "TYPE IN STUDENT NAME,                 User instruction.
    NUMBER, YEAR, AND SCORE."
 40 FOR S=1 TO 8
 50 INPUT A$,B$,C$,D                            Inputs data.
• 60 PRINT# 1 ; A$,B$,C$,D                      Records data in buffer assigned to the file.
 70 NEXT S
 80 ASSIGN# 1 TO *                              Closes the file.
 90 DISP "END"                                  Signals end of program.
100 END                                         Halts program.
```

Now run the program to input the student information and record it in a data file.

```
RUN
TYPE IN STUDENT NAME, NUMBER, YE
AR, AND SCORE.
?
ADAMS E, 00001, FRSH, 87
?
BURNSIDE G, 00002, FRSH, 91
?
CABOT A, 00003, SOPH, 63
?
DICKENSON K, 00004, FRSH, 77
?
EVERETT S, 00005, SOPH, 85
?
FULTON W, 00006, FRSH, 75
?
GREENE M, 00007, JUNR, 98
?
HOWE O, 00008, SOPH, 78
END
```

The student data is now recorded. At this point the recorded tape can be removed from the tape drive and stored for future use. Now let's discuss the serial READ# statement to retrieve the data you've recorded.

Reading Serial Files

Before you can use data that has been stored in a data file with a PRINT# statement, you must read the data back into computer memory with a READ# statement. Data is merely copied into computer memory.

Using the serial READ# statement, you can read numbers, strings, or array values from recorded files into computer memory in the same way that they were recorded using a PRINT# statement.

READ# buffer number ; read list

The read list, like the print list, is the sequential listing of data items identified for reading from the tape. Items in the read list are separated by commas. The variables in the read list do not have to have the same names as specified in the PRINT# statement. But, the variable names in the read list must match in number and type (string vs. numeric) the PRINT# statement(s) print list previously stored. If the READ# statement(s) specify more data items than were originally stored, an EOR (or EOF) error occurs, meaning there is no more data.

Remember, data read must correspond to the type—numeric or string—that was printed. However, a numeric item need not be of the same precision (REAL, INTEGER, or SHORT). Precision is automatically converted.

As we shall see, you can also record an entire array and read it back as simple variables or as other arrays, and vice versa.

To begin reading from the beginning of the file, you must reposition the pointer (using a random READ# statement, as you will see later) or do another ASSIGN. Data can be updated and restored into the file or into a new file.

Example: Write a program (or modify the last program, page 186) to read Professor Tehlsum's student data from the CLASS1 file and display it on the CRT.

Note: You do not need to use another CREATE statement; the file already exists on the tape.

```
 20 ASSIGN# 1 TO "CLASS1"                  Opens the file (assigns a buffer to it).
 30 DISP "HERE ARE STUDENT NAMES           User instructions.
    , NUMBERS, YEARS, AND SCORES
    ."
 40 FOR S=1 TO 8
 60 READ# 1 ; S$,N$,Y$,T1                  Variable names do not need to be the same
                                           as in the PRINT# statement but they
                                           must agree in number and type (numeric
                                           or string).
 65 DISP S$,N$,Y$,T1
 70 NEXT S
 80 ASSIGN# 1 TO *                         Closes the file.
 90 DISP "END"
100 END
```

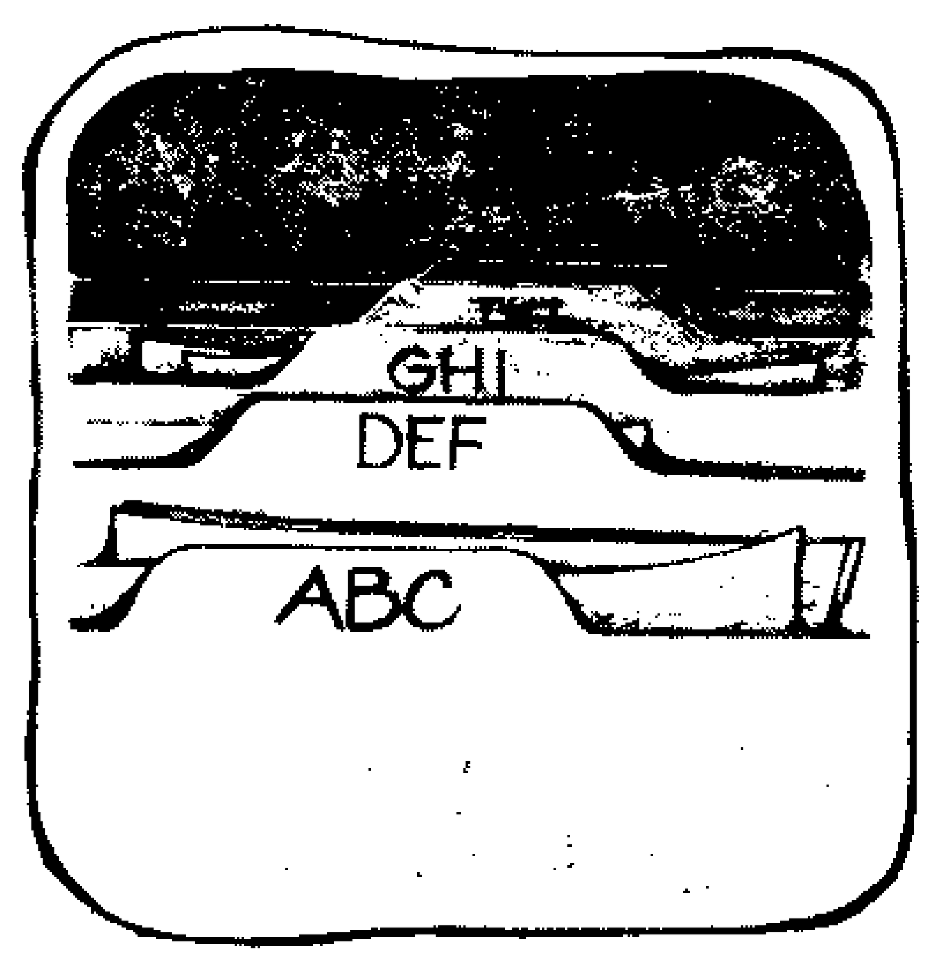
Now run the program to display the information that was previously printed in the data file.

```
RUN
HERE ARE STUDENT NAMES, NUMBERS,
YEARS, AND SCORES.
ADAMS E                 00001
FRSH                     87
BURNSIDE G              00002
FRSH                     91
CABOT A                 00003
SOPH                     63
DICKENSON K             00004
FRSH                     77
EVERETT S               00005
SOPH                     85
FULTON W                00006
FRSH                     75
GREENE M                00007
JUNR                     98
HOWE O                  00008
SOPH                     78
END
```

Note that in serial READ# operations the data pointer moves in the same manner as it does in serial PRINT# operations, marking in computer memory where the last data item ended and where the next data item should begin.

Random File Access

Random file access is used to store or retrieve data items from a specific logical record, thus, the name—you can access any record at random. Random file access requires you to specify the logical record that you wish to access. The file pointer is positioned at the beginning of that record.



You should use random file access methods when you wish to be able to store or retrieve groups of data items from specific logical records by referring to the record number.

Random Writing

The random PRINT# statement is nearly identical to the serial PRINT# statement except that:

- The record number must be specified.
- Data is recorded into the file starting at the beginning of the specified record.
- End of record marks are not ignored; thus, data cannot be larger than the logical record length (e.g., a long string must fit entirely within the record specified with random file access).

PRINT# buffer number , record number [; [print# list]]

The print# list is identical to that used in the serial PRINT# statement. The random PRINT# statement records data into the specified record of the file. With random file access, printing always starts at the beginning of the specified logical record. Any previous data is overwritten. Any data not overwritten because the new logical record is shorter is rendered inaccessible by the new end-of-record mark.

Again, the print# data list must fit in the logical record or else an EOR error occurs. If you attempt to specify a logical record number greater than the number of records specified in the CREATE statement, an EOF error occurs.

When no print# list is specified and a semicolon follows the record number, the data pointer is repositioned to the beginning of the specified record.

Examples:

`170 PRINT# 3,1 ; A,B`	Records A and B in record 1.
`180 PRINT# 3,2 ; C,D`	Records C and D in record 2.
`190 PRINT# 3,3 ;`	Repositions the pointer to the beginning of record 3.

Example: Write a program that creates a new file for Professor Tehlsum's student data so that information pertaining to a particular student may be accessed at random; i.e., create one record for each student. Then read the information from the serial file that you created (page 182) and write it into the new file.

`10 CREATE "STUDE1",8,40`	Creates a file of 8 records, 40 bytes each. (The system actually creates 12 records to span two physical records.)
`20 ASSIGN# 1 TO "STUDE1"`	Opens the file.
`30 ASSIGN# 2 TO "CLASS1"`	Opens previously recorded file (see program, page 186).
`40 FOR I=1 TO 8`	
• `50 READ# 2 ; S$,N$,Y$,T1`	Reads data from old file into buffer 2.
• `60 PRINT# 1,I ; S$,N$,Y$,T1`	Writes data into specified record.
`70 NEXT I`	
`80 ASSIGN# 1 TO *`	Closes STUDE1 file.
`90 ASSIGN# 2 TO *`	Closes CLASS1 file.
`100 DISP "END"`	Signifies end of program.
`110 END`	

In the program above, when I = 1, statement 60 writes the values of the variables into the first record:

```
PRINT# 1,1 ; S,N$,Y$,T1
```

When I = 2, statement 60 writes the values of the variables into the second record:

```
PRINT# 1,2 ; S$,N$,Y$,T1
```

And so on for the remaining six records.

Random Reading

The random READ# statement is like the serial READ# statement except that reading of data into computer memory starts at the beginning of the specified logical record and will not read past an EOR or EOF mark. It is called "random READ#" because you can read a particular record at random.

READ# buffer number , record number [; variable list]

Again, as in the serial READ# statement, the variables into which you read values do not necessarily have to have the same names, precision, or type (INTEGER, REAL, or SHORT) as specified in the PRINT# statement. But the variable names must match the data as to string or numeric type.

If the number of items making up the variable list is greater than the data in the logical record, an EOR error occurs.

Example: Now professor Tehlsum can easily access and change or update the data in each student's record. Write a program that will read data from a particular record of your choosing, display the data, and accept changes. Then incorporate the changes by rewriting the record.

```
  10 ASSIGN# 1 TO "STUDE1"                  Opens the file.
  20 DISP "WHICH RECORD DO YOU
     WISH TO VIEW (1-8)";
  30 INPUT R
• 40 READ# 1,R ; S$,N$,Y$,T1                Reads data from record.
  50 DISP S$,N$,Y$,T1                       Displays contents of record.
  60 DISP "ANY CHANGES (Y/N)";
  70 INPUT A$
  80 IF A$="N" THEN 120
  90 DISP "ENTER NEW TEST SCORE"
 100 INPUT A1                               Inputs new score.
•110 PRINT# 1,R ; S$,N$,Y$,A1               Rewrites record with change.
 120 DISP "DO YOU WISH TO VIEW AN
     OTHER RECORD (Y/N)";
 130 INPUT A$
 140 IF A$="Y" THEN 20
 150 ASSIGN# 1 TO *                         Closes the file.
 160 DISP "END CHANGES"
 170 END
```

Repositioning the Pointer

If the semicolon and variable list are omitted from the random READ# statement or from the random PRINT# statement, the file pointer is repositioned to the beginning of the specified record. To reposition the pointer to the beginning of a file (e.g., for use with serial file access) execute:

```
READ# buffer number , 1            All statements reposition the file pointer
PRINT# buffer number , 1           to the beginning of the specified record—
PRINT# buffer number , 1 ;         in this case, record 1.
```

Storing and Retrieving Arrays

Entire arrays can be stored and retrieved by using the following notation with the `PRINT#` and `READ#` statements with serial *or* random file access:

One-dimensional arrays: array name `()`.
e.g., `L()`, `X5()`, `T()`.

Two-dimensional arrays: array name `(,)`.
e.g., `L(,)`, `X5(,)`, `T(,)`.

Most of the time, a comma is used for documentation purposes. If an array has been dimensioned previously, its use is optional.

Arrays are stored and retrieved element by element without regard to dimensionality with the last subscript varying the fastest (in other words, by rows).

Example: The following program reads test scores and averages into array T, then, with one program statement, writes the entire array into one record.

```
 10 REM *SCORES AND AVES*
 20 OPTION BASE 1              Specifies lower bound 1 in arrays.
 30 DIM T(3,5)                 Dimensions array T, 3×5.
 40 FOR I=1 TO 3
 50 READ T(I,1),T(I,2),T(I,3),T(    Reads array elements.
    I,4)
 60 T(I,5)=(T(I,1)+T(I,2)+T(I,3)    Finds average.
    +T(I,4))/4
 70 NEXT I
 80 CREATE "TESTS",4,128       Creates file of 4 records, 128 bytes per record.
 90 ASSIGN# 1 TO "TESTS"       Opens file (assigns buffer).
•100 PRINT# 1,2 ; T(,)         Writes entire array in record 2.
110 ASSIGN# 1 TO *             Closes file (dumps buffer).
120 DISP "Array of test scores a
    re now recorded in RECORD 2"
130 DATA 78,43,69,81,98,99,92,97
    ,55,50,75,72
140 END
```

Now you can retrieve the data in record 2 in the same manner:

```
160 ASSIGN# 1 TO "TESTS"
•170 READ# 1,2 ; T(,)          Reads entire array from the file.
180 DISP "AVERAGES"
190 FOR I=1 TO 3
200 DISP T(I,5);
210 NEXT I
220 DISP
230 END
```

If you are appending this to the previous program, be sure to delete statement 140. (Also, delete statement 80 if you have created the file previously.)

Or you can read the data in record 2 of `TESTS` back into simple variables serially; but first you must reposition the pointer.

```
170 READ# 1,2                  Positions the pointer at the beginning of record 2.
180 READ# 1; A,B,C,D,E         Reads first five items.
190 DISP A,B,C,D,E             Displays test scores and average in first row of array.
```

Purging a File

The PURGE statement prevents access to any file (program, data, etc.) by removing its name from the name column in the tape directory. PURGE makes the specified file available for the storage of new programs or data.

PURGE "file name" [, purge code]

The file name must be the name of the file you wish to purge; the same name that is listed in the directory. If no purge code is specified, the system purges the particular file specified by returning the records of the file to "available storage space" on the tape. This is indicated by the word NULL in the type column of the directory. For example:

`PURGE "DATA1"`	Renders the previously created file inaccessible by removing its name in the directory.

Any future program or data file that fits will be stored or created in the first available NULL file; otherwise it is stored at the end of the list of previously stored programs or data files. Regardless of how small the program or data file is, if a large enough NULL file exists, the program or data file will be created within the NULL file. The directory will assume that you have used the entire NULL file to store the program (or create the data file, as we have seen earlier).

Sometimes it may be desirable to purge the tape *from a specific file to the end of the tape*. If, for instance, you eject the tape when it was in the process of recording information onto a file, that file and the rest of the tape might be rendered inaccessible. In order to make that part of the tape available for storage once again, specify a purge code of 0 following the file name. For example:

`PURGE "DATA1",0`	Returns DATA1 and any programs or data files that follow it on the tape to "available storage space."

A purge code of 0 removes the names of all files following and including the specified file from the directory. A NULL file marker is not written into the directory; the system will assume that you've only recorded programs or data up to the first purged file.

Any number, other than zero, used with PURGE operates in the same manner as specifying PURGE without a purge code; the specified file only is purged from the tape directory.

Renaming a File

The RENAME statement is used to give a file a different name.

RENAME old file name TO new file name

For example:

```
RENAME "TESTS" TO "TERM1"
```

Any file may be given a new name, including data files and secured files, as long as a name for the file exists in the directory.

Binary Programs

Some of the programs or routines in the application pacs are binary programs. A binary program is equivalent to a plug-in ROM except that you load it from a tape cartridge. Although binary programs will come with instructions about their use, for now, note that you can load or store binary programs with the LOADBIN and STOREBIN statements.

STOREBIN program name

A binary routine is retrieved and added to the current program in memory using the LOADBIN statement.

LOADBIN program name

The program name for both statements may be a quoted string from one through six characters in length or it may be any string expression specifying the program name. For example:

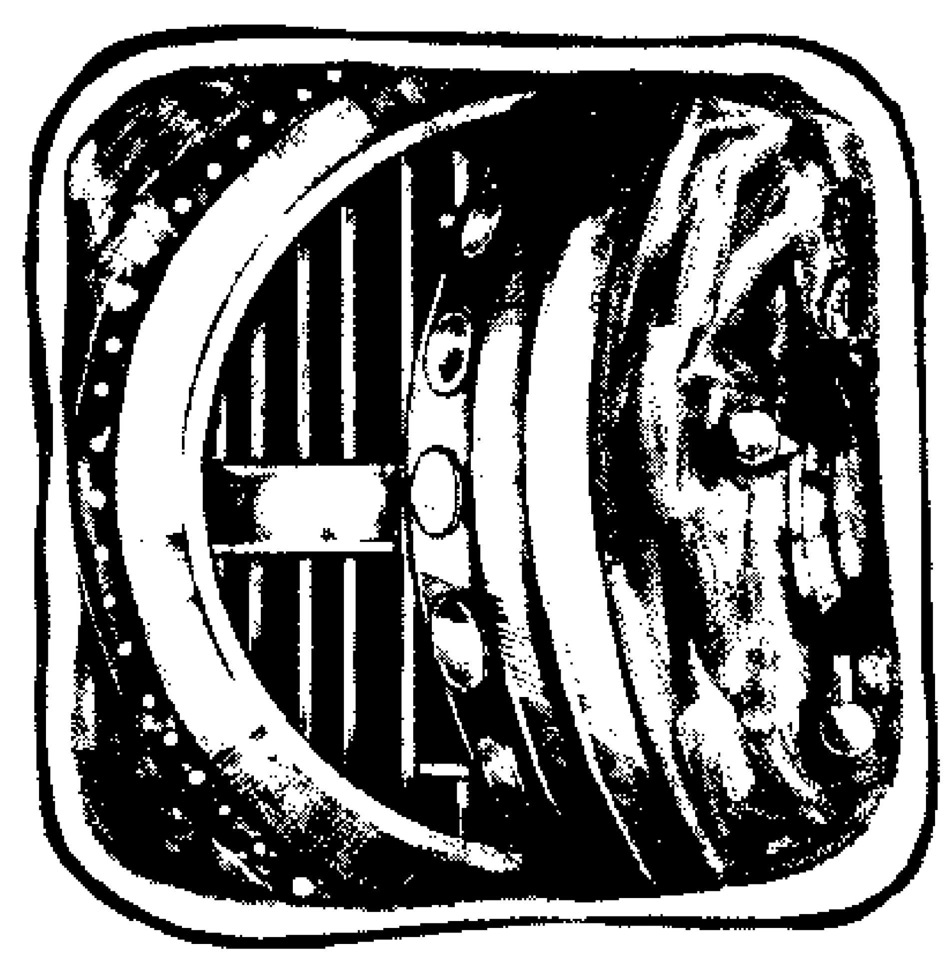
```
100 STOREBIN "ROOTS"
200 LOADBIN "MUSIC"
```

Stores binary program named ROOTS.
Loads binary program named MUSIC, without altering any existing program or data in memory.

One binary program, at most, may be in computer memory at any time. Remember that in order to edit a program that uses a binary routine, that binary routine must be present in computer memory.

Securing Files

File security enables you to prevent your programs or data from being copied or changed, or accidently overwritten.



The SECURE command is used to prevent specific program files from being listed, edited, or stored, to prevent any file's name from appearing in the directory listing, and to protect the user from writing over a file. The UNSECURE command is used to remove security on secured programs or data files.

SECURE file name , security code , secure type

The file name that is being secured must already be listed in the directory (i.e., the program or file being secured must already exist on the tape). It may be specified as either a quoted string or as any other string expression that specifies the file. The security code may be any string of characters except the null string. The system takes the first two characters of the string specified and stores them as the security code. If only one character is specified, the second character is a blank.

The security type is specified by a number from 0 through 3. Possible security types are:

Number	Secured Against
0	LIST, PLIST, and Editing
1	LIST, PLIST, Editing, and STORE (duplication)
2	STORE (overwriting), PRINT#, STOREBIN
3	CAT (a blank appears where name should be)

Security types 0 and 1 can be used with program files only. Type 2 can be used with any file. Type 3 is intended for data file security but it may be used with program files.

Examples:

```
SECURE "SPECS","DD",1
```

Secures program SPECS so that it cannot be edited, stored, or listed.

```
SECURE "DATA18","18C",3
```

Secures data file DATA18 so that in place of the name, blanks appear in the directory listing. The security code is 18, the first two characters of the string.

The SECURE command is programmable. You can secure a program file with more than one security type. For instance, you can secure a program file against listing, editing, and storing (duplicating) or secure its name from being listed in the directory or both. The only types you cannot secure at the same time are types 0 and 1.

The difference between STORE in types 1 and 2 is that a type 1 security prevents the program from being duplicated or stored in another file. Type 2 security protects a file from being over-written. If you attempt to store a program using the same name as a file secured with type 2 security, a warning that the file is write-protected will be output. Type 2 security protects against STORE, STORE BIN, and PRINT# only; you can always purge the file regardless of its security.

The UNSECURE command operates in the same manner as the SECURE command to remove the security on specified files. But remember you *cannot* unsecure an open data file. In other words, the SECURE and UNSECURE commands affect the tape file and the directory only—not the memory buffer. If you attempt to unsecure an open data file, you'll probably get a SECURE error since the buffer will register that the file is secured.

UNSECURE file name , security code , secure type

The file name must be that of a secured file and the security code types 0 and 1 must have the same first two characters as the code that was originally used to secure the file. You can use any two characters as the security code for unsecuring types 2 and 3.

Examples:

`UNSECURE "SPECS","DD",1`	Unsecures program `SPECS` so that it can be stored, listed, and edited.
`UNSECURE "DATA18","KH",3`	Unsecures data file `DATA18`. Note that the security code need not match the original security code for types 2 and 3. Now the data file name will appear in a directory listing.

The `UNSECURE` command is not programmable.

The HP-85 provides two more tape operations: `REWIND` and `CTAPE` (condition tape). They are discussed in appendix B.

Graphics

The graphics capabilities of your HP-85 truly enhance your BASIC programming power. HP-85 graphics enable you to:

- Plot data on the graphics display, thus clarifying complex sets of information in pictorial form.
- Scale the display yourself to desired proportions.
- Generate an unlimited number of lines, curves, diagrams, and designs on the display.
- Copy anything from the graphics display to the printer with one command.
- "Draw" and label graphs with ease.
- Interact with the graphics display from the keyboard.
- Execute any of the graphics commands from the keyboard or in a program.

The Graphics Display

The HP-85 provides two different display areas or modes: *alphanumeric* and *graphics*. Normally the display is in *alpha* mode, but you can view the current graphics display at any time by pressing the GRAPH key or by executing the statement, GRAPH.

GRAPH	Sets the display to graphics mode.

Any of the graphics statements that directly manipulate the graphics display also set the display to graphics mode automatically. You can return to *alpha* mode by pressing any alphanumeric key or display control key (such as the space bar or the ↓ key) or by executing the ALPHA statement:

ALPHA	Sets the display to alpha mode.

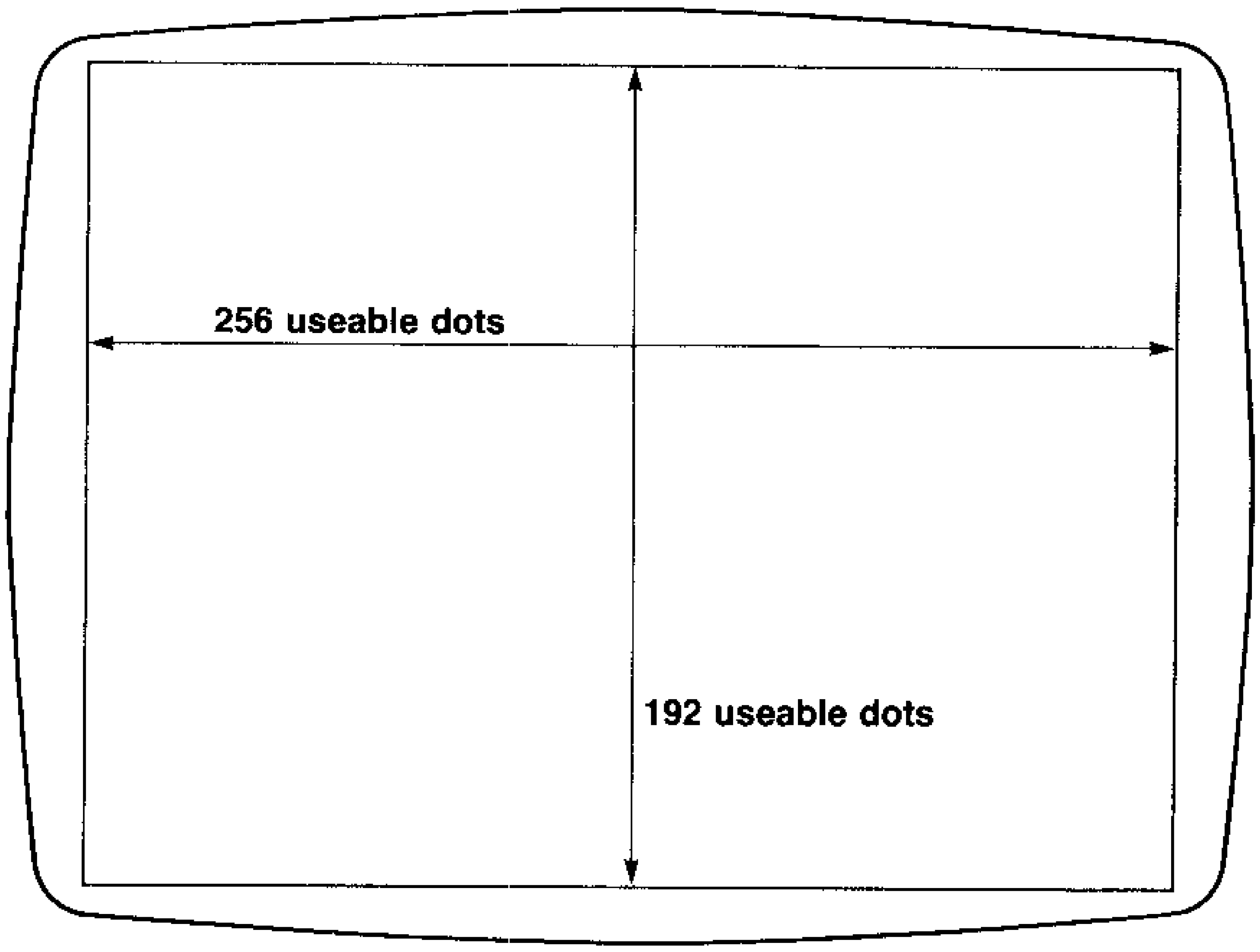
To get an idea of the graphics display area available for your use, enter the following program into the computer and then press RUN. This program will frame (draw a box around) the CRT graphics display area that is available to you.

First, press SCRATCH to clear the system memory of previous programs.

```
10 GCLEAR                          Clears the graphics display.
20 SCALE 0,100,0,100               Scales the graphics display.
30 XAXIS 0 @ XAXIS 100             Draws a frame around the plotting area.
40 YAXIS 0 @ YAXIS 100
50 END
```

This example program (and others like it found throughout this section) is given to provide some hands-on experience with HP-85 graphics and to illustrate various statements. Each of the graphics statements will be explained at appropriate places in the section.

When you run the program, the display shows:



This frames the graphics display area. You have 256 useable dots in the horizontal direction and 192 useable dots in the vertical direction, yielding a total of 49,152 points available for plotting.

By useable dots, we mean the actual physical dots of the graphics display screen. As you shall see, the display may be scaled to horizontal and vertical units of your own choosing. Points are plotted according to the current scale; they are automatically *mapped* onto the graphics display screen.

Line Generation

Line generation refers to the process of producing a line on the graphics display, which is similar to drawing a line with a pen. But the display has no actual pen. The display *does* have a point, referred to as "the pen," which when moved produces a line (or row of dots) if line generation is turned on *(pen down)*. If line generation is turned off *(pen up)* no line is produced, but the point moves.

Graphics and the Printer

The general method of performing HP-85 graphics is:

1. First generate your graph or design on the graphics display using the graphics statements either from the keyboard or within a program.
2. Then, to produce a hard copy of your graphics, simply set the display to graphics mode by pressing (GRAPH) and then press (COPY). In a program, these same operations can be performed by executing the GRAPH statement followed by the COPY statement. (GRAPH need not be executed if the display is already in graphics mode.)

The printer generates the graphics display sideways to assure that it fits properly on the paper and to enable strip charting.

Clearing the Graphics Display

The GCLEAR statement clears the graphics display of any previously plotted data.

GCLEAR [Y-coordinate]

The GCLEAR statement clears the graphics screen from the specified Y-value to the bottom of the screen. For instance, if the graphics display is scaled from 0 to 100 in the vertical direction, execute the following to clear the lower half of the display:

```
GCLEAR 50
```

Clears lower half of graphics display with vertical scale of 0 to 100.

If no parameter is specified, GCLEAR clears the entire graphics screen.

It is advisable to use the GCLEAR statement before you begin a new plot in a program, thus assuring that you do not plot over any previous graphics.

Execute GCLEAR now to clear the frame from our first graphics program. The display will change to alpha mode when you type in a graphics statement. It reverts back to graphics mode to show the change in the graphics display once the command is executed. The GCLEAR statement clears the graphic display to the current background "color" (more about this later).

Setting Up the Graphics Display

A program written to plot or draw lines on the graphics display usually includes some initial set-up operations to define the plotting area. Typical set-up operations might be clearing the display and framing it, as we did earlier. Most often, the display is *scaled* to the desired proportions before any plotting is done.

For instance, you might use the following group of statements to set up the graphics display.

```
10 GCLEAR
20 SCALE -10,10,-10,10
30 XAXIS 0,1
40 YAXIS 0,1
50 END
```

These statements will be discussed in the following pages.

The SCALE Statement

The SCALE statement defines the minimum and maximum values of the X (horizontal) and Y (vertical) directions for the graphics display. This enables you to specify your own units for plotting.

SCALE Xmin , Xmax , Ymin , Ymax

The first two parameters specify the values represented by the left and right boundaries of the graphics display. The last two parameters specify the values represented by the lower and upper boundaries of the display. If Xmax is less than Xmin or Ymax is less than Ymin, an error occurs.

At power on or after pressing (RESET), the minimum and maximum values of both X and Y directions are 0 and 100:

```
SCALE 0,100,0,100
```

Specifies X and Y units from 0 to 100.

The SCALE statement may be used to place the origin (point 0,0) on or off the graphics display.

For example, if you want to plot the average annual rainfall at a weather station for a 10-year period, the `SCALE` statement might look like this:

```
SCALE 1968,1978,0,20
```

The left edge of the graphics display area would represent the year 1968 and the right edge would represent 1978. Rainfall would be plotted in the Y direction in volume units (e.g., inches). This enables you to plot data in years and volume units (e.g., point 1976, 7) directly on the graphics display area.

More Examples:

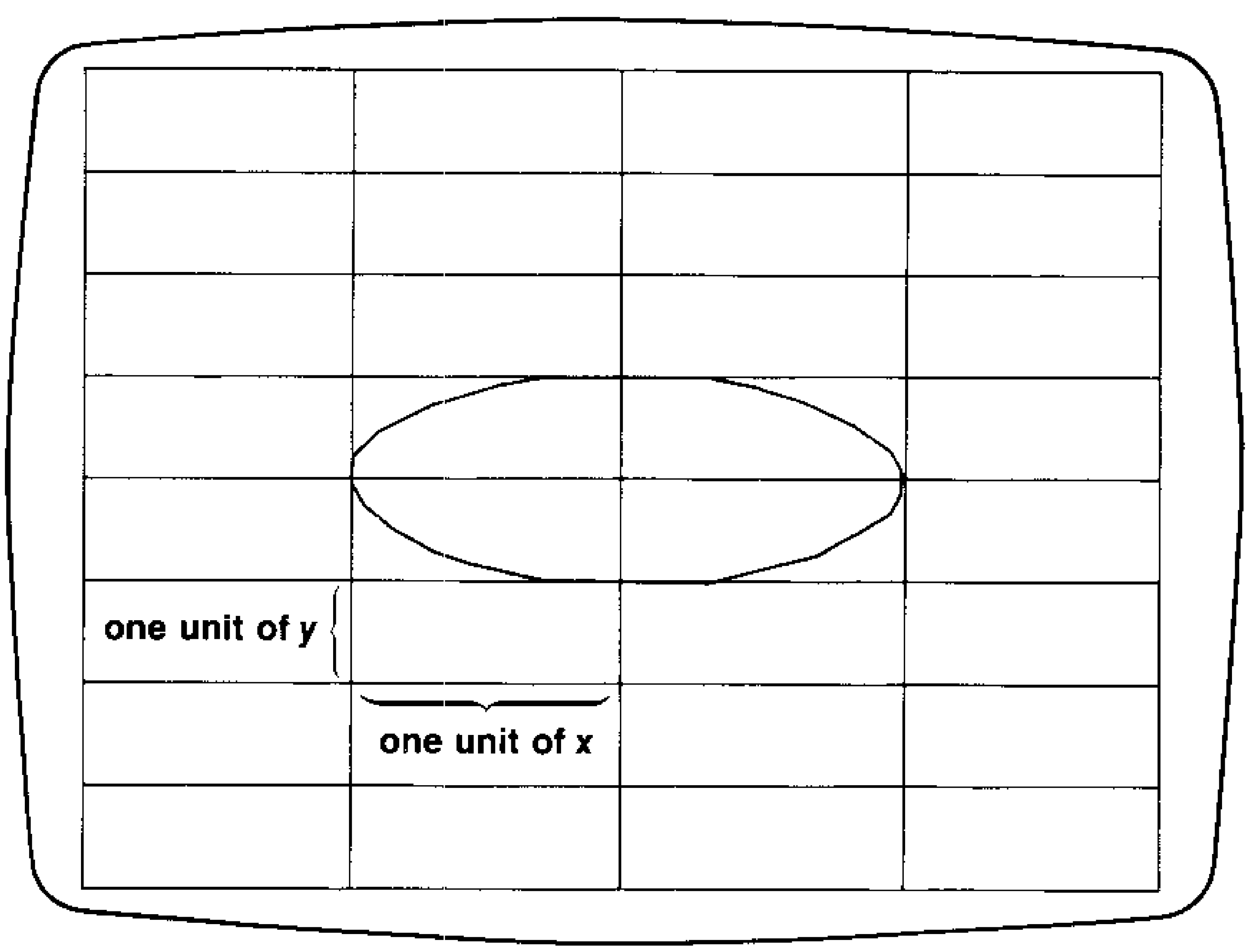
`SCALE 0,10,0,10`	Scales X and Y from 0 to 10.
`SCALE -30,20,-10,20`	Scales the graphics display 50 X-units wide and 30 Y-units high.

Unequal Unit Scaling

The scaling factors for X and Y are completely independent of each other. Therefore, plots are stretched or shrunk independently in the X and Y directions to fit the graphics display area *(anisotropic scaling)*. An X-unit of measure may not necessarily equal a Y-unit of measure.

Example: This program demonstrates the effects of the `SCALE` statement and unequal unit scaling on the plotting area. Note that the length of a unit-of-measure in the X and Y direction are not necessarily equal.

```
  10 GCLEAR                    Clears the graphics display.
  20 DEG                       Sets degree mode.
• 30 SCALE -2,2,-4,4           Specifies X and Y units-of-measure.
  32 ! DRAW A CIRCLE
  40 MOVE 1,0                  Moves to start of circle.
  50 FOR A=0 TO 360 STEP 15    FOR-NEXT loop to specify angle
  60 DRAW COS(A),SIN(A)        measures of circle.
  70 NEXT A
  80 END
```



Because of unequal unit scaling, our "circle" is shaped like an oval; one unit of X does not equal one unit of Y.

Equal Unit Scaling

Particularly with symmetrical plots and curves, it is important to scale the display proportionately in the X and Y directions so that one length of measure in the X direction will equal one length of measure in the Y direction *(isotropic scaling)*.

Since there are 256 useable dots in the horizontal direction and 192 useable dots in the vertical direction (a ratio of four X dots to three Y dots), scale the display so that the number of dots in a unit length of X is equal to the number of dots in a unit length of Y.

The actual ratio of intervals between dots on the display is 255 to 191. But, in most instances, you can use the following equation to determine the number of units in the X and Y directions for equal scaling:

$$X = {}^4/_3 Y$$

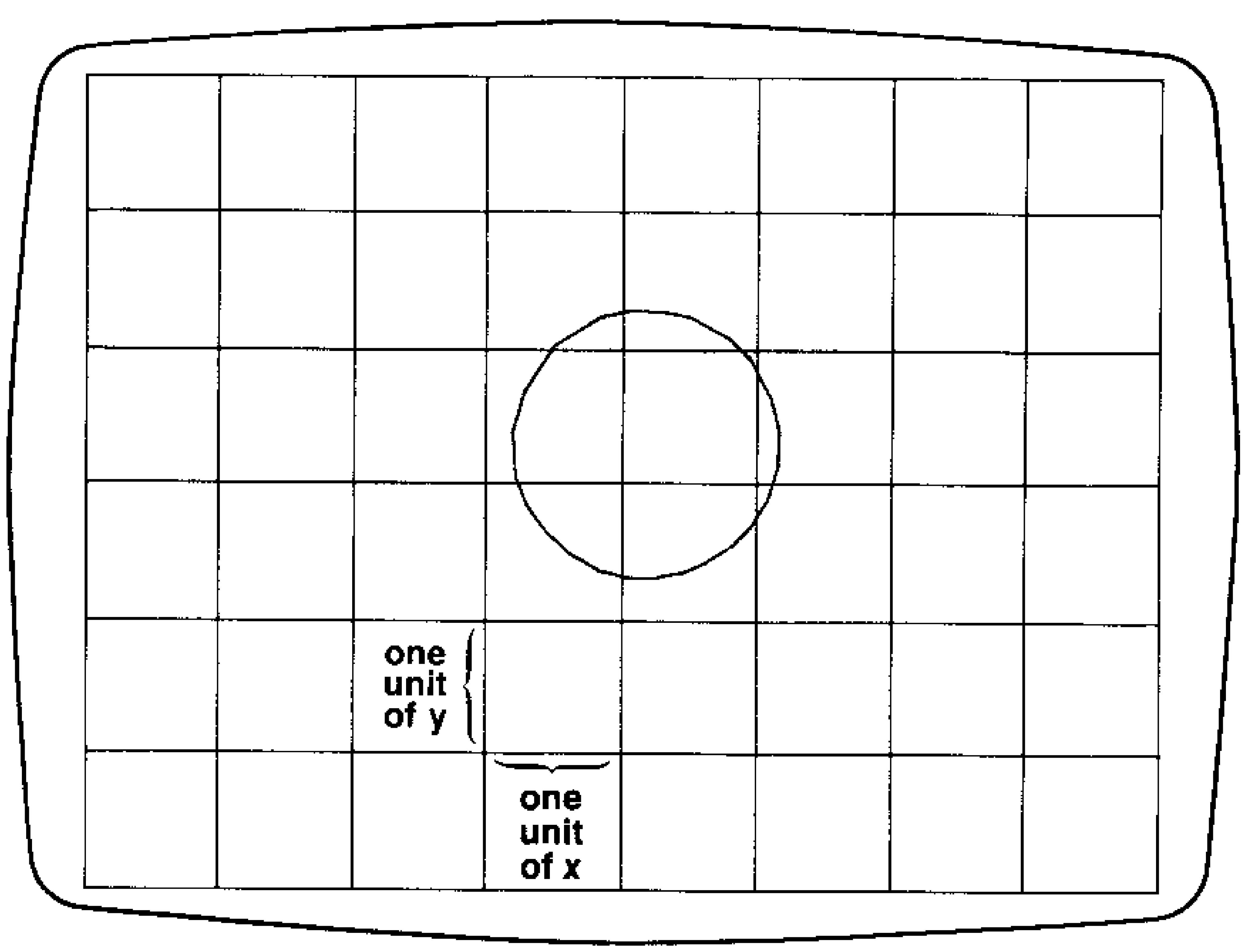
where: X is the number of units in the horizontal direction and
Y is the number of units in the vertical direction.

Example: Modify the `SCALE` statement from the last example so that the circle is drawn in correct proportions. One solution is to change statement 30 to read:

```
30 SCALE -4,4,-3,3
```

Scales 8 X-units by 6 Y-units; ratio of 4X to 3Y. Yields 32 dots per unit length of X and Y.

If you now run the modified program to generate a circle, the following will appear on the display:



As long as the number of dots per unit length of X is equal to the number of dots per unit length of Y, your plots will be drawn symmetrically in both X and Y directions.

More Examples of "Isotropic" Scaling:

```
SCALE 0,4,0,3
```
Scales 4 X-units by 3 Y-units; 64 dots per unit length.

```
SCALE -6,10,-3,9
```
Scales 16 X-units by 12 Y-units; 16 dots per unit length.

```
SCALE -5,55,-5,40
```
Scales 60 X-units by 45 Y-units; 4.262 dots per unit length.

Note: The exception to our rule of scaling $X = {}^{4}/_{3}Y$ is if you scale graphics display to the number of dots on the graphics screen. Then you should use the actual ratio of intervals, $X = {}^{255}/_{191}Y$.

```
SCALE 0,255,0,191
```
Scales 255 X-units by 191 Y-units (256 X dots by 192 Y dots).

or

```
SCALE 1,256,1,192
```

Both statements scale the graphics display so that one unit length is equal to the distance between two adjacent dots.

Drawing Coordinate Axes

The `XAXIS` and `YAXIS` statements draw an X-axis and a Y-axis, respectively, on the graphics display, with optional tic marks.

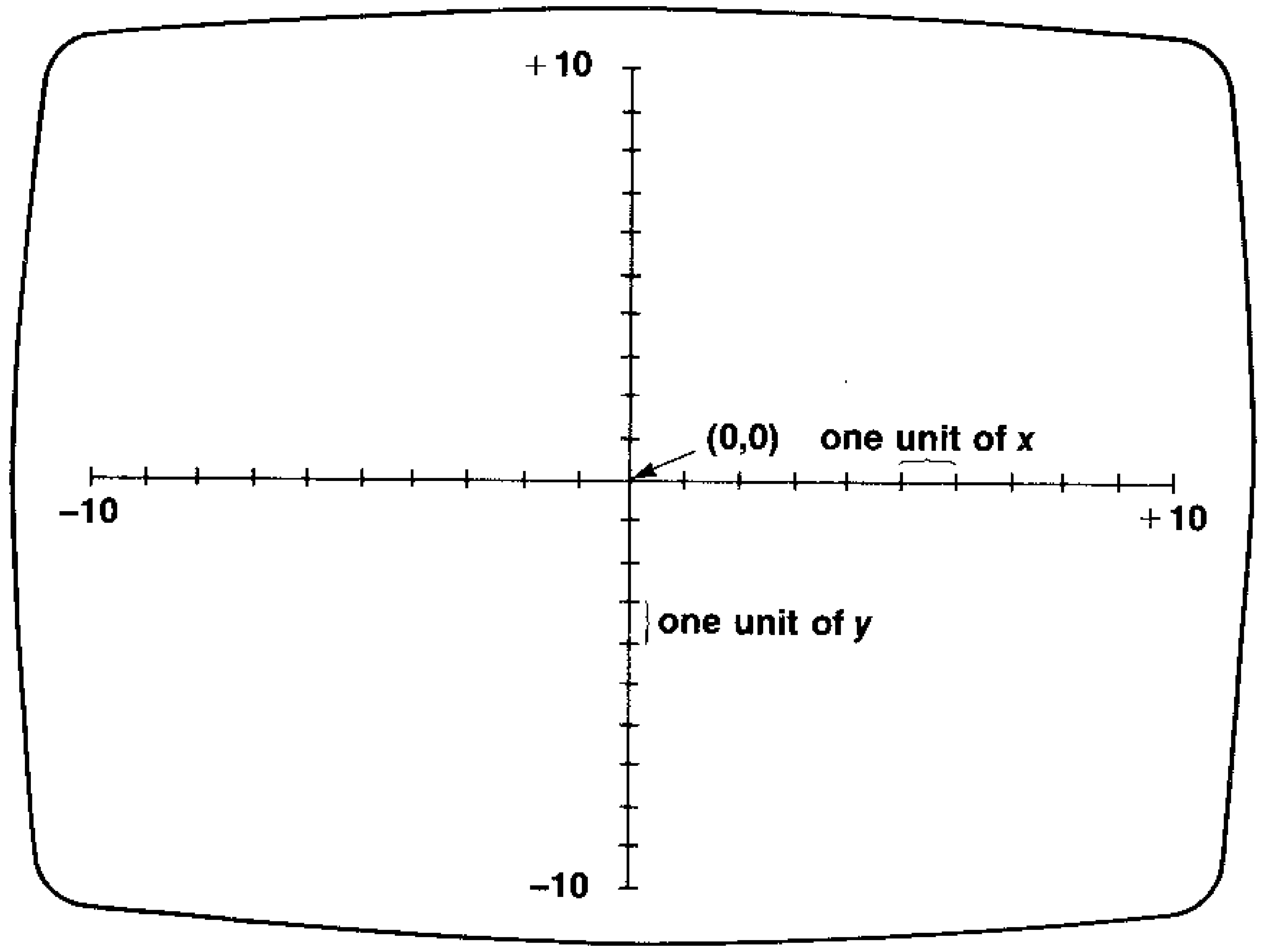
`XAXIS` Y-intercept[, tic spacing[, Xminimum , Xmaximum]]
`YAXIS` X-intercept[, tic spacing[, Yminimum , Ymaximum]]

The `XAXIS` statement generates an X-axis at the specified Y-intercept value on the display. The `YAXIS` statement generates a Y-axis at the specified X-intercept value on the display. An intercept value must be specified with an axis statement; the remaining parameters are optional.

The X and Y tic-spacing parameters are interpreted in the current scaled units. The sign of the tic-spacing parameter determines whether the tics will be drawn in increasing magnitude (positive) or decreasing magnitude (negative). For example, a negative tic parameter in an `XAXIS` statement means that tics will be drawn from right to left in the intervals specified.

Example: The following program first scales the display to be 20 X-units wide (from −10 to +10) and 20 Y-units long (from −10 to +10), then draws a pair of axes with tic marks at each scaled unit on the axes.

```
10 GCLEAR                   Clears the graphics display.
20 SCALE -10,10,-10,10      Scales the graphics display.
30 XAXIS 0,1                Draws an X-axis at Y-intercept 0, and
                            marks one tic every X-unit.
40 YAXIS 0,1                Then draws a Y-axis at X-intercept 0, and
                            marks one tic every Y-unit.
50 COPY
60 END
```

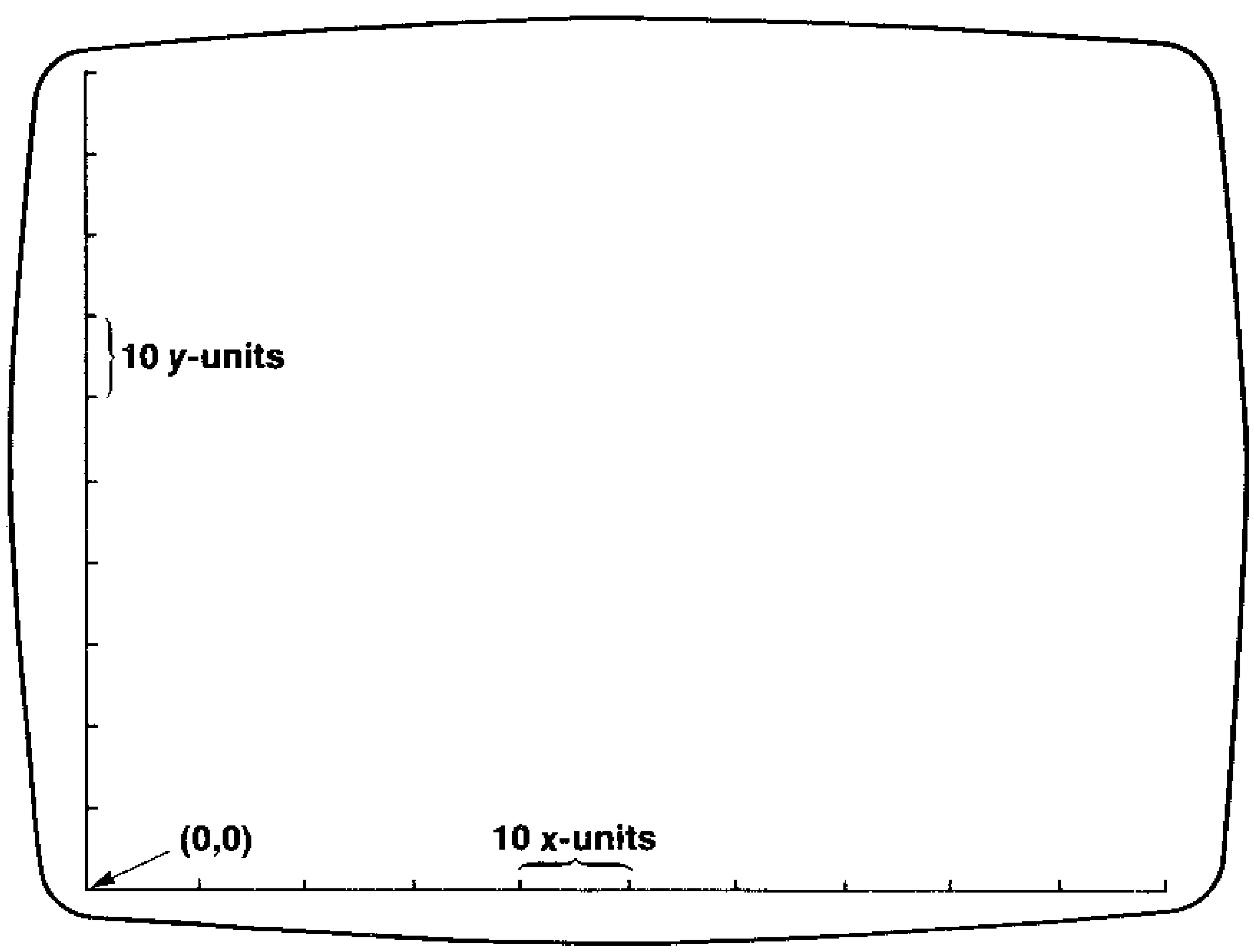


When the axes lie on the boundaries of the graphics display area, only half of each tic mark is shown; for example:

```
 10 GCLEAR
 20 SCALE 0,100,0,100
•30 XAXIS 0,10
•40 YAXIS 0,10
 50 COPY
 60 END
```

This is the default scale at power on or after (RESET).

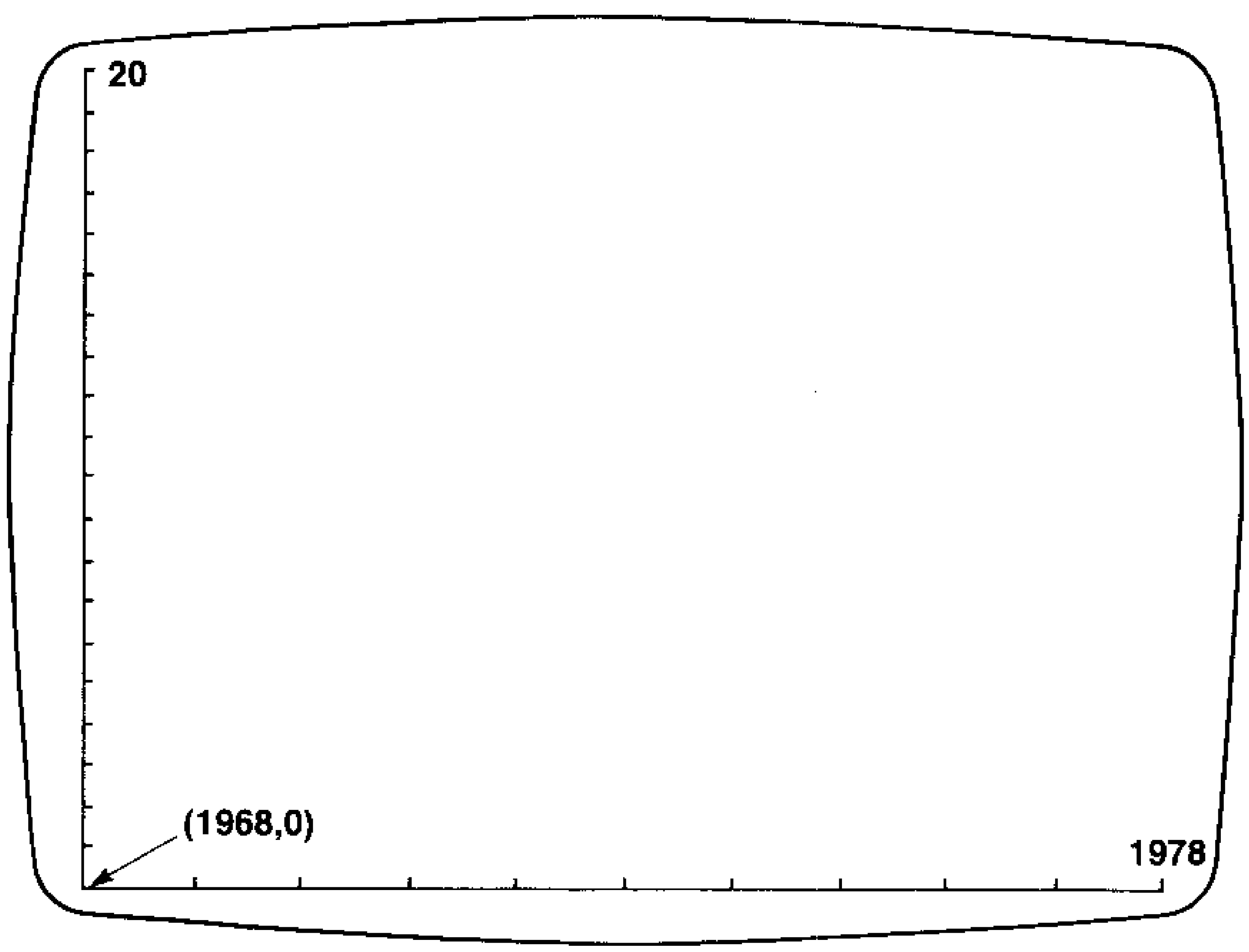
Draws an X-axis, marking tics every 10 units; draws a Y-axis, marking tics every 10 units, then copies the display onto paper.



To draw axes for the weather station graph, you might execute the following statements:

```
 10 GCLEAR
 20 SCALE 1968,1978,0,20
•30 XAXIS 0,1
•40 YAXIS 1968,1
 50 END
```

Notice that the origin has been scaled outside of the graphics display area.



A positive tic parameter instructs the system to draw tic-marks, at the specified interval, from left to right on the X-axis and from bottom to top on the Y-axis. In the example above, tics are drawn on the X-axis at 1968, 1969, 1970, ..., 1978. On the Y-axis, tics are drawn at 0, 1, 2, ..., 20.

A negative tic parameter instructs the system to draw tics, at the specified interval, from right to left on the X-axis. If you use negative X- or Y-values, be aware of the sign of the tic parameters so that you space the tics correctly on the axes.

The minimum and maximum parameters specify the length of the axes within the current scale of the display. These parameters are especially useful when you want to allow space on the display for labels.

The following program illustrates the use of negative tic marks and maximum/minimum X-axis and Y-axis specifications:

```
 10 GCLEAR
 20 SCALE -10,2,-10,2
```

Scales the display at −10 to 2 from left to right, and −10 to 2 from bottom to top.

```
•30 XAXIS 0,-1,-8.5,0
```

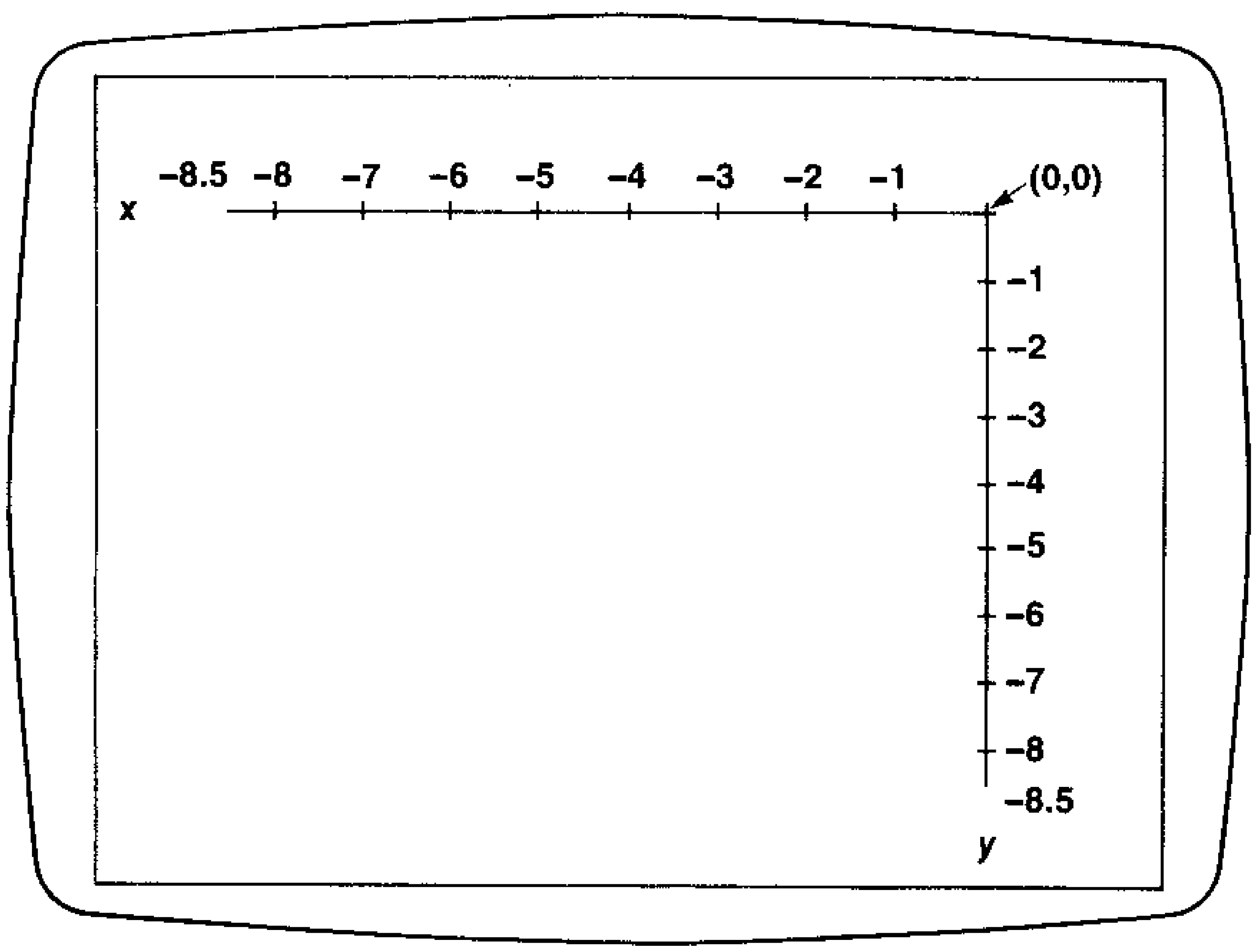
Draws an X-axis at Y=0. Marks one tic for each X-unit from the right side of the axis to the left. Displays only that portion of the X-axis from −8.5 to 0.

```
•40 YAXIS 0,-1,-8.5,0
```

Draws a Y-axis at X=0. Marks one tic for each Y-unit from the top of the axis to the bottom. Displays the Y-axis from −8.5 to 0.

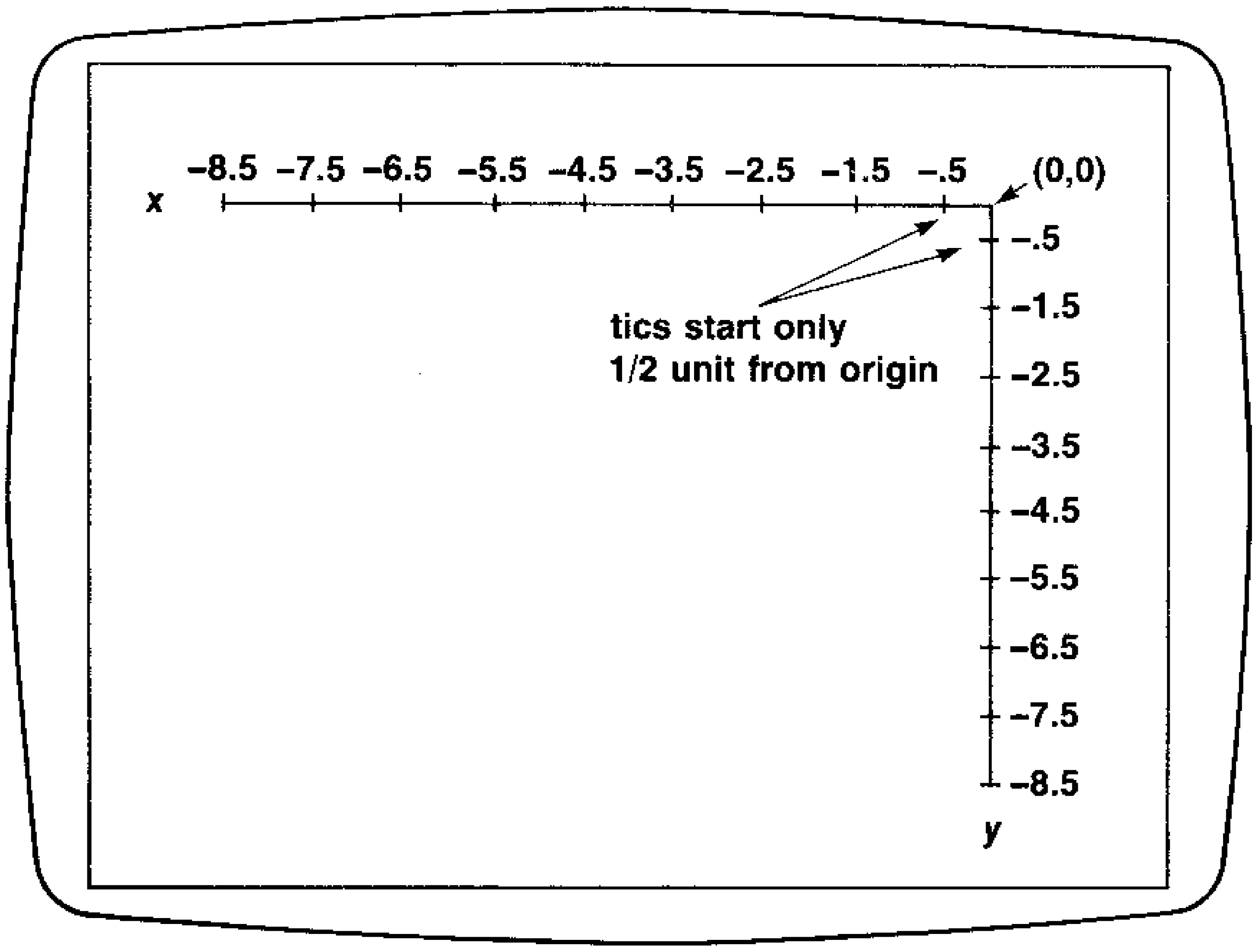
```
 50 XAXIS 2 @ XAXIS -10
 60 YAXIS -10 @ YAXIS 2
 70 END
```

Frames the display with a line at each of the graphics display boundaries.



If our program had been the following, using positive tic parameters, the tics would have been spaced incorrectly as shown:

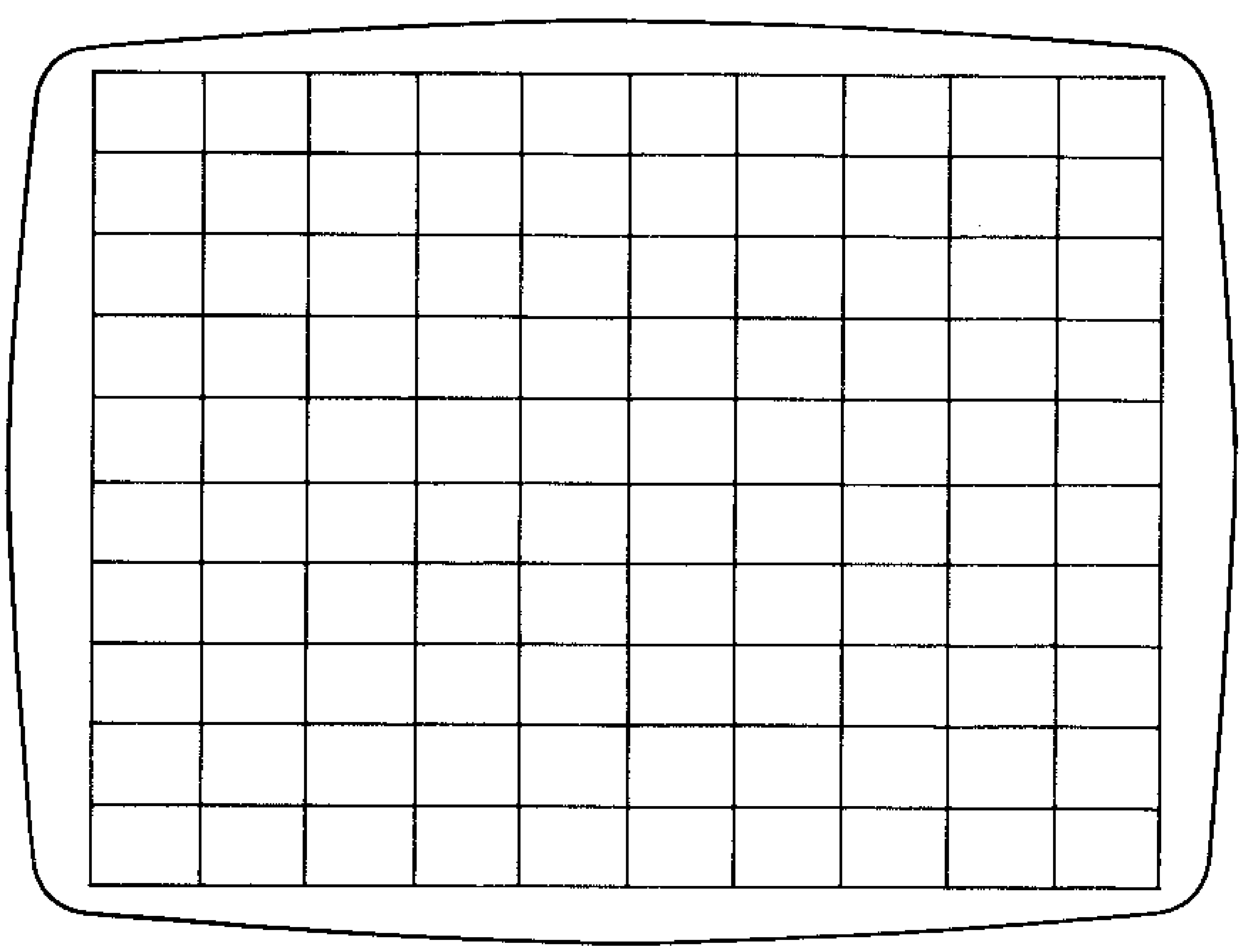
```
 10 GCLEAR
 20 SCALE -10,2,-10,2
•30 XAXIS 0,1,-8.5,0
•40 YAXIS 0,1,-8.5,0
 50 XAXIS 2 @ XAXIS -10
 60 YAXIS -10 @ YAXIS 2
 70 END
```



As you have seen from our examples of framing the display, the axes statements may be used more than once in a program. In fact, the easiest way to draw a vertical or horizontal line is to use an axis statement specifying the X or Y position on the display.

For example, you might use the following program to draw and copy a grid of 10 X-units wide and 10 Y-units long.

```
10 GCLEAR
20 SCALE 0,10,0,10
30 FOR I=0 TO 10
40 XAXIS I @ YAXIS I
50 NEXT I
60 COPY
70 END
```



Plotting Operations

In the following pages, we present graphics statements that enable you to control the pen's movement and "color" in order to produce lines for graphic display.

PENUP

The PENUP statement lifts the pen so that you can move the pen without generating a line on the graphics display. Regardless of whether PENUP is executed from the keyboard or in a program, the statement's form is simply:

PENUP	Raises the pen; stops line generation.

The pen up or pen down status can be automatically controlled by using the DRAW, MOVE, IDRAW, and IMOVE statements.

PEN

The PEN statement specifies whether plotting is done with white dots or black dots. Thus, PEN enables you to draw lines and then erase them. The syntax for the PEN statement is:

PEN numeric expression

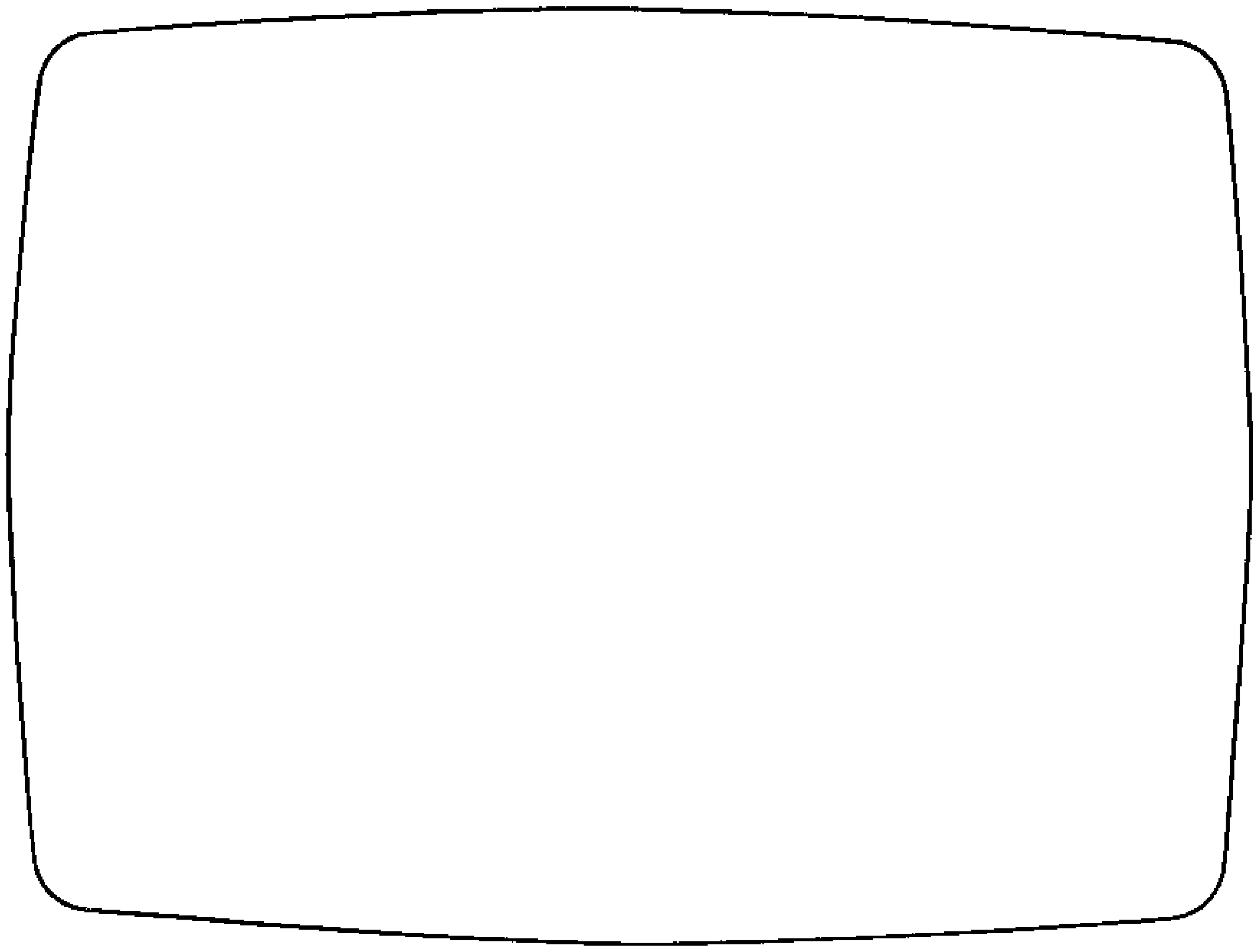
If the numeric expression is positive or zero, white dots are specified for plotting, black dots for clearing. If the numeric expression is negative, black dots are specified for plotting, white dots for clearing. The default pen status at power on or after pressing (RESET) is positive (white dots on black background) and PENUP.

You can think of the pen as a drawing instrument with two colors of ink—black and white—and appropriate erasers for the background color. A positive pen number generates white lines on a black background. A negative pen number selects an "eraser" so that a line redrawn with a negative pen number will be erased with the color of the background. When a line is erased, the intersecting points of any intersecting lines will also be erased.

If you clear the graphics display following the execution of a negative pen number, that portion of the display specified by the GCLEAR statement will be cleared white.

For example, enter and execute the following program:

```
 10 SCALE 0,100,0,100
•20 PEN 1                Sets positive pen.
 30 GCLEAR               Clears the graphics display to black.
•40 PEN -1               Specifies black plotting dots, white
                         clearing dots.
 50 GCLEAR 50            Clears lower half of screen to white.
 60 END
```



PLOT

The PLOT statement makes a dot at the specified X,Y coordinate position *or* draws a line to that position in current units using the current pen number.

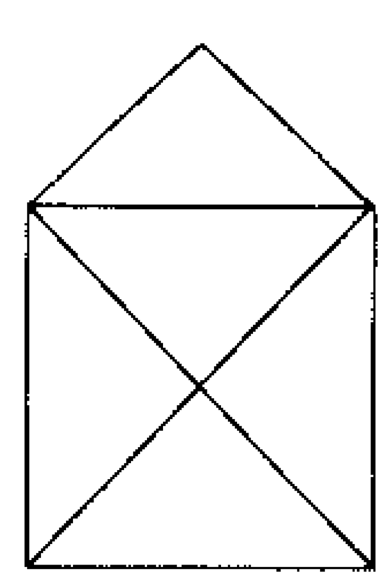
PLOT X-coordinate , Y-coordinate

The X and Y parameters are interpreted according to the current graphics display scale.

If the pen is up when PLOT is executed, the pen moves from the current point to the specified X,Y position, then drops to the screen, makes a dot, and stays down. If the pen is down when PLOT is executed, it stays down and draws a line from the current point to the specified point. If you do not wish to draw a line, the statement preceding PLOT should be a PENUP or MOVE statement.

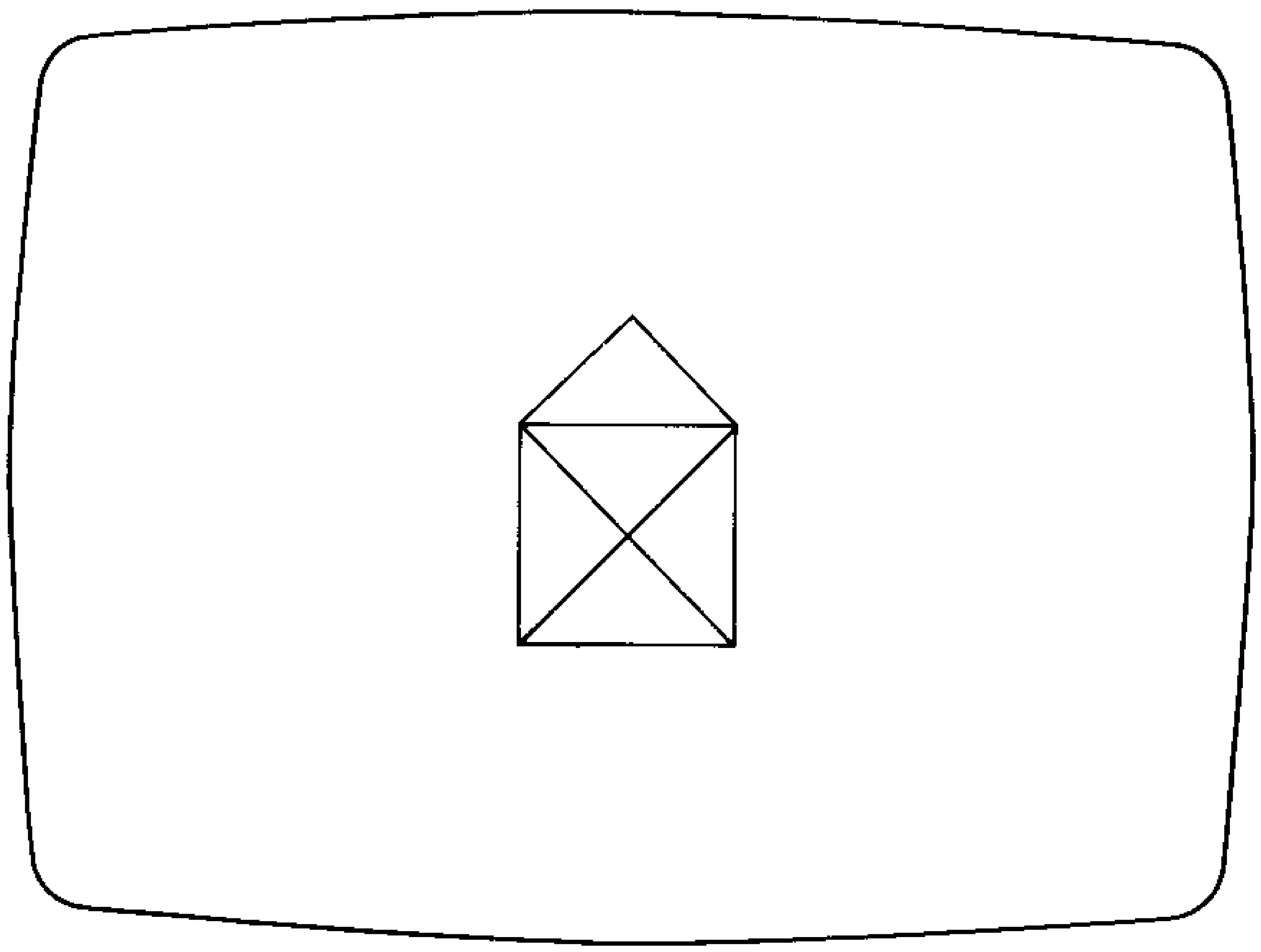
Example: Write a program to draw the figure below with these stipulations:

1. Once the pen is down, you cannot lift it until you have finished drawing the figure.
2. You cannot cross over any line that has been drawn previously.
3. No line can be drawn twice.



Your program might look like this:

`10 PEN 1 @ GCLEAR`	Clears graphics display.
`20 SCALE 0,20,0,15`	Equal unit scale; 20X by 15Y (ratio of 4X to 3Y).
`30 PENUP`	Lifts the pen.
`40 FOR I=1 TO 11`	Start of loop to plot figure.
`50 READ X,Y`	Reads coordinate values.
•`60 PLOT X,Y`	Plots accordingly.
`70 NEXT I`	Reads next values until done.
`80 DATA 8,5,8,9,10,11,12,9,8,9,` `10,7,12,9,12,5,10,7,8,5,12,5`	X,Y coordinate positions for `PLOT`.
`90 END`	

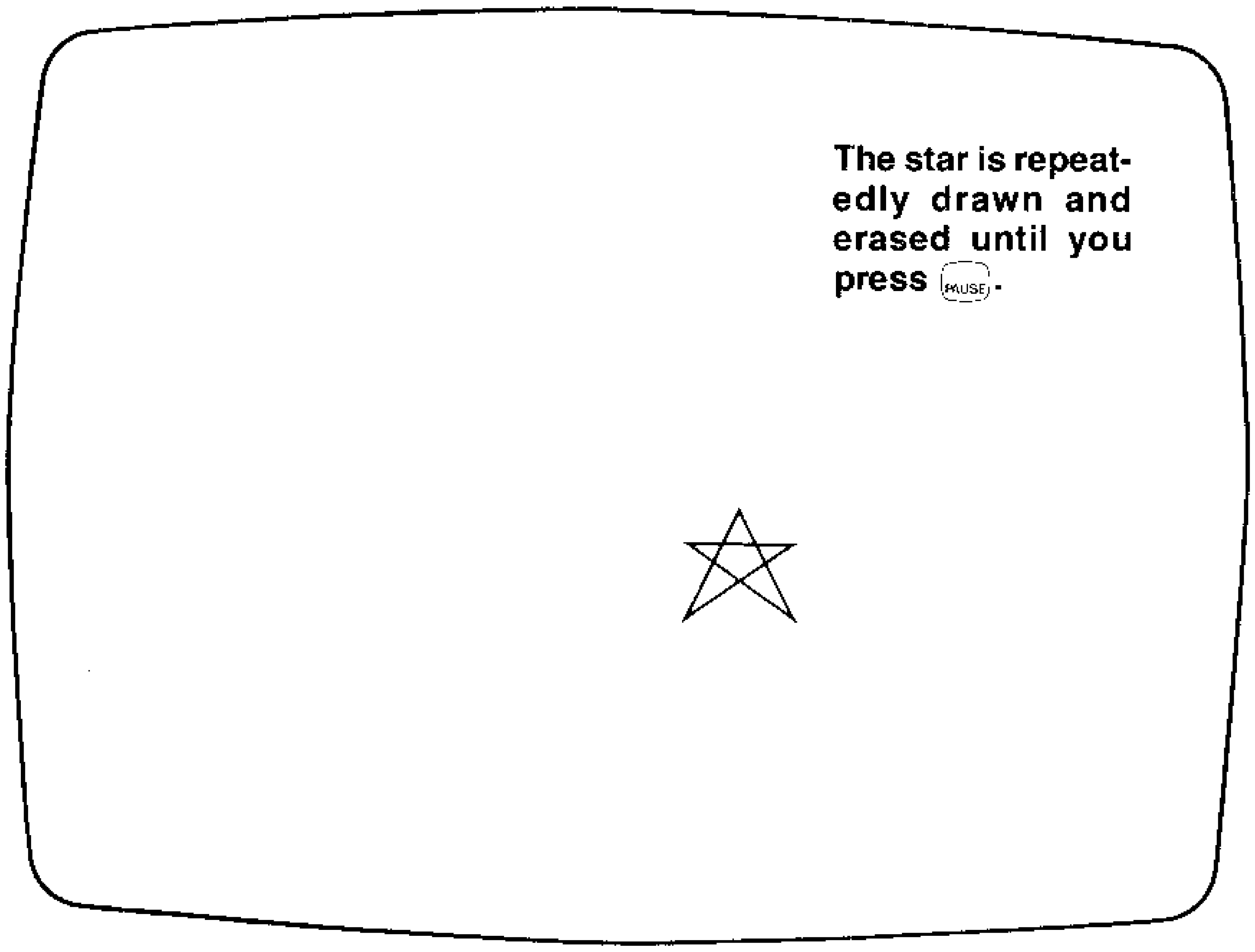


Example: Now write a program to generate a "twinkling" star on the display. First plot the star, then set up a loop to alternately erase it with the opposite pen color and plot it again.

Here's our solution:

```
  10 PEN 1 @ GCLEAR
  20 SCALE -10,10,-5,10
  30 P=1
  40 PENUP
  50 PEN P
  60 FOR I=1 TO 6
  70 READ X,Y
• 80 PLOT X,Y
  90 NEXT I
 100 P=-P
 110 RESTORE
 120 GOTO 50
 130 DATA 0,0,1,2,2,0,0,1.4,2,1.4
     ,0,0
 140 END
```

Press (PAUSE) to stop the program.

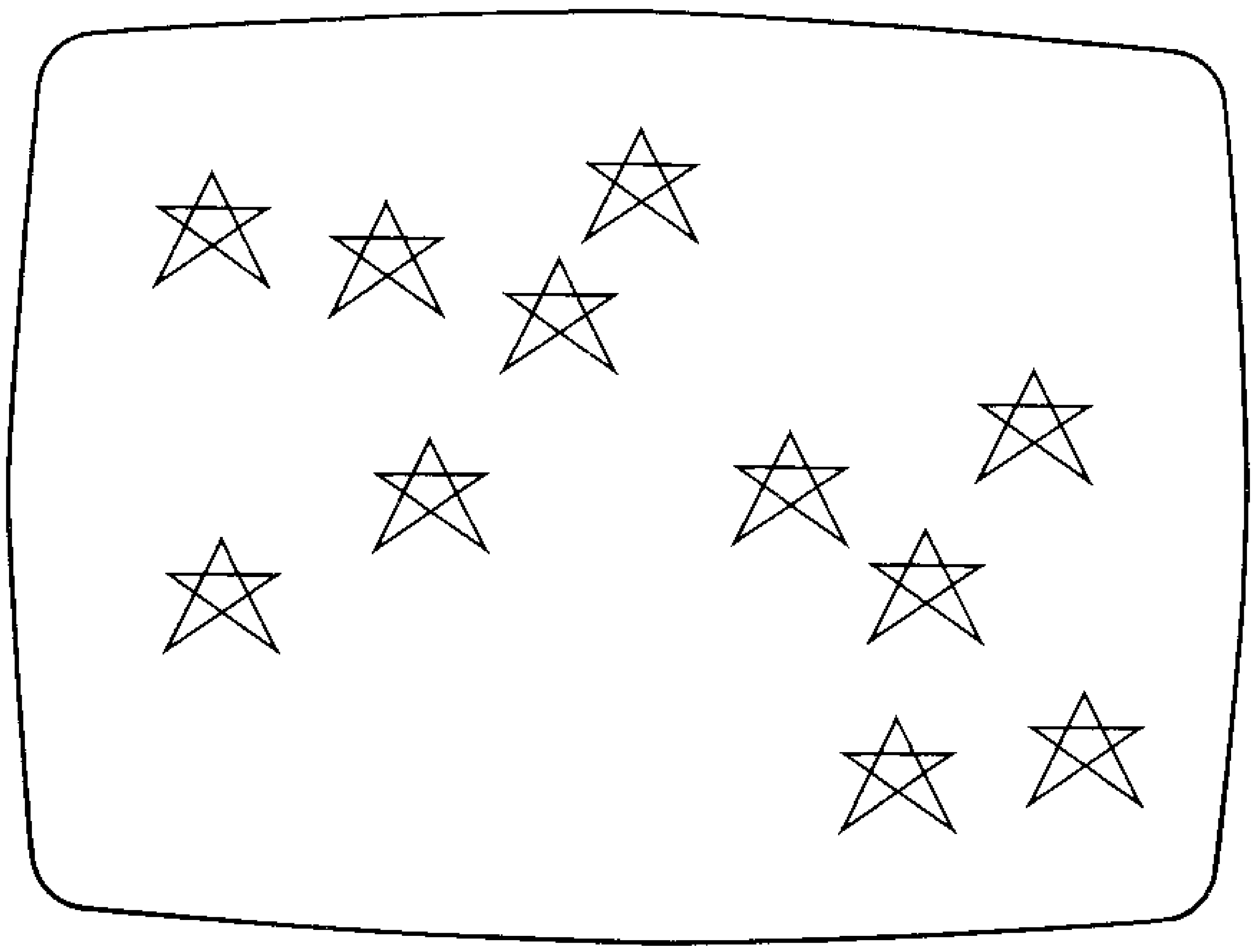


Example: Now that you have star drawing abilities, write a program to generate star clusters. **Hint:** Use the RND function to generate random increments for a given star pattern. In general, you can generate a sequence of random integers from a to b using the following formula: IP((b + 1 −a)*RND+a).

Here's a sample solution:

```
  10 PEN 1 @ GCLEAR
  20 SCALE 0,20,0,15
  30 PENUP
  40 GOSUB 1000
  50 FOR I=1 TO 6
  60 READ X,Y
  70 PLOT X+X1,Y+Y1
  80 NEXT I
  90 RESTORE
 100 GOTO 30
 110 DATA 0,0,1,2,2,0,0,1.4,2,1.4
     ,0,0
1000 X1=18*RND
1010 Y1=13*RND
1020 RETURN
1030 END
```

Each time you run the program, you can generate a different "constellation." Press PAUSE to stop the program, CONT to continue with the current display, and RUN to begin with a clear screen.



Moving and Drawing

The most useful of the graphics statements, DRAW and MOVE, automatically control the pen up or down positions. We will discuss the most efficient way to use them on page 212.

MOVE

The MOVE statement lifts the pen and then moves the pen to the specified X,Y coordinate position in current units and leaves the pen up. This statement provides an easy way of moving the pen without drawing a line on the graphics display, regardless of whether the pen is currently up or down.

MOVE X-coordinate , Y-coordinate

The X and Y parameters are interpreted according to the current scaled units.

Example program lines:

```
30 MOVE 2,5          Moves with pen up to point 2,5.
60 MOVE 25,50        Moves with pen up to 25,50.
```

DRAW

The DRAW statement drops the pen and then draws a line to the specified X,Y coordinate position in current units. This statement provides an easy way of drawing a line from the current pen's location to a new location regardless of whether the pen is currently up or down.

DRAW X-coordinate , Y-coordinate

The X and Y parameters are interpreted according to the current scaled units.

Example program lines:

```
20 DRAW 2,5
70 DRAW 25,50
```

Draws a line from current pen location to point 2,5.
Draws a line from current pen location to point 25,50.

Drawing Curves

The concept of incremental drawing proves extremely useful when implemented to draw curved figures. As you know, you can approximate curves with line segments; many short line segments approximate a curve better than several long line segments.

With HP-85 graphics, you always plot directly from one X,Y coordinate position to another, in this way generating ''lines'' which are actually a series of dots in a straight line. Since you do not have a pen with ink to draw a continuous curve, you must evaluate the equation for a curve in small enough intervals to generate enough ''line segments'' to simulate the curved figure.

This can be done very easily in BASIC programming language with a FOR-NEXT loop using a STEP interval. How small of an interval is small enough? This, of course varies with the curve you wish to display. Generally, 20 to 30 intervals provide enough points to adequately plot a curve.

Earlier in this section, we plotted a circle using a FOR-NEXT loop using a STEP interval. Below, we have rewritten the program for a circle to illustrate our discussion. The first loop computes the step value; in other words, it determines the number of points that will be used to plot the circle. As the step value becomes smaller, the figure displayed makes a closer approximation to a circle.

```
 10 DEG
 20 PEN 1 @ GCLEAR
 30 SCALE -4/3,4/3,-1,1
 40 ! SET INCREMENT VALUE
 50 FOR I=4 TO 30 STEP 2
 60 S=360/I
 70 ! NOW DRAW A CIRCLE
 80 MOVE 1,0
 90 FOR A=0 TO 360 STEP S
100 DRAW COS(A),SIN(A)
110 NEXT A
120 MOVE 0,0
130 LABEL "I="&VAL$(I)
140 WAIT 3000
150 PEN -1
160 MOVE 0,0
170 LABEL "I="&VAL$(I)
180 PEN 1
190 NEXT I
200 END
```

Line 30: Scale changed to draw a larger circle.

Line 50: Sets the step increment value.

Line 70: Draws the circle; plots as many points as STEP will allow.

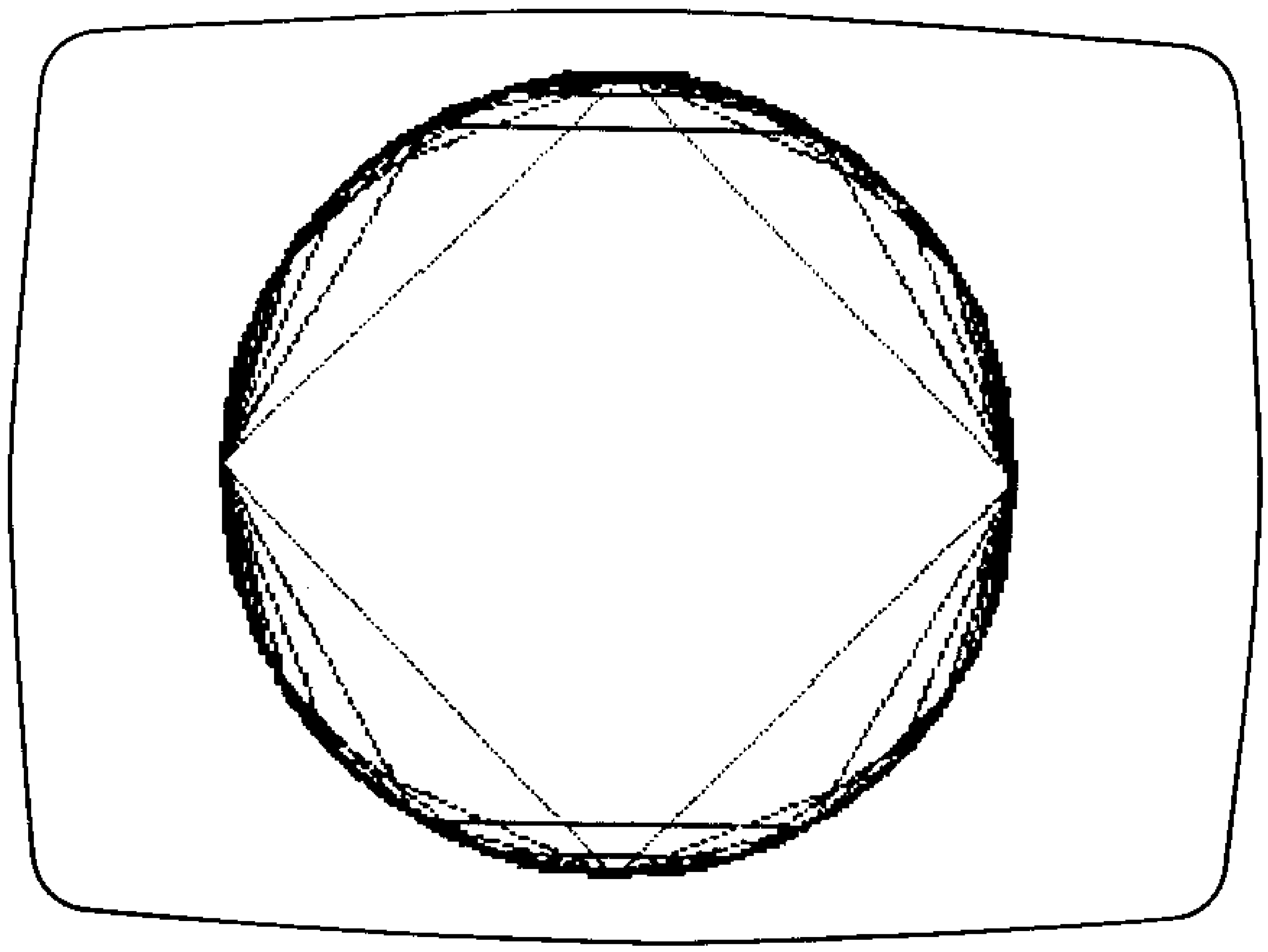
Line 130: We'll discuss this statement in the next section.

Line 140: Displays circle for approximately 3 seconds.

Line 170: Now erases the label by selecting the opposite pen color and relabeling.

Enter the program and press (RUN) to execute it. If you wish to have a printed copy of the figure on the display, just press the (COPY) key to copy the display. Remember you can press (COPY) while a program is running without interrupting program execution.

Below is the plot for I values from 4 to 30 in increments of two.



As you ran the program, you may have noticed that there were very small differences in the circle between values of I from 24 to 30. As you become more familiar with graphing curves on the HP-85 you'll become a better judge of the number of intervals that are necessary to plot a curve.

If you choose an increment value that is too small, it may take the system a long time to plot the graph or curve. You can stop program execution at any time by pressing (RESET), and then edit your increment values if you wish.

Padding the Increment Loop

At this point, we digress a moment from our discussion of the graphics statements to point out an important concept about drawing lines with FOR-NEXT loops. If you thoroughly understand the STEP incrementing process with loops, skip to the problems on page 216.

When a fractional number of increment intervals are specified to complete a graphics figure, it is often necessary to "pad" the final value of the loop counter so that the figure is drawn completely.

Example: The equation for a cardioid is:

$$r = a(1-\cos\theta)$$

where;

> r is the directed distance from the origin to a point on the curve,
> a is any positive constant, and
> θ is the angle measure.

Write a program that plots a cardioid of the form $r=1-\cos\theta$ in radians mode and copies it on the printer.

Suppose your program looked like this:

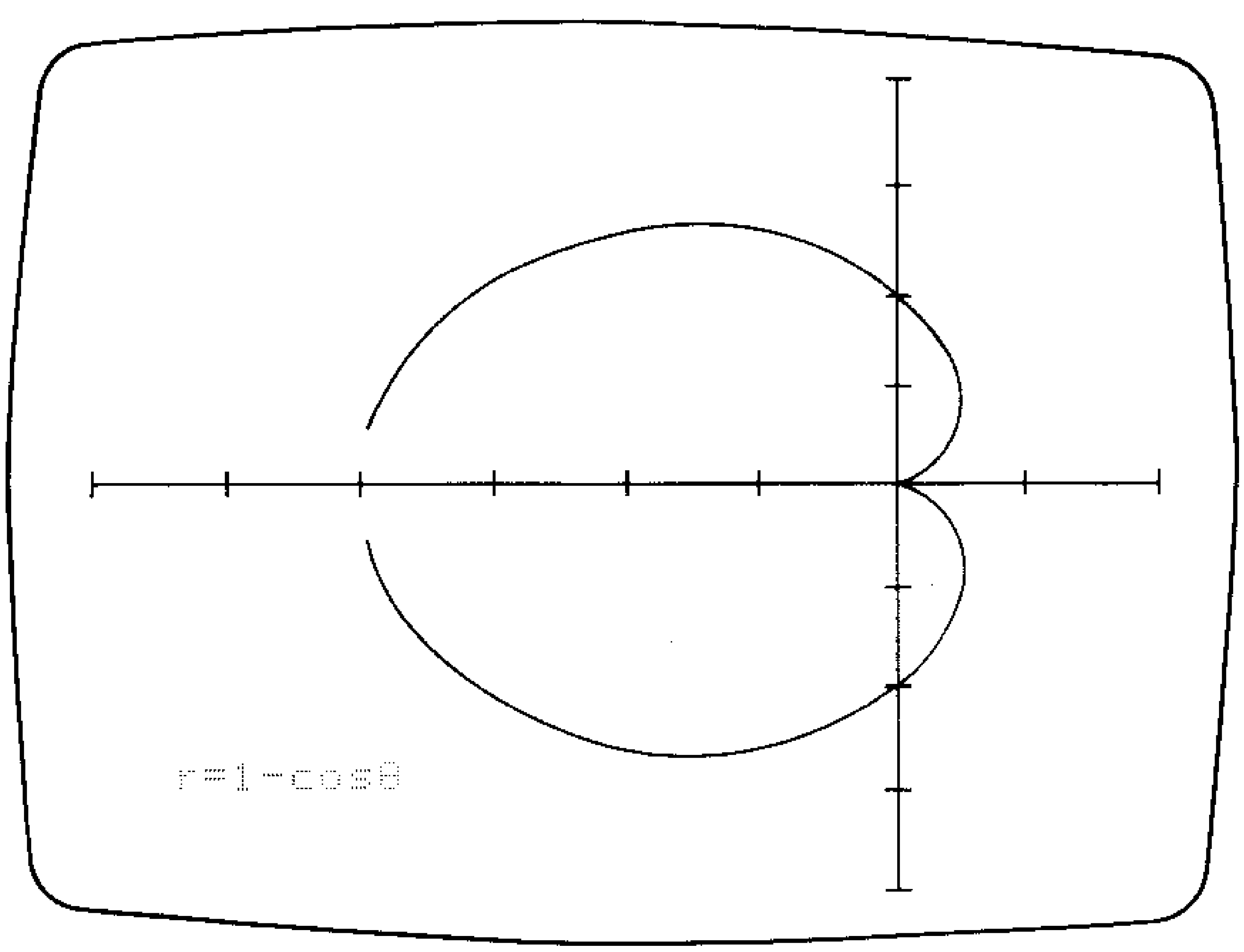
```
   10 PEN 1 @ GCLEAR
   20 SCALE -3,1,-2,2
   30 XAXIS 0,.5
   40 YAXIS 0,.5
   50 RAD
   60 MOVE 0,0
•  70 FOR T=0 TO PI STEP .15
   80 R=1-COS(T)
   90 DRAW R*COS(T),R*SIN(T)
  100 NEXT T
  110 MOVE 0,0
• 120 FOR T=0 TO -PI STEP -.15
  130 R=1-COS(T)
  140 DRAW R*COS(T),R*SIN(T)
  150 NEXT T
  160 MOVE -2.75,-1.5
* 170 LABEL "r=1-cosθ"
  180 END
```

Isotropic scale.
Tic marks every 1/2 unit.

Sets radians.
Moves to point 0,0.
Begins plotting cardioid in increments of 0.15 radians.
Polar/rectangular coordinate conversions.

Move to point 0,0 to plot the other half of the cardioid.

Move to point −2.75,−1.5.
Label graph. (Again, we'll discuss labeling in the next section).



* Type character θ by pressing (CTRL) (P).

As you can see, the figure was not completely drawn. Parts of the curve closest to the value of π were omitted. Examine the `FOR-NEXT` loops:

```
 70 FOR T=0 TO PI STEP .15
.
.
.
100 NEXT T
120 FOR T=0 TO -PI STEP -.15
.
.
.
150 NEXT T
```

This loop is executed at T=0, T=0.15, T=0.3, ..., continuing in increments of 0.15 through T=3. But when T=3+0.15 =3.15, which is greater than the final value of the loop counter (π), the program exits the loop. The fractional part of π is not evaluated.
Similarly, the second loop does not evaluate the portion of the curve closest to $-\pi$ below the X-axis.

In both loops, when the absolute value of T is greater than π, the program exits the loop.

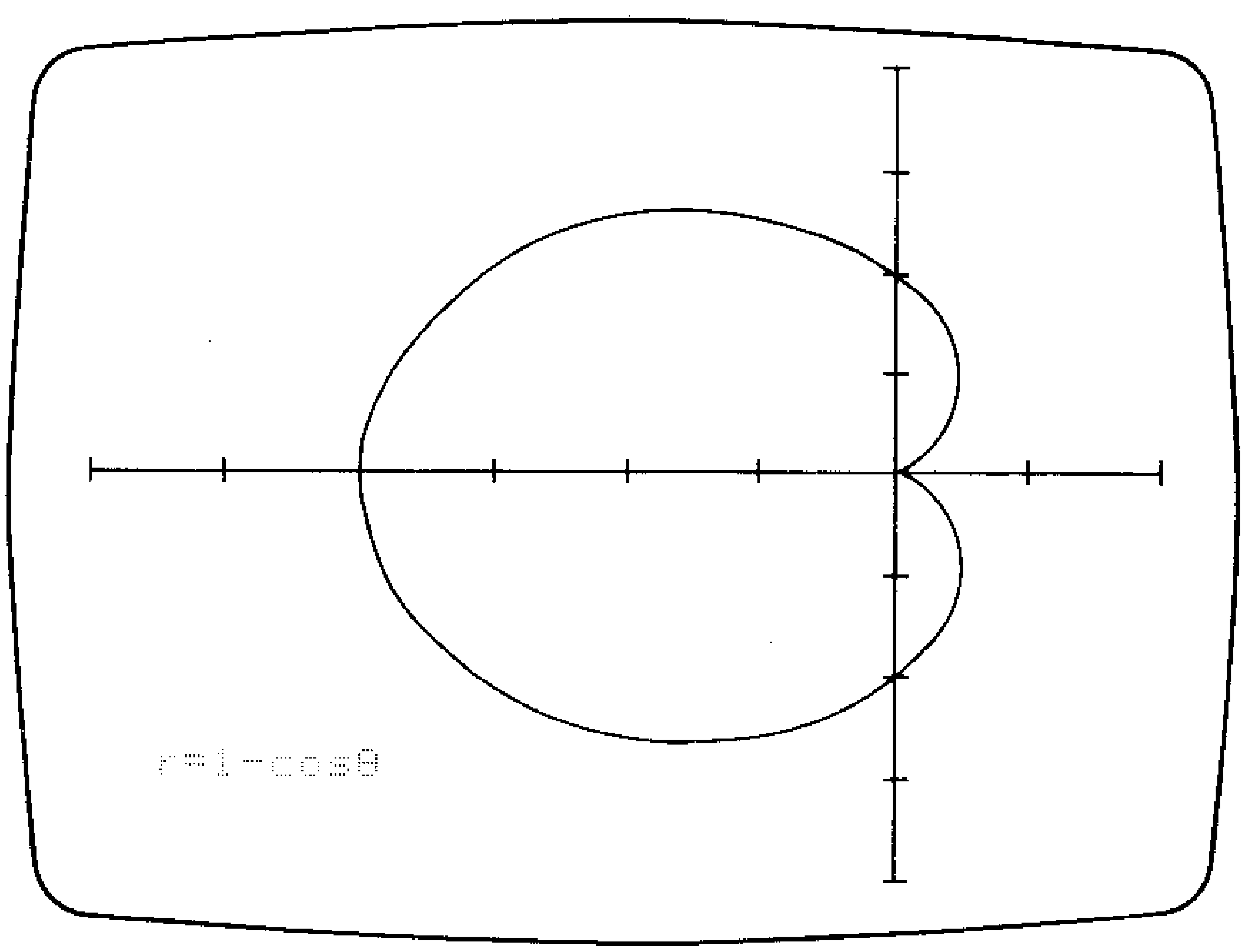
You can correct the effect of the increment value by "padding" the final value of the loop counter. In the cardioid program, extend the final value that follows `TO` in statements 70 and 110 by 0.1, so that they read:

```
• 70 FOR T=0 TO PI+.1 STEP .15
 :
•120 FOR T=0 TO -PI-.1 STEP -.15
```

Add 0.1 to π here.

Subtract 0.1 from $-\pi$ here.

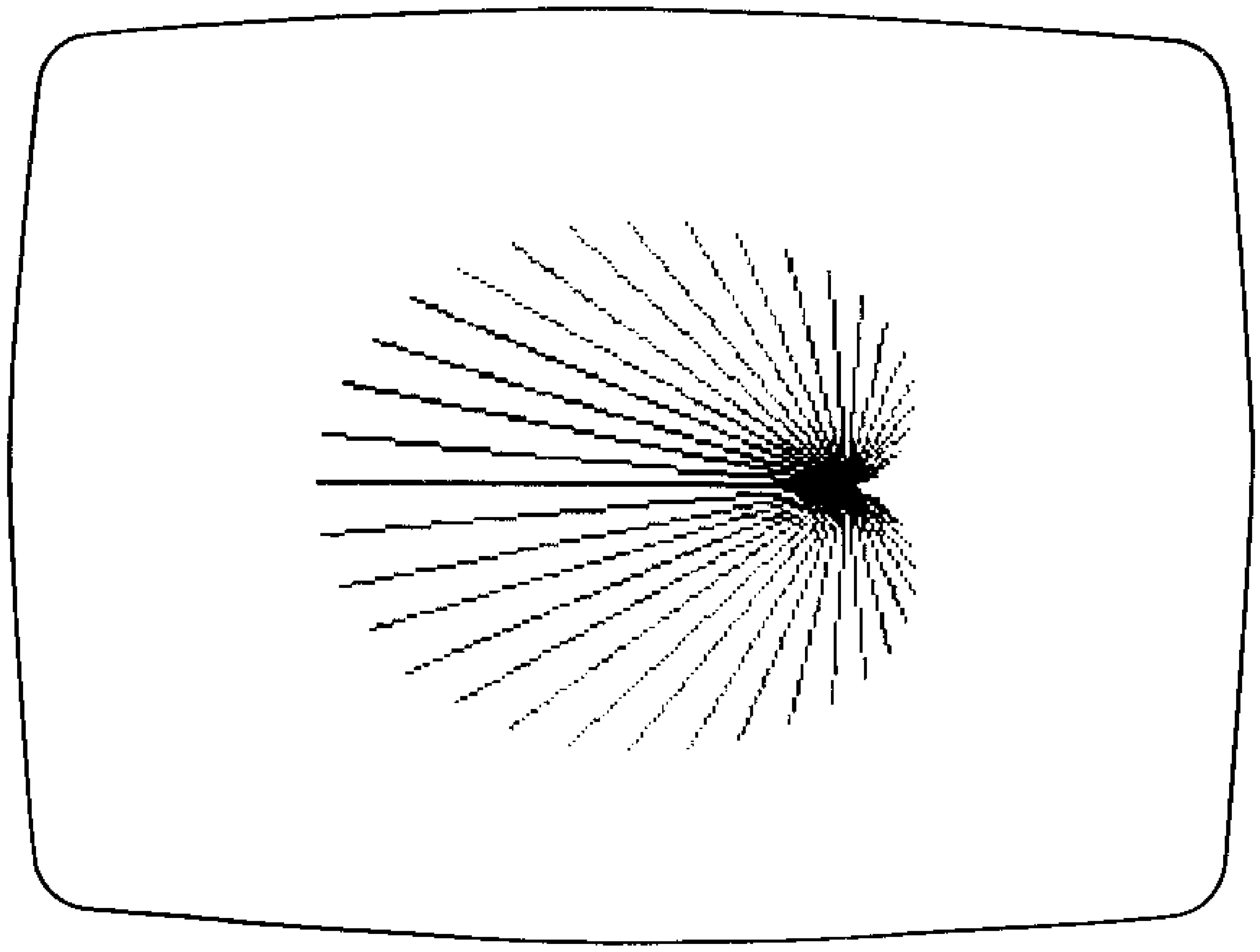
Now the full range of values for each loop will be evaluated. Run the program again to display and copy the completed cardioid.



The first loop is executed at T=3.15 because T is still less than $\pi + 0.1$; the second loop is executed at T=−3.15 because T is greater than $-\pi - 0.1$.

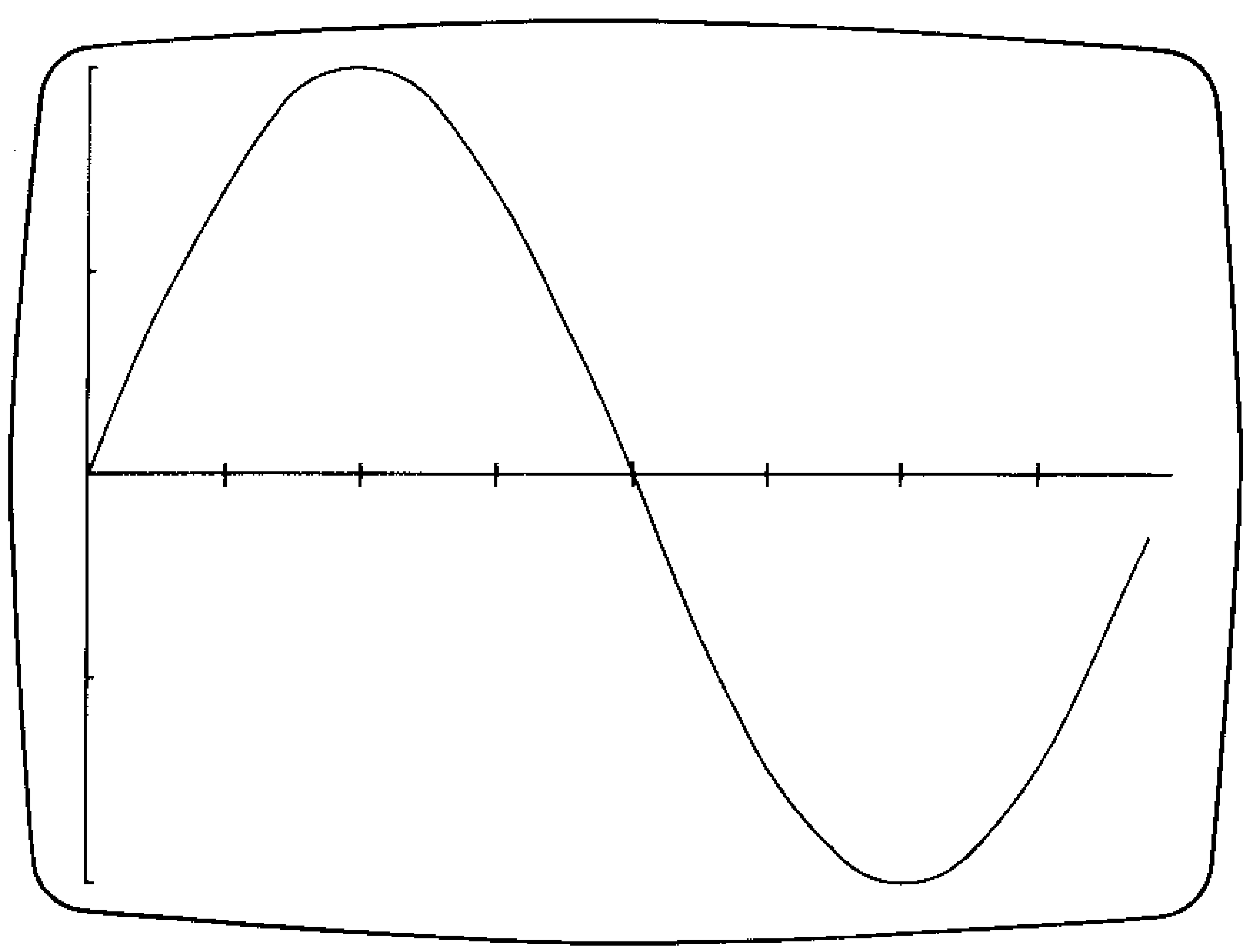
Problems

12.1 Now that you've had some experience in graphing cardioids, write a program to display the following:

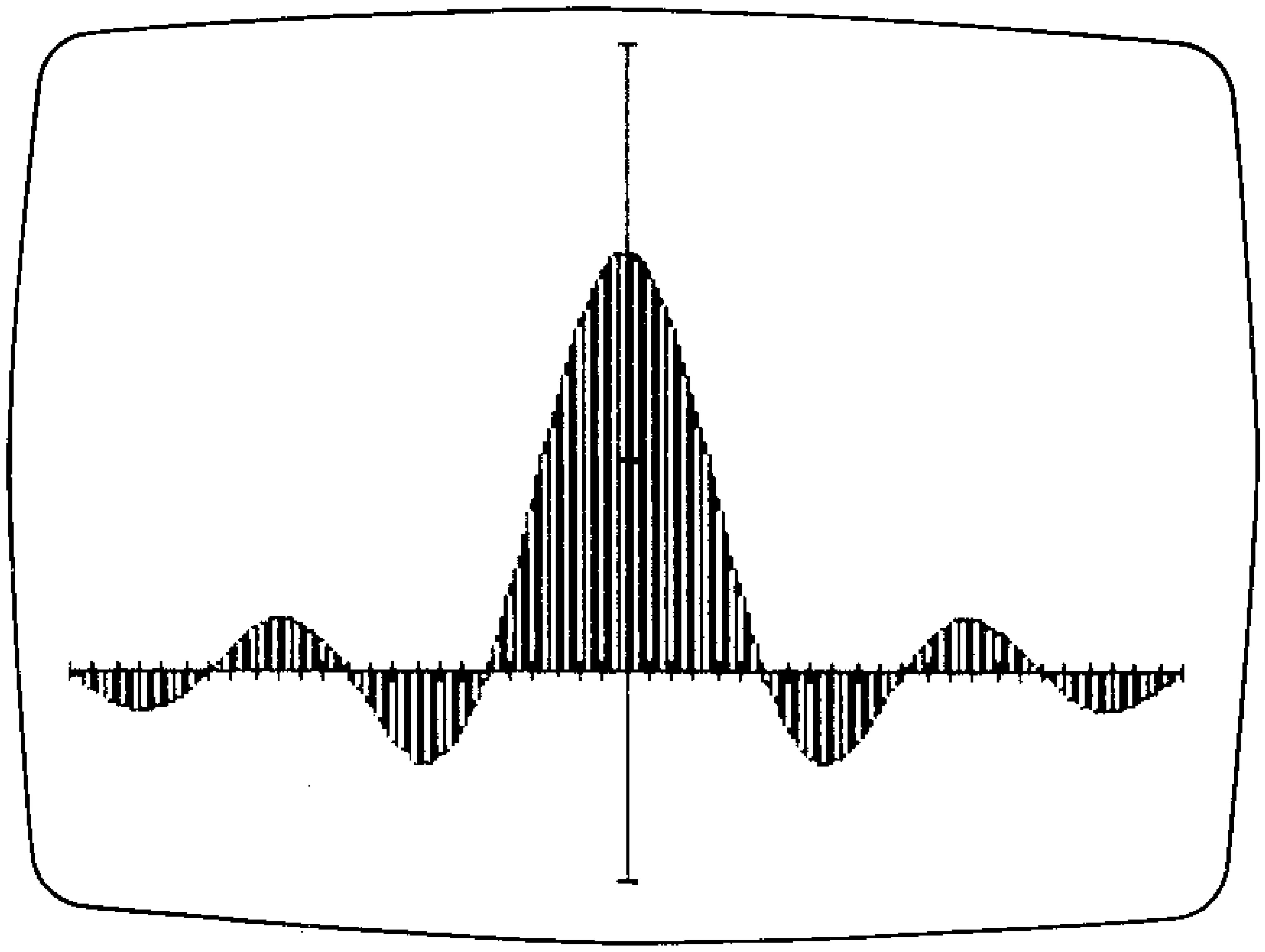


12.2 Pad the FOR-NEXT loop in the following program to complete the sine curve.

```
 10 PEN 1 @ GCLEAR
 20 SCALE 0,2*PI,-1,1
 30 XAXIS 0,PI/4
 40 YAXIS 0,.5
 50 RAD
 60 MOVE 0,0                          Moves to start of curve.
 70 FOR X=0 TO 2*PI STEP PI/20
 80 DRAW X,SIN(X)
 90 NEXT X
100 END
```



12.3 Write a program to generate the curve of the SIN (X)/X from -4π to $+4\pi$. As you plot the curve, also draw "fill" lines from the curve to the X-axis. Be sure to check for X=0, so that you don't divide by zero.



IMOVE

The IMOVE *(incremental move)* statement provides incremental moving capability. The origin is assumed to be the current pen position.

IMOVE X-increment , Y-increment

The IMOVE statement interprets the X and Y parameters according to the current scaled units relative to a local origin. The local origin is that of the pen position before the IMOVE statement is executed (i.e., the current pen position).

Thus, the IMOVE statement moves the pen, without drawing a line, from the current pen position to that position plus or minus the increment in each coordinate value.

Example program statements:

`50 IMOVE 1,3`	Moves the pen from current pen position (say X,Y), one unit to the right and three units up (or, to point X + 1, Y + 3).
`80 IMOVE -5,2`	Moves the pen from current pen position (X,Y), five units to the left and two units up (to point X−5, Y+2).

IDRAW

The IDRAW *(incremental draw)* statement provides incremental drawing capability. The origin is assumed to be the current pen position.

IDRAW X-increment , Y-increment

The IDRAW statement interprets the X and Y parameters according to the current scaled units relative to a local origin. The local origin is that of the pen position before the IDRAW statement is executed (i.e., the current pen position).

Thus, the IDRAW statement draws a line from the current pen position to that position plus or minus the increment in each coordinate value.

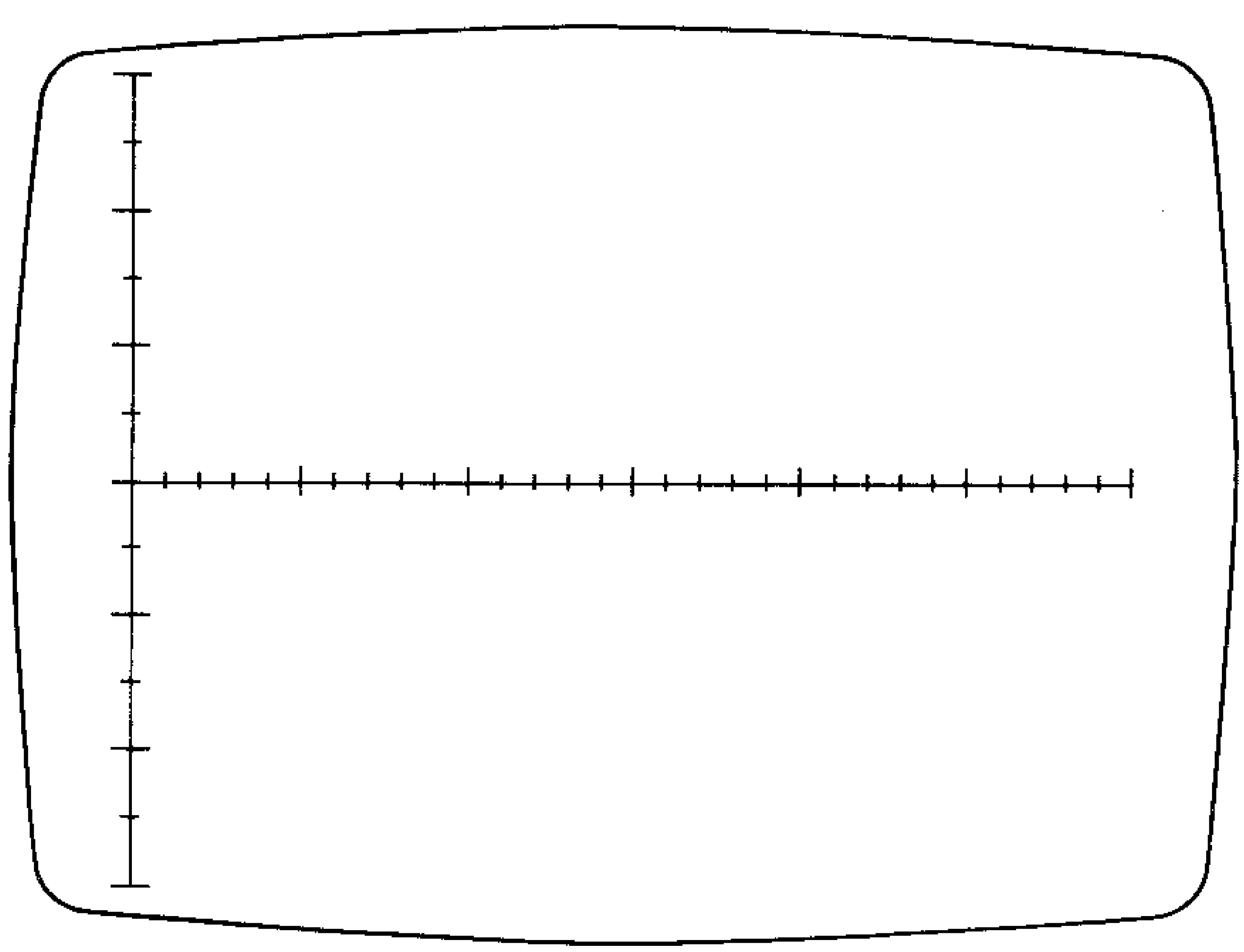
Example program statements:

`30 IDRAW 1,3`	Draws a line to a point one unit to the right and three units up from current pen position.
`60 IDRAW -5,2`	Draws a line to a point five units to the left and two units up from current pen position.

The IMOVE and IDRAW statement are particularly useful for plotting lines or figures of similar slope and size when the exact coordinate positions are unknown.

For instance, suppose you wish to make larger tic marks on the X-axis, every five units, and on the Y-axis, every two units. You might write a program like this.

```
 10 PEN 1 @ GCLEAR
 20 SCALE -2,30,-6,6
 30 XAXIS 0,1,0,30
 40 YAXIS 0,1
 50 FOR X=0 TO 30 STEP 5
 60 MOVE X,.2
•70 IDRAW 0,-.4
 80 NEXT X
 90 FOR Y=-6 TO 6 STEP 2
100 MOVE -.5,Y
•110 IDRAW 1,0
120 NEXT Y
130 END
```

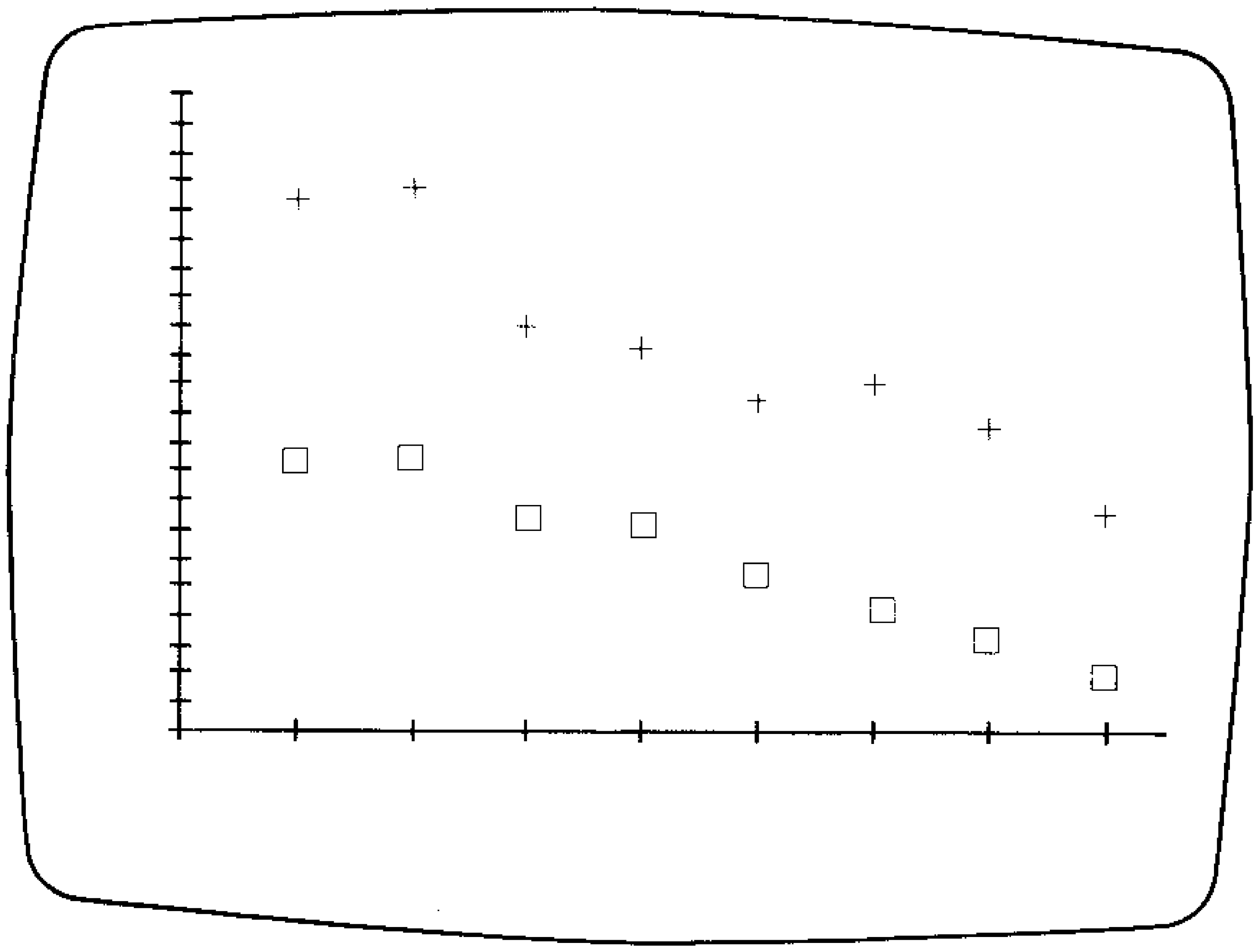


Example: Using the information from the table below, graph the Summer Olympic records for the 100-meter freestyle swimming—men and women—from 1948 to 1976. Instead of plotting a point, make a "+" symbol for each of the women's records and "□" symbol for each of the men's records. (Since we will use this example later in the section, you may wish to store the program on tape after you've entered it into computer memory.)

Summer Olympics Winners and Records: 100-Meter Freestyle

Year	Men	Time (seconds)	Women	Time (seconds)
1948	Ris	57.3	Anderson	66.3
1952	Scholes	57.4	Szoke	66.8
1956	Henricks	55.4	Frazer	62.0
1960	Devitt	55.2	Frazer	61.2
1964	Schollander	53.4	Frazer	59.5
1968	Wenden	52.2	Henne	60.0
1972	Spitz	51.22	Neilson	58.59
1976	Montgomery	49.99	Ender	55.65

`10 REM *100-METER FREESTYLE SUMMER OLYMPICS 1948-1976`	
`20 GCLEAR`	
`30 SCALE 1940,1978,42,70.5`	Equal unit scaling.
`40 XAXIS 48,4,1944,1978`	Displays X-axis from 1944 to 1978 with tics every 4 years.
`50 YAXIS 1944,1,48,70`	Displays Y-axis from 48 to 70 with tics every second.
`60 FOR I=1948 TO 1976 STEP 4`	
`70 READ M`	Reads men's record in year set by loop counter.
`80 GOSUB 1000`	Go plot square.
`90 READ W`	Reads women's record in year set by loop counter.
`100 GOSUB 2000`	Go plot cross.
`110 NEXT I`	
`120 DATA 57.3,66.3,57.4,66.8,55.4,62,55.2,61.2,53.4,59.5,52.2,60,51.22,58.59,49.99,55.65`	
`130 STOP`	
`1000 REM *PLOT SQUARE*`	
`1010 MOVE I,M`	Moves to year, men's record in seconds.
`•1020 IMOVE -.2,.2`	Moves to a point −0.2 left and 0.2 up.
`•1030 IDRAW .4,0 @ IDRAW 0,-.4`	Draws top and right side of square.
`•1040 IDRAW -.4,0 @ IDRAW 0,.4`	Draws bottom and left side of square.
`1050 RETURN`	
`2000 REM *PLOT CROSS*`	
`2010 MOVE I,W`	Moves to year, women's record in seconds.
`•2020 IMOVE 0,.3 @ IDRAW 0,-.6`	Moves 0.3 unit up then draws line down.
`•2030 IMOVE .3,.3 @ IDRAW -.6,0`	Moves to a point 0.3 unit up and 0.3 unit right, then draws line left.
`2040 RETURN`	
`3000 END`	



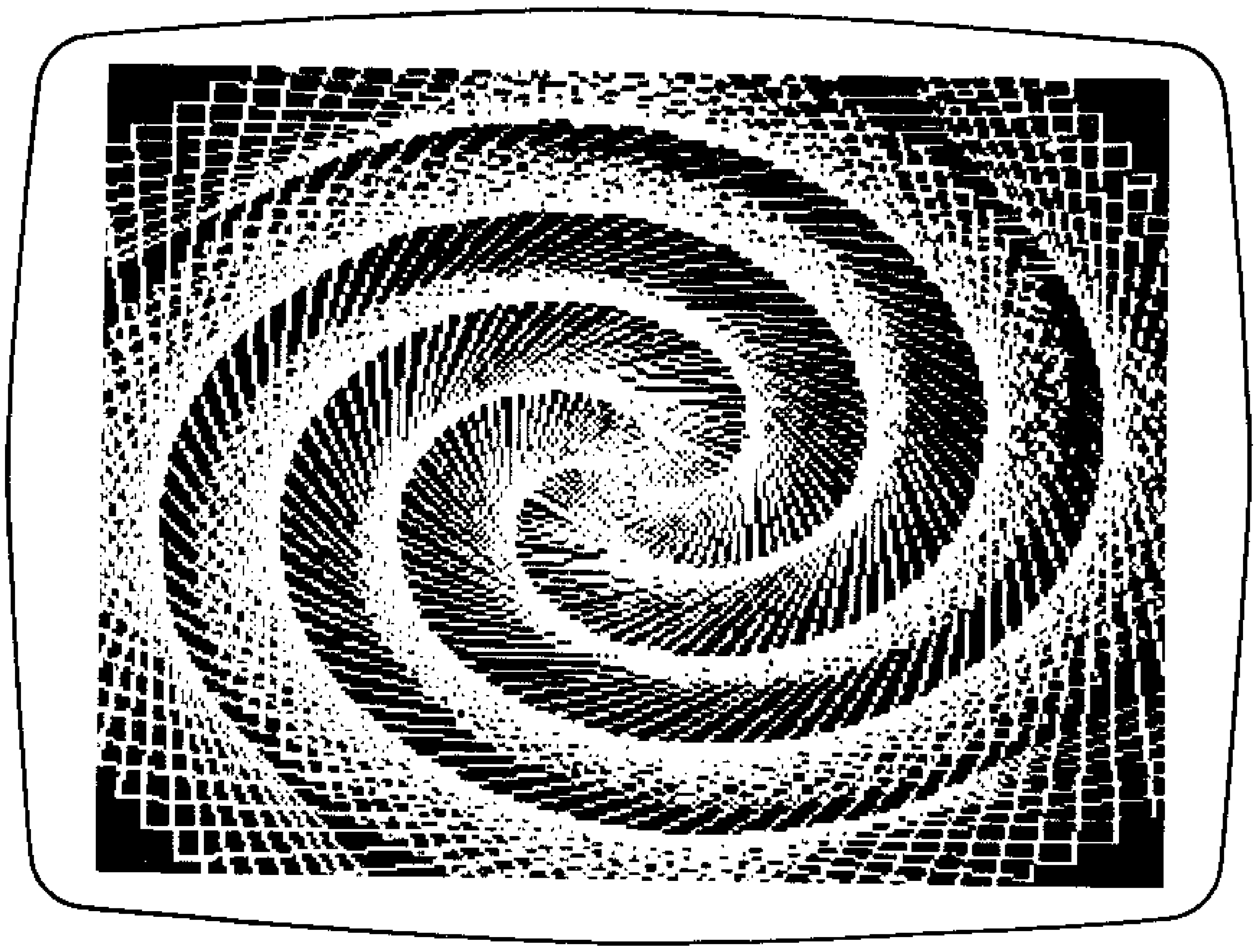
Problem

12.4 The following program, using IDRAW, generates some interesting graphic designs based on the form of a hyperbolic spiral. What angle increments generate the most interesting designs? Why do we include statement 120? How would you modify the program to generate a spiral twice as wide?

```
 10 DEG
 20 CLEAR
 30 DISP "INVERSE VIDEO: YES OR
    NO";
 40 INPUT P$
 50 IF P$="YES" THEN PEN -1 ELSE
    PEN 1
 60 SCALE -36000,36000,-36000,36
    000
 70 DISP "ANGLE INCREMENT VALUE"
    ;
 80 INPUT A
 90 GCLEAR
100 MOVE 0,0
110 FOR I=0 TO 36000 STEP A
120 IF A>90 THEN J=2*I ELSE J=I
130 IDRAW J*COS(I),J*SIN(I)
140 NEXT I
150 END
```

RUN

```
INVERSE VIDEO: YES OR NO?
YES
ANGLE INCREMENT VALUE?
91
```



Labeling Graphs

As you have seen from the "circle approximation" program and the "cardioid" program, you can further enhance the legibility of data plots by labeling graphs using the LABEL statement.

LABEL string expression

Note that only string expressions may be used with the LABEL statement. The string expression may include quoted character strings, string variables, string functions, substrings, and the string concatenator, &. The size of the character(s) specified with the LABEL statement is the same on the graphics display as the alpha display. Examples of LABEL statements that we have used already are:

```
 30 SCALE -4/3,4/3,-1,1
 :
 :
 120 MOVE 0,0
•130 LABEL "I="&VAL$(I)
```

Example from page 212.

Moves to point 0,0.
Then labels.

```
 20 SCALE -3,1,-2,2
 :
 :
 160 MOVE -2.75,-1.5
•170 LABEL "r=1-cosθ"
```

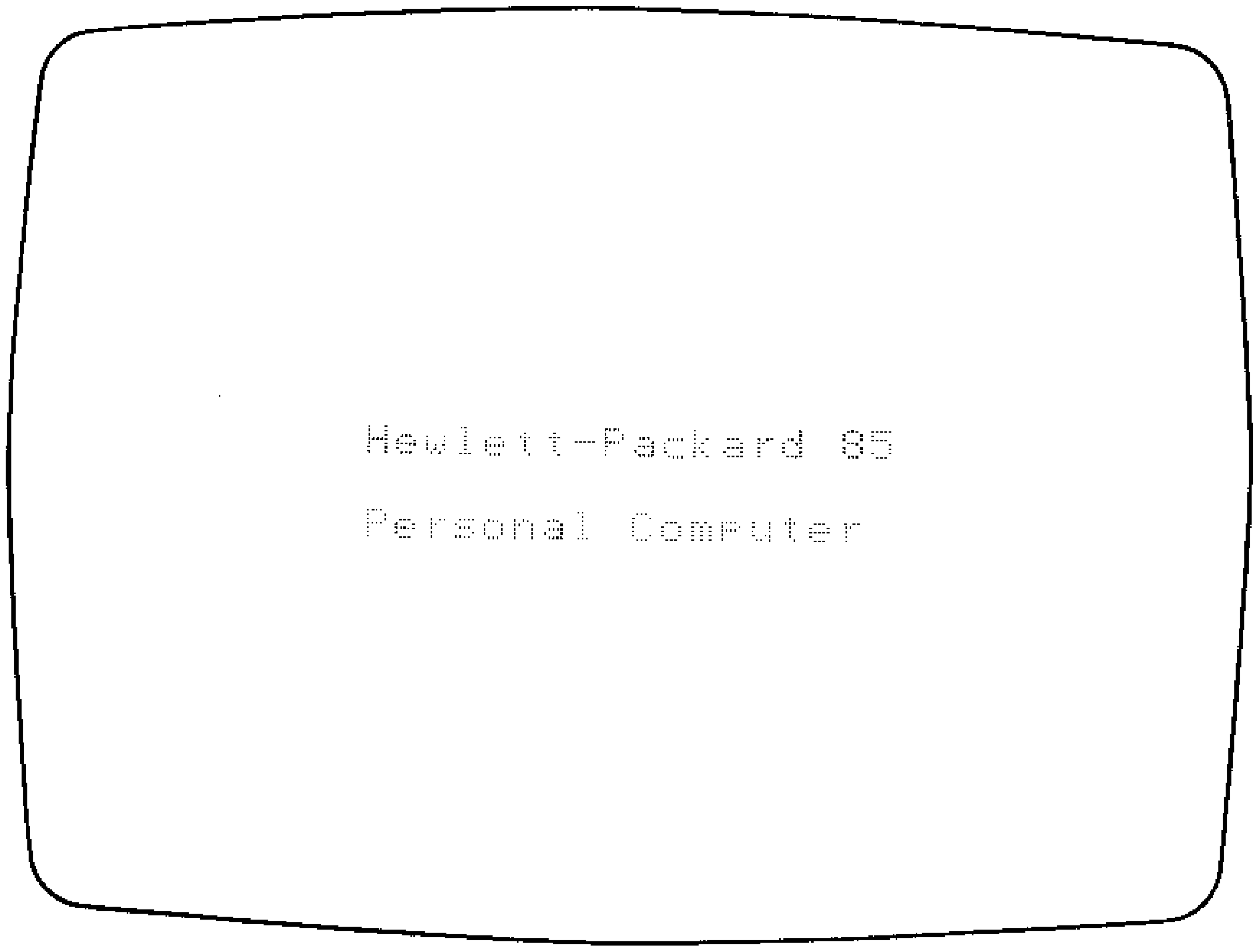
Example from page 214.

Moves to point −2.75, −1.5.
Then labels.

In each of the examples, we first direct the pen to the starting position of the label, then we specify the expression.

Example: Enter and run the following program.

```
 10 GCLEAR                              Clears graphics display.
 20 SCALE -1,1,-1,1                     Scales X and Y units.
 30 MOVE -.5,.1                         Moves to point -0.5,0.1.
•40 LABEL "Hewlett-Packard 85"          Writes expression on graphics display.
 50 MOVE -.5,-.1                        Moves to point -0.5,-0.1.
•60 LABEL "Personal Computer"           Writes expression.
 70 END
```



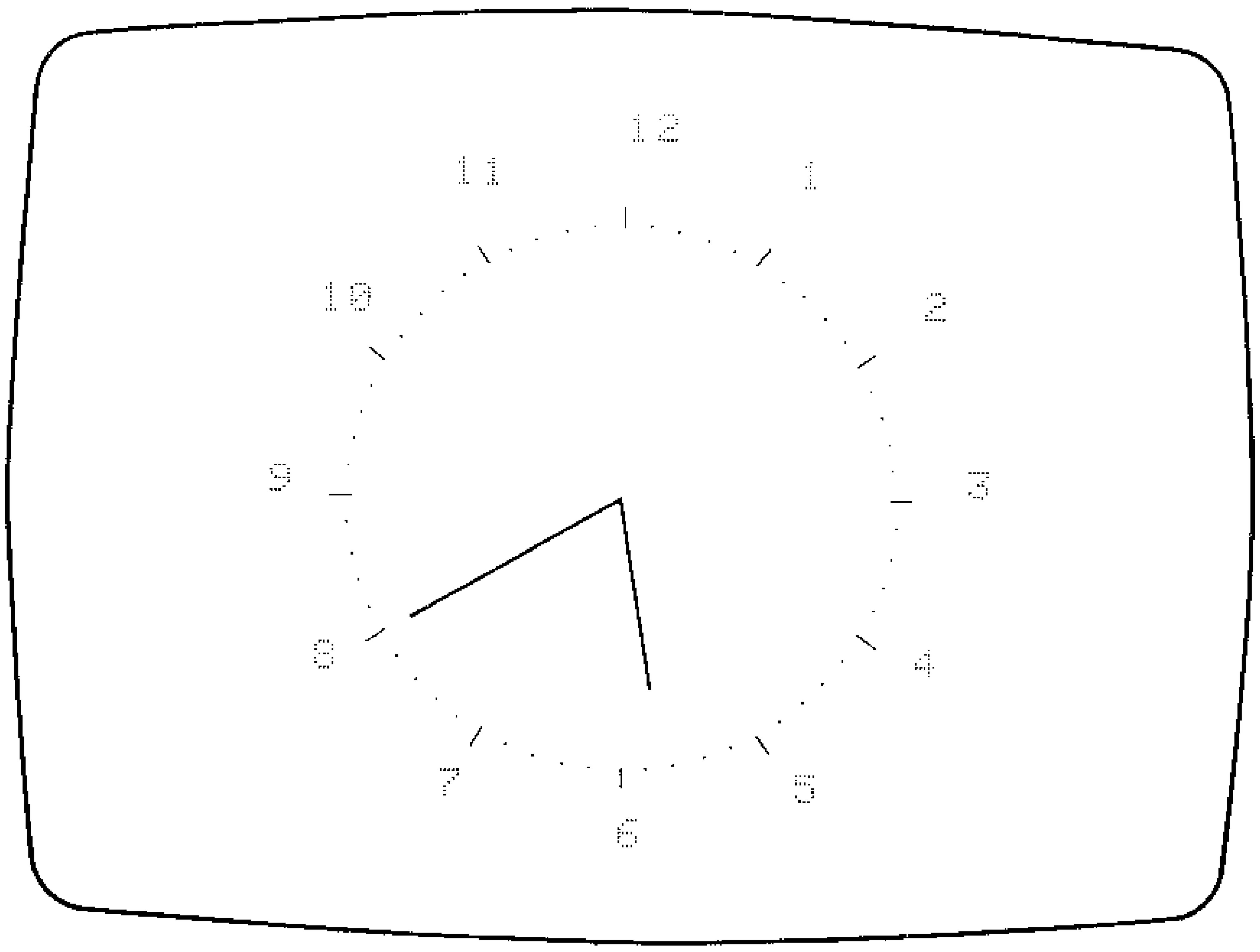
Example: Draw and label the face of a clock. Include a subroutine to draw the hour hand and the second hand for a time that you input. Write the program in such a way that the old time will be erased before a new time is drawn.

`10 PEN 1 @ GCLEAR`	Specifies positive pen; clears graphics display.
`20 SCALE -2,2,-3/2,3/2`	Equal unit scale.
`30 DEG`	Sets degrees mode.
`40 ! FACE OF CLOCK`	
`50 FOR M=0 TO 360 STEP 6`	Draws minute marks.
`60 MOVE SIN(M),COS(M)`	
`70 IDRAW SIN(M)/50,COS(M)/50`	
`80 IF M MOD 5 THEN 100`	Draws larger marks for 5-minute intervals.
`90 IDRAW SIN(M)/15,COS(M)/15`	
`100 NEXT M`	
`110 FOR I=1 TO 12`	
`120 MOVE 1.3*SIN(30*I),1.3*COS(30*I)`	Labels the clock with hours.
`130 LABEL VAL$(I)`	
`140 NEXT I`	
`150 ALPHA`	Puts display in alpha mode for input.
`160 DISP "INPUT TIME; HH.MM"`	
`170 INPUT T`	Inputs time in form HH.MM.
`180 GOSUB 1000`	Goes to subroutine to draw hands.
`190 PAUSE`	Pauses to display clock.
`200 PEN -1`	Specifies negative pen.
`210 GOSUB 1000`	Gosub to erase hands.
`220 PEN 1`	Specifies positive pen again.
`230 GOTO 150`	Goes back to line 150 to input new time.
`1000 REM *DRAW HANDS OF CLOCK*`	
`1010 MOVE 0,0`	Moves to middle of clock.
`1020 H=30*IP(T)`	
`1030 M=6*FP(T)*100`	
`1040 DRAW .7*SIN(H+M/12),.7*COS(H+M/12)`	Shorter hour hand.
`1050 MOVE 0,0`	
`1060 DRAW .9*SIN(M),.9*COS(M)`	Larger minute hand.
`1070 RETURN`	

Run the program for times of 5:40 and 10:15:

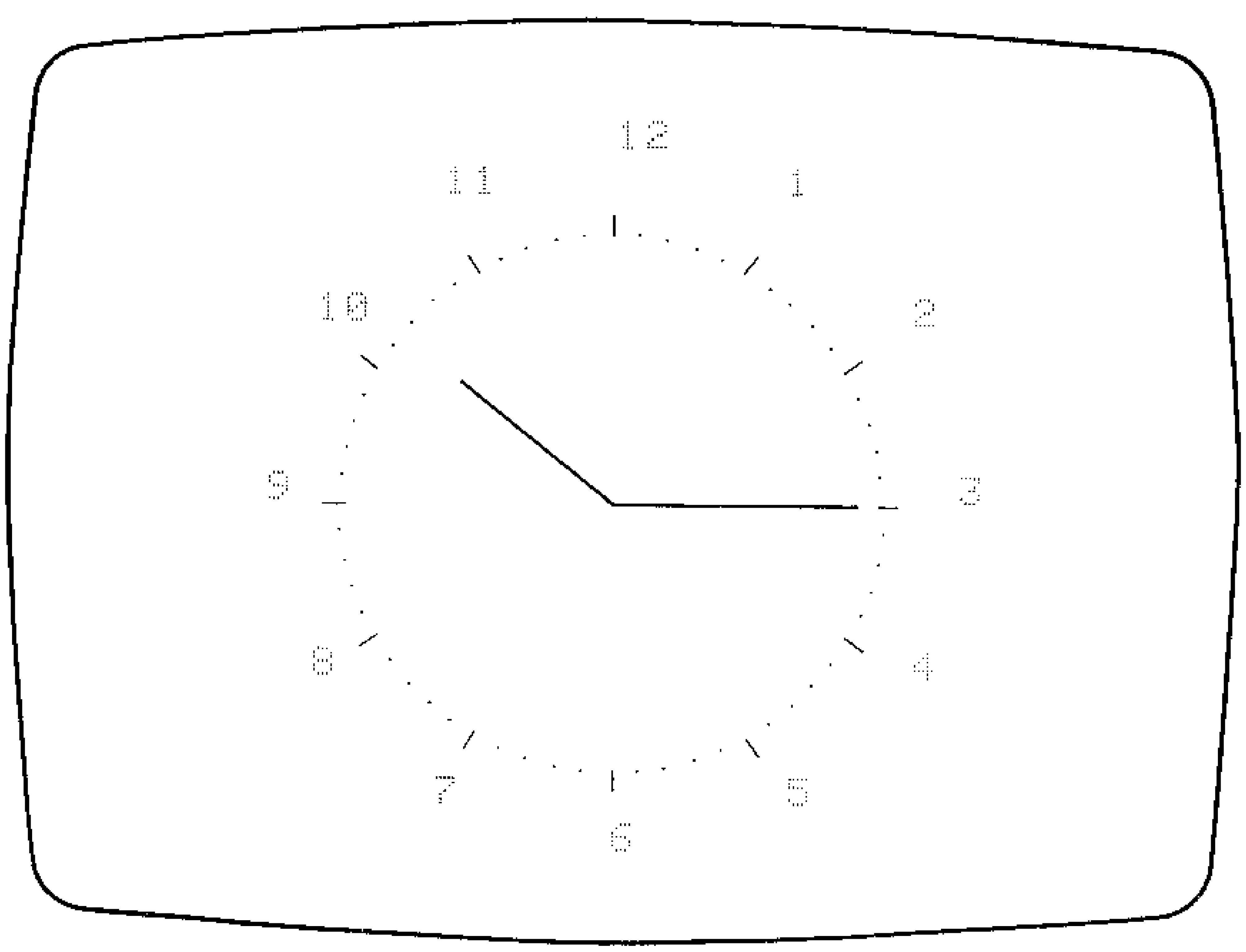
RUN

```
INPUT TIME; HH.MM
?
5.40
```



CONT

```
INPUT TIME; HH.MM
?
10.15
```



Label Direction

Character positions are much more flexible in graphics mode than alpha mode. Labels can be positioned either vertically or horizontally by using the `LDIR` *(label direction)* statement.

`LDIR` numeric expression

If the expression has a rounded integer value less than 45, labels will be positioned horizontally. If the value of the numeric expression (rounded to an integer) is greater than or equal to 45, labels will be positioned vertically. Thus:

`LDIR 0`	Specifies horizontal labels.
`LDIR 90`	Specifies vertical labels.

Example:

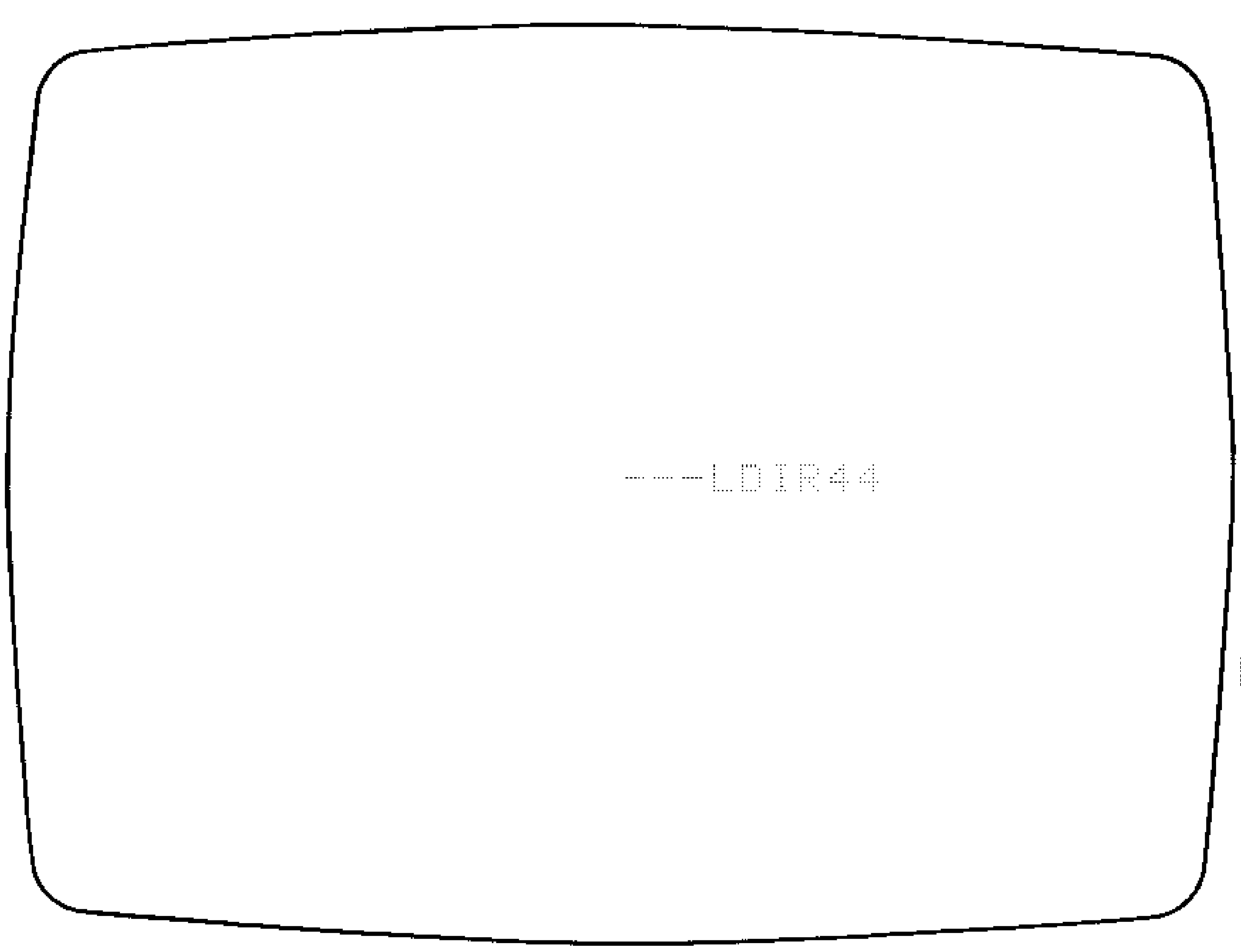
`10 SCALE -10,10,-10,10`	
`20 ALPHA`	Sets display to alpha mode for displaying input message.
`30 DISP "ENTER A NUMBER FROM 0 THROUGH 90"`	
`40 INPUT D`	Inputs label direction.
`50 PEN 1 @ GCLEAR`	Clears graphics display.
• `60 LDIR D`	Sets label direction.
`70 MOVE 0,0`	Moves pen to point 0,0.
`80 LABEL "---LDIR"&VAL$(D)`	Writes label on graphics display.
`90 PAUSE`	
`100 GOTO 20`	
`110 END`	

Run the program above with test values of D. Press (CONT) each time you wish to enter a new label direction.

We run the program with values of 44 and 45.

(RUN)

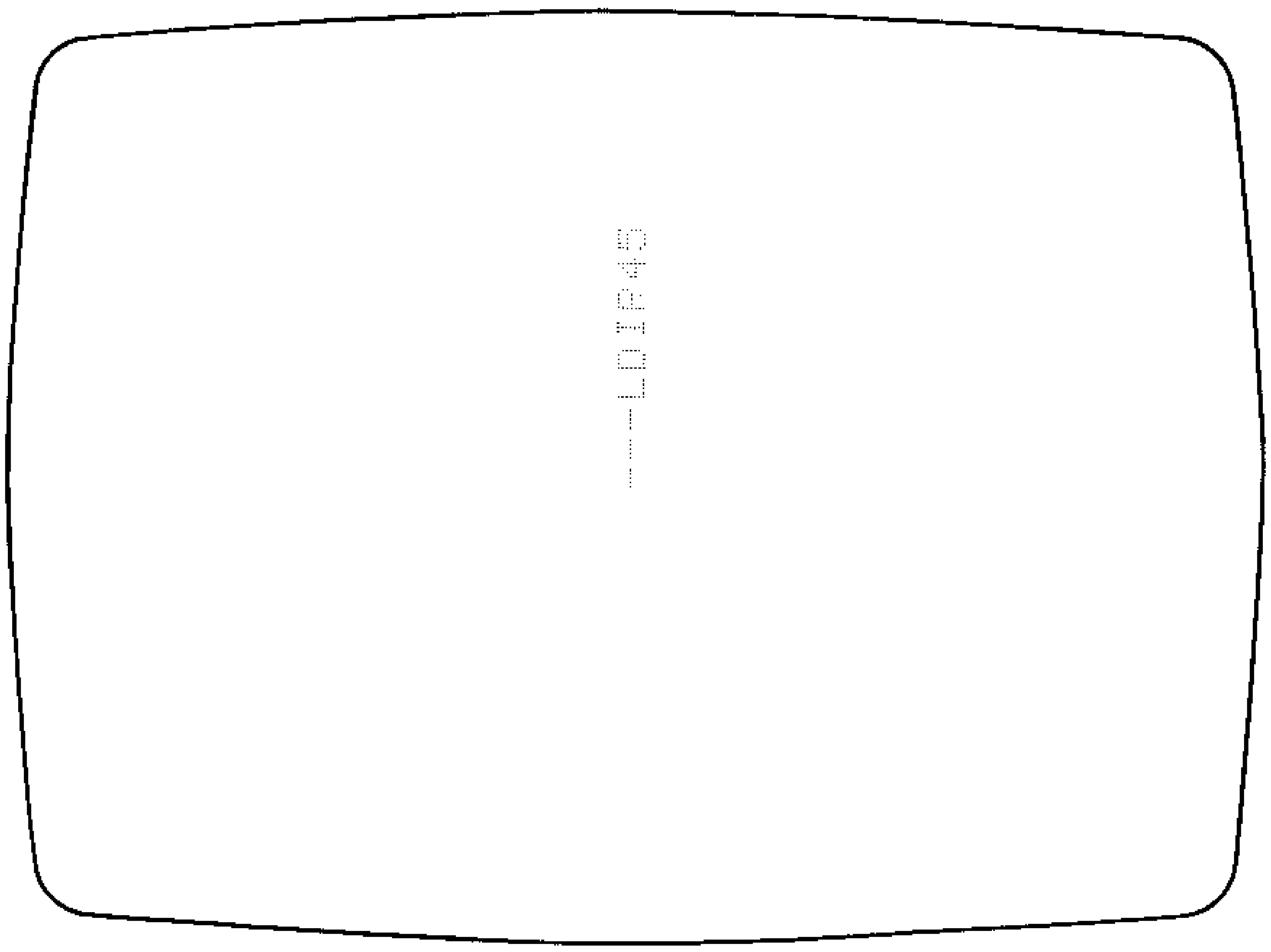
```
ENTER A NUMBER FROM 0 THROUGH 90
?
44
```



`LDIR44` yields a horizontal label.

CONT

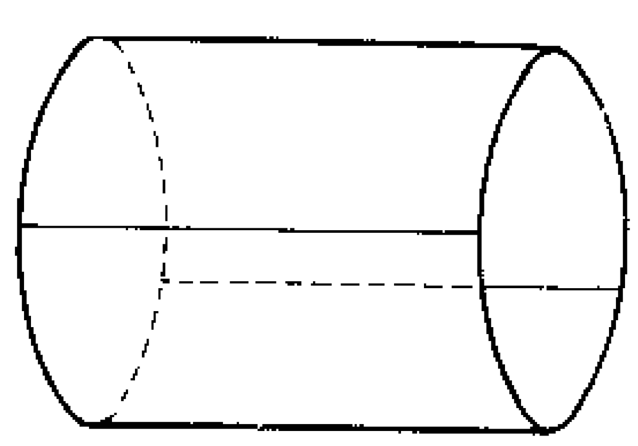
```
ENTER A NUMBER FROM 0 THROUGH 90
?
45
```



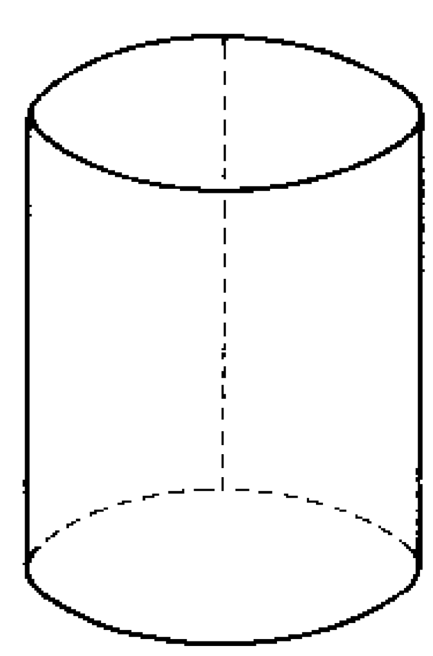
LDIR45 yields a vertical label.

Label Length

You can think of the alpha display as a cylinder with four displays connected from top to bottom. But the graphics display treats characters as if it were a cylinder composed of *one* display with the right and left edges connected:



Alpha Display

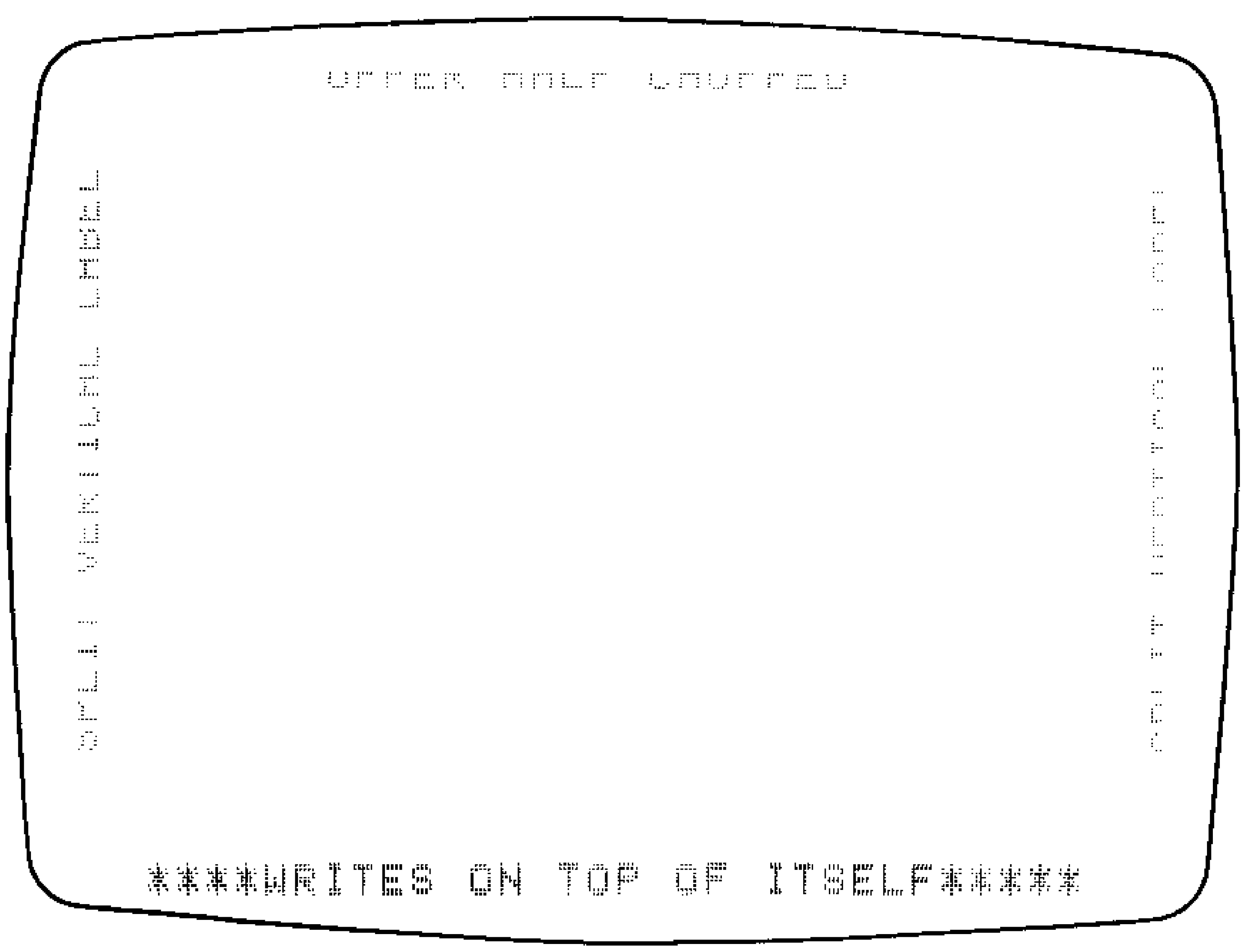


Graphics Display

Thus, characters or lines do not cause the graphics display to scroll. In fact, characters will be chopped off if they are positioned too high on the graphics display. If vertical labels are positioned too far to the left of the display, part of the label will be written on the right boundary of the graphics display. A horizontal label longer than 32 characters will wrap around on top of itself, one dot below the original starting position.

Example: This program illustrates the effects of the graphics display on character labels.

```
  10 GCLEAR
  20 SCALE 0,10,0,10
  30 LDIR 0                                  Horizontal label setting.
  40 MOVE 1,9.8                              Moves to point 1,9.8.
• 50 LABEL "   UPPER HALF CHOPPED            Writes lower part of label.
     "
  60 MOVE 0,0                                Moves to point 0,0.
• 70 LABEL "****WRITES ON TOP OF             Writes label.
     ITSELF**********WRITES ON TOP
      OF ITSELF*****"
  80 LDIR 90                                 Changes label direction.
  90 MOVE .1,1.2                             Moves to point 0.1,1.2.
•100 LABEL "SPLIT VERTICAL LABEL"            Writes label.
 110 END
```



A label direction setting will remain the same until it is changed by another LDIR statement. Unless otherwise specified, labels will automatically be positioned horizontally. As we shall see, any input in graphics mode resets LDIR to horizontal labeling.

If a vertical label begins at the bottom of the display, a maximum of 24 characters will be written on the graphics display. Any remaining characters will be chopped off of the top of the display.

Positioning Labels

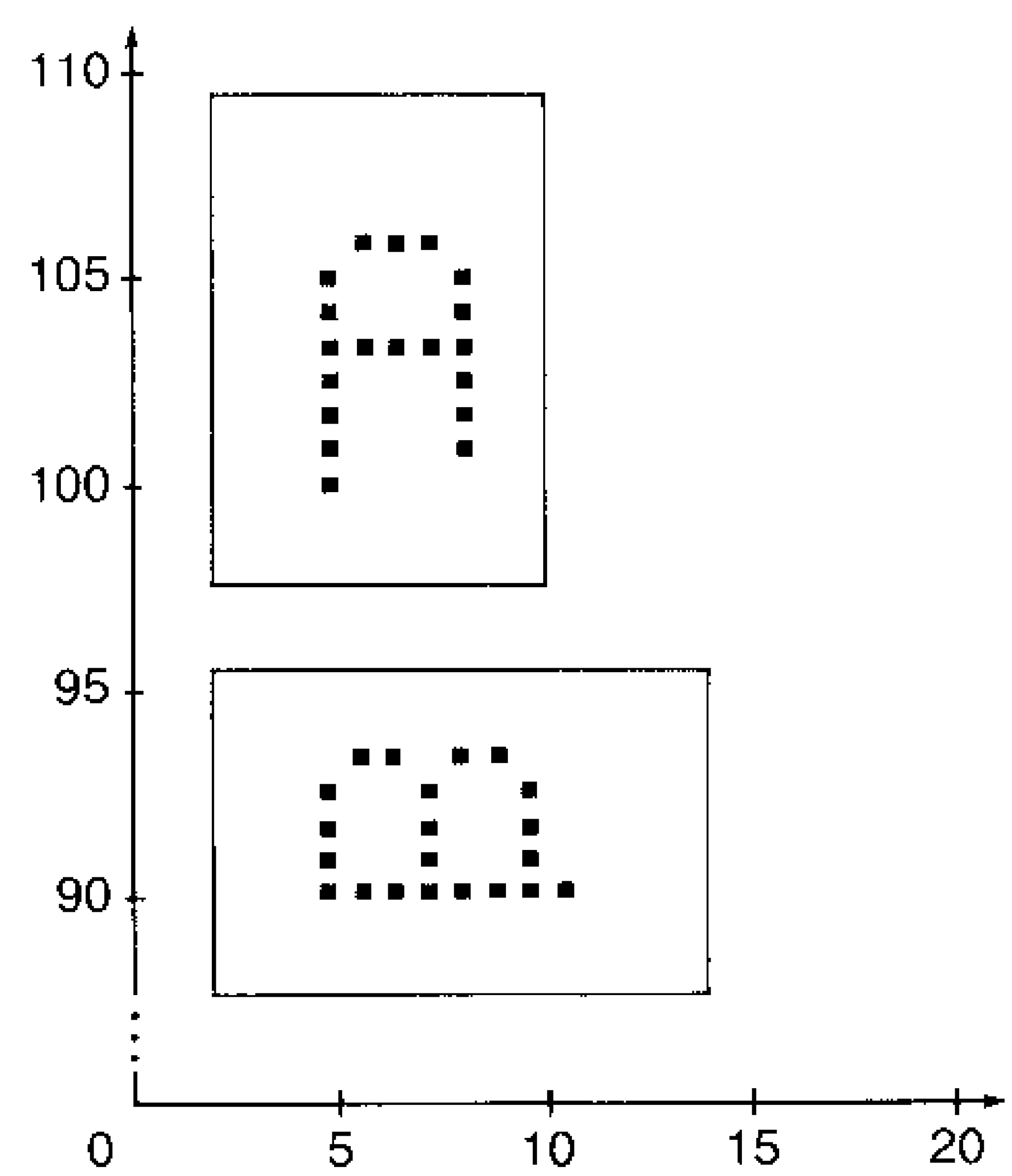
The position of a label is determined both by the LDIR statement and by the current pen location. Horizontal labels begin directly above the point specified by the current pen location. Vertical labels begin directly to the left of the specified pen location.

It is often easier to scale the display to the number of plotting dots available to specify exact label locations, from 0 to 255 in the horizontal direction (256 dots) and from 0 to 191 in the vertical direction (192). Since a character is composed of a 5 × 7 dot character on an 8 × 12 dot field, you can easily calculate the number of dots necessary for a particular label.

To illustrate label starting positions, we ran the following program and then enlarged the display area around the labels.

```
  10 GCLEAR
  20 SCALE 0,255,0,191
  30 PENUP
  40 PLOT 4,100
• 50 LDIR 0
• 60 LABEL "A"
  70 PENUP
  80 PLOT 11,90
• 90 LDIR 90
•100 LABEL "B"
 110 END
```

Let's look at the section of the display around the labels:



A label is positioned directly above (horizontal) or to the left (vertical) of the current pen location to allow for underscored characters (character codes 128 through 255). Since we plotted point (4,100) immediately before the LABEL statement, A was positioned as shown above. If we had plotted or moved to (25,90) for instance the A would be positioned so that the left leg of the A would be directly above point (25,90). And if A was a vertical label it would be plotted so that the left leg of the A would be directly to the left of the point (25,90).

Example: Earlier, we wrote a program to plot the men's and women's records for 100-meter freestyle swimming races in the Summer Olympics from 1948 through 1976. Now let's see how easy it is to label the graph. If you stored the program on page 219, load it now and add statements 55 and 3000 through 4000 of the following program:

```
 10 REM *100-METER FREESTYLE SUM
    MER OLYMPICS 1948-1976
 20 GCLEAR
 30 SCALE 1940,1978,42,70.5
 40 XAXIS 48,4,1944,1978
 50 YAXIS 1944,1,48,70
 55 GOSUB 3000
```

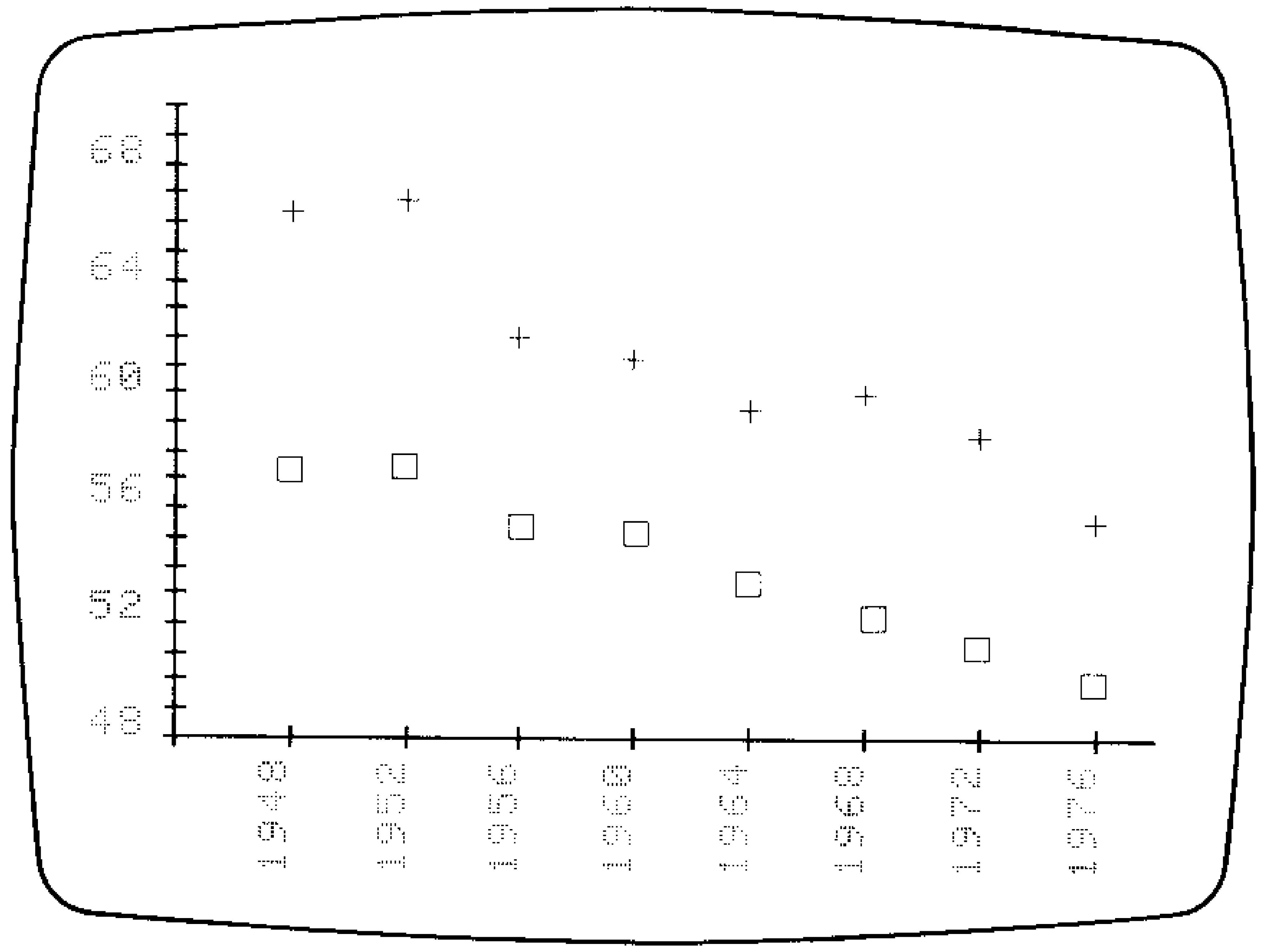
Go to the subroutine to label the axes.

```
 60 FOR I=1948 TO 1976 STEP 4
 70 READ M
 80 GOSUB 1000
 90 READ W
100 GOSUB 2000
110 NEXT I
120 DATA 57.3,66.3,57.4,66.8,55.
    4,62,55.2,61.2,53.4,59.5,52.
    2,60,51.22,58.59,49.99,55.65
130 STOP
1000 REM *PLOT SQUARE*
1010 MOVE I,M
1020 IMOVE -.2,.2
1030 IDRAW .4,0 @ IDRAW 0,-.4
1040 IDRAW -.4,0 @ IDRAW 0,.4
1050 RETURN
2000 REM *PLOT CROSS*
2010 MOVE I,W
2020 IMOVE 0,.3 @ IDRAW 0,-.6
2030 IMOVE .3,.3 @ IDRAW -.6,0
2040 RETURN
3000 REM *LABEL X-AXIS*
3010 LDIR 90
3020 FOR X=1948 TO 1976 STEP 4
3030 MOVE X,43
• 3040 LABEL VAL$(X)
3050 NEXT X
```

Labels X-axis from 1948 through 1976 in increments of 4 years.
Moves to the designated X,Y position, then labels.

```
3060 REM *LABEL Y-AXIS*
3070 LDIR 0
3080 FOR Y=48 TO 70 STEP 4
3090 MOVE 1941, Y
• 3100 LABEL VAL$(Y)
3110 NEXT Y
3120 RETURN
4000 END
```

Labels Y-axis from 48 to 70 in increments of 4 seconds.

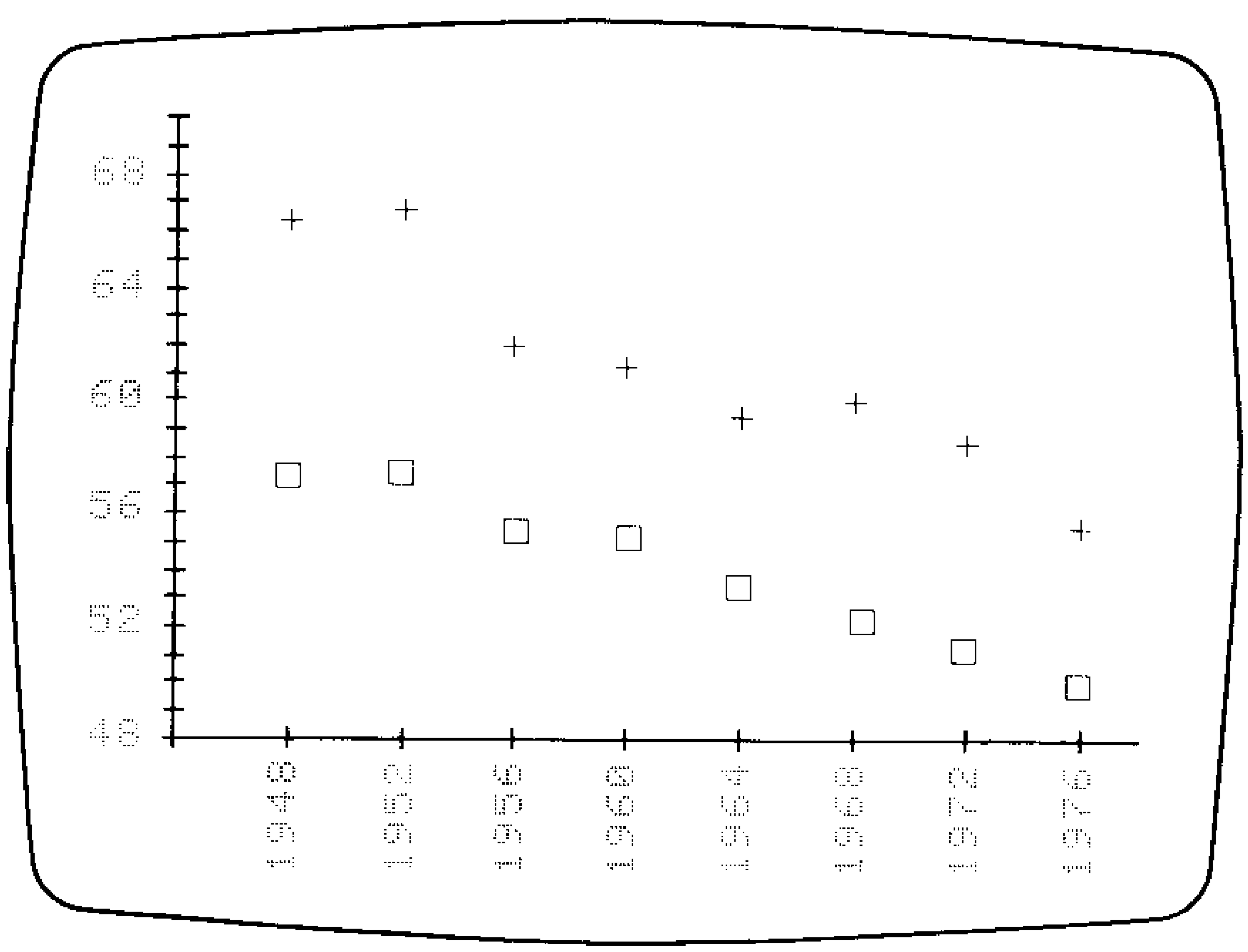


Horizontal labels are positioned immediately above the specified point; vertical labels are positioned immediately to the left of the specified point.

To center the labels next to the tic marks on the axes, change statements 3030 and 3090 to read as follows:

```
3030 MOVE X+.5,43
3090 MOVE 1941,Y-.5
```

Now run the program again:



Labels are centered on the tic marks.

You'll find that labels can be positioned easily at the desired location by adding or subtracting fractions of the units that you specify.

Of course, you could calculate the exact location of the label by scaling the graphics display to the number of plotting dots available, as we suggested earlier.

Problem

12.5 If you toss an unbiased coin a number of times, you will get all heads or all tails or more likely, some combination of heads and tails. Test your graphics programming skills by plotting a histogram of the theoretical probability distribution of the various numbers of heads you might obtain by tossing a fair coin ten times. Label the number of heads along the X-axis from 0 to 10. Since you will be graphing a histogram, center the labels under each unit on the axis. Label the probability along the Y-axis in intervals of 0.02 from 0 to 0.26.

Hints:

1. To find the various probabilities, evaluate each term of the binomial expansion:

$$(p+q)^n = \sum_{r=0}^{n} \frac{n!}{r!\,(n-r)!} p^{n-r}q^r \quad \text{where } p = q = 1/2 \text{ and } n = 10.$$

   For example, to find the probability of obtaining three heads and seven tails in 10 tosses, evaluate the term:

$$\frac{10!}{3!\,7!} \quad (1/2)^7\,(1/2)^3 \approx 0.117$$

2. Define a factorial function to use in the above computation.
3. Remember to allow enough space in the SCALE statement for labels along the axes.

INPUT in Graphics Mode

One of the most useful features of HP-85 graphics is the system's ability to take inputs from the keyboard while the display remains in graphics mode. Thus, you can study the graphics display and input information to a program without the display reverting back to alpha mode.

There is an important difference between input in alpha mode and input in graphics mode. Whereas all of the display editing keys are active on input in alpha mode (e.g., the [→] key causes the cursor to move right, etc.), *only the* [BACK SPACE] *key is active on input in graphics mode*—the rest of the display editing keys will display their respective key-codes when pressed in response to input on the graphics display. Remember, the [COPY], [PAPER ADV], [PAUSE], [GRAPH], [KEY LABEL], [ROLL], and [CLEAR] keys are still active with graphics mode input. Refer to the table of key responses in appendix C.

Since the (BACK SPACE) key is the only editing feature allowed in graphics mode, it has been given some special capabilities. We'll discuss the backspace features in conjunction with graphics input in the following program.

Example: Write a program that generates 90 random numbers between 0 and 20 and plots them in order of generation on a horizontal scale from 0 to 30. Using LABEL statements on the graphics display, prompt for inputs for a graph heading, and for X- and Y-axis labels.

```
  5 REM *RANDOM DATA PLOT AND LA
    BEL
 10 DIM H$[32],Y$[14],X$[27]
 20 SCALE -5,30,-10,25
 30 PEN 1 @ GCLEAR
 40 XAXIS 0,1,0,30
 50 YAXIS 0,1,0,20
 60 RANDOMIZE
 70 FOR I=1 TO 90
 80 Y=20*RND
 90 PENUP
100 PLOT I/3,Y
110 NEXT I
120 MOVE -5,-4
130 LDIR 0
•140 LABEL "ENTER HEADING"
150 MOVE -5,-6
•160 INPUT H$
170 H=LEN(H$)
180 H1=10-INT(H/2)*30/32
190 MOVE H1,22
200 LABEL H$
210 GCLEAR -1
220 MOVE -5,-4
•230 LABEL "LABEL Y-AXIS (MAX 14
    CHARS)"
240 MOVE -5,-6
•250 INPUT Y$
260 Y=LEN(Y$)
270 Y1=10-INT(Y/2)*10/7
280 MOVE -1,Y1
290 LDIR 90
300 LABEL Y$
310 GCLEAR -1
320 MOVE -5,-6
330 LDIR 0
•340 LABEL "NOW LABEL X-AXIS (MAX
    27 CHARS)"
350 MOVE -5,-8
•360 INPUT X$
370 X=LEN(X$)
380 X1=12-INT(X/2)*32/35
390 GCLEAR -2
400 MOVE X1,-3
410 LABEL X$
420 END
```

Dimensions string input variables. (10)
Scales X and Y units. (20)

Draws axes. (40–50)

Plots random points between 0 and 20 in order of generation. (100)
Moves to desired point, then labels. (120–140)

Moves to desired input point on graphics display. (150)

Centers heading. (180)

Then labels. (200)
Clears previous prompt and input from graphics display. (210)
Prompts for next input. (230)

Moves to desired point of question mark appearance. (240)

Centers label along Y-axis. (270)

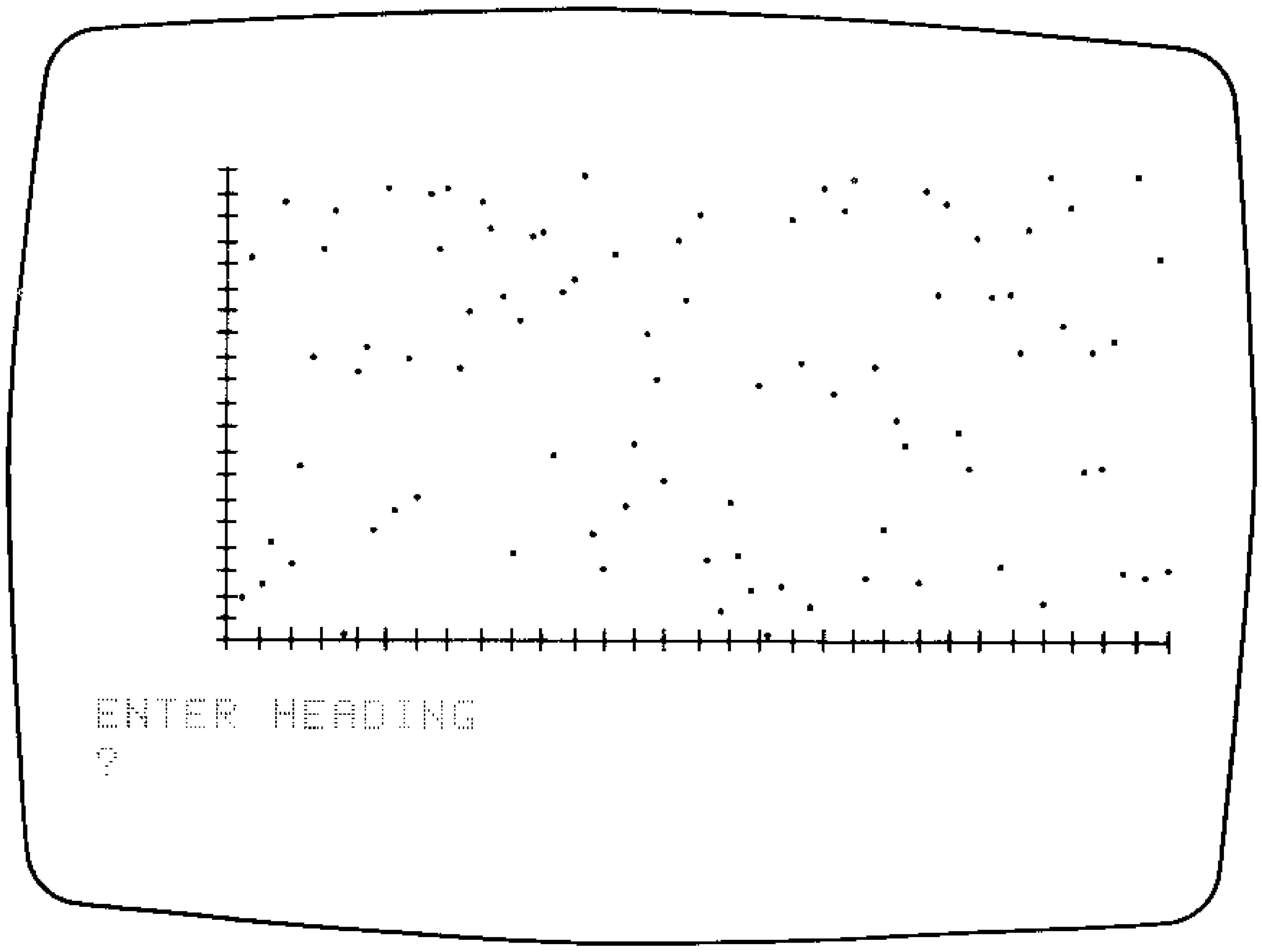
Changes label direction. (290)
Labels Y-axis. (300)
Clears previous prompt and input from graphics display. (310)

Inputs X-axis label. (360)

Centers label under X-axis. (380)
Clears prompt and input. (390)

Then labels. (410)

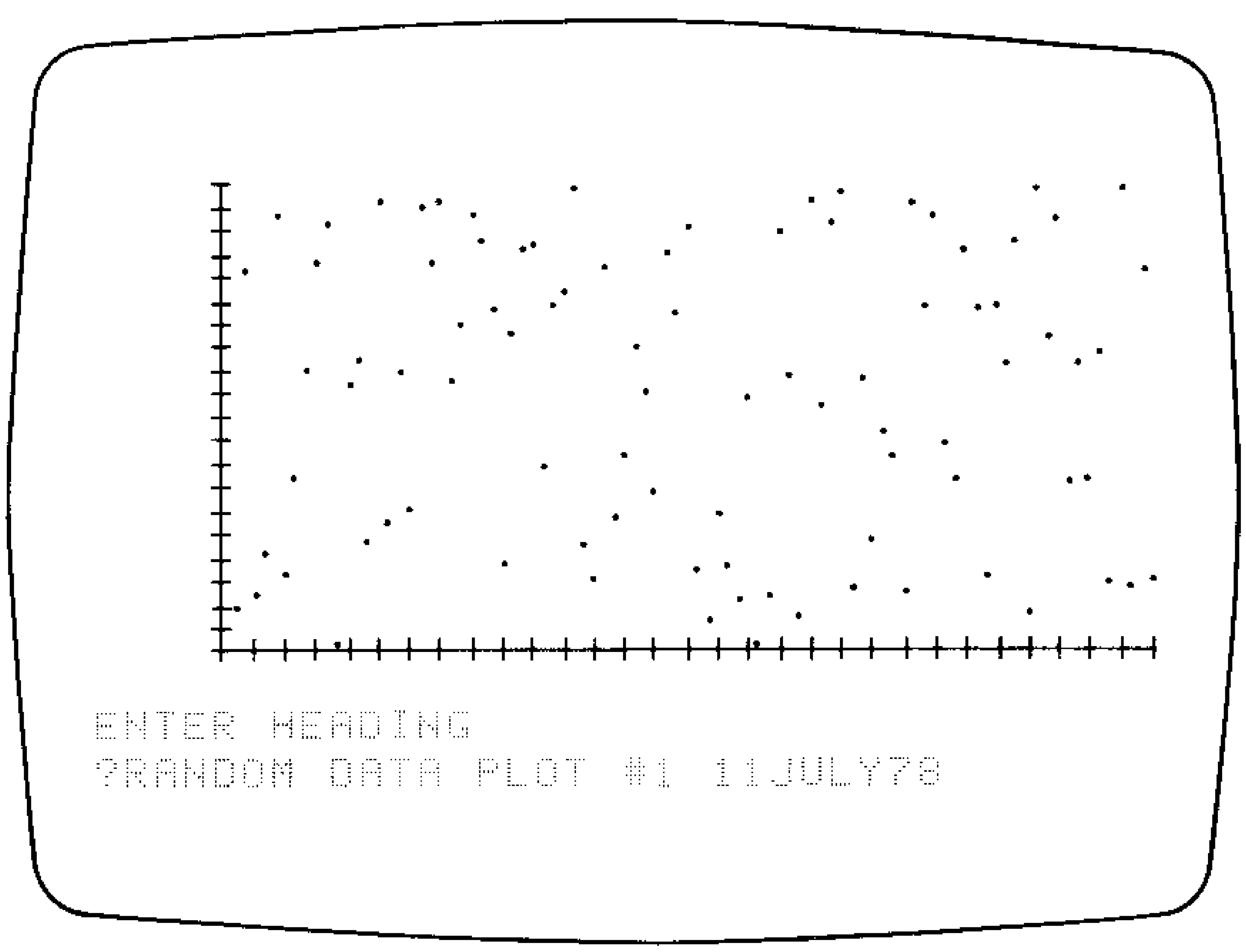
After you have entered the program, press (RUN). The graphics display shows:



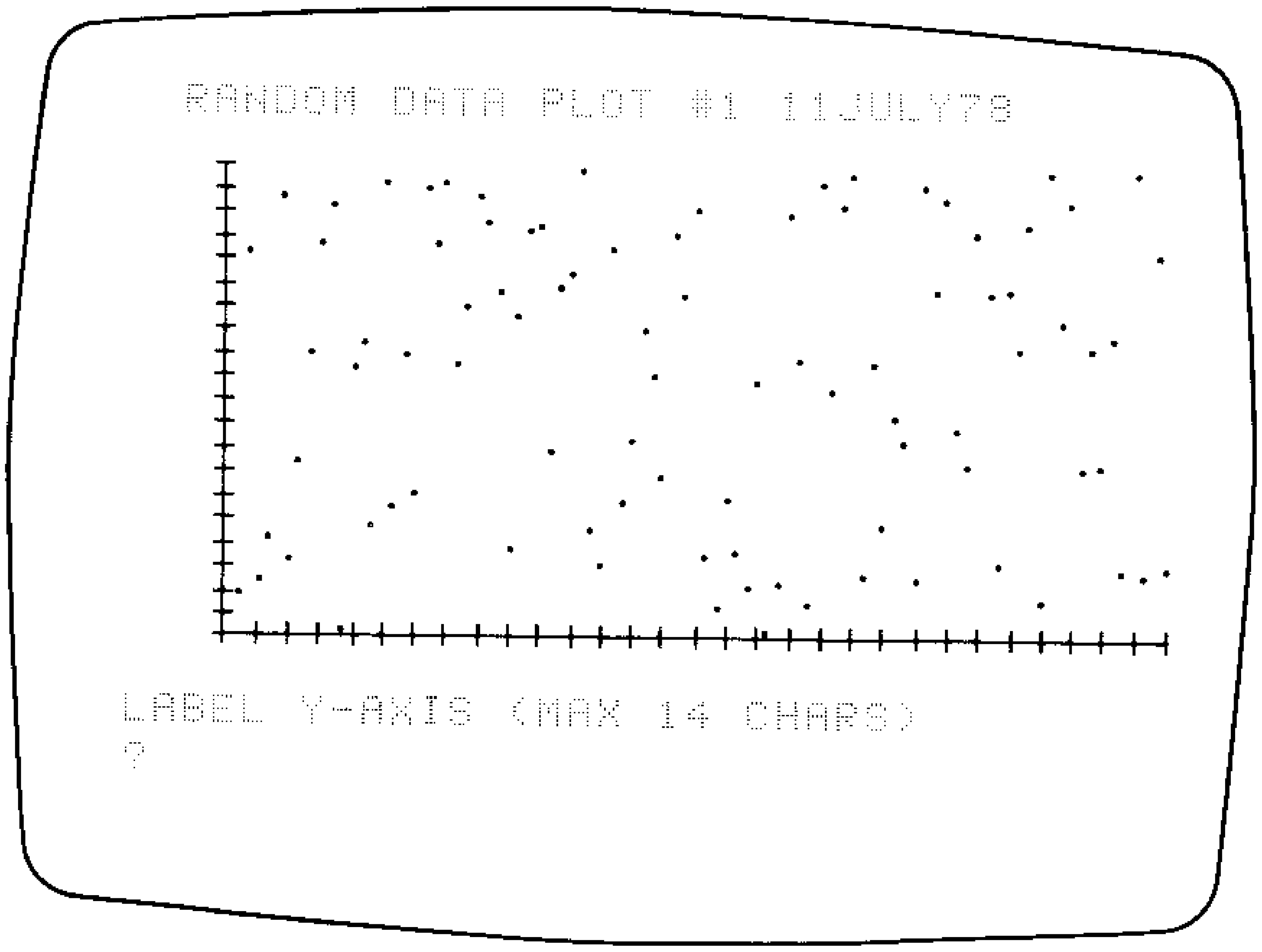
The input prompt, ?, appears on the graphics display if an INPUT statement is executed when the CRT display is in graphics mode.

Since you moved to point −5,−6 on the graphics display before the input statement, the question mark appears at point −5,−6. Now enter a heading for the data plot. If you make a typing mistake, backspace to erase and correct the error.

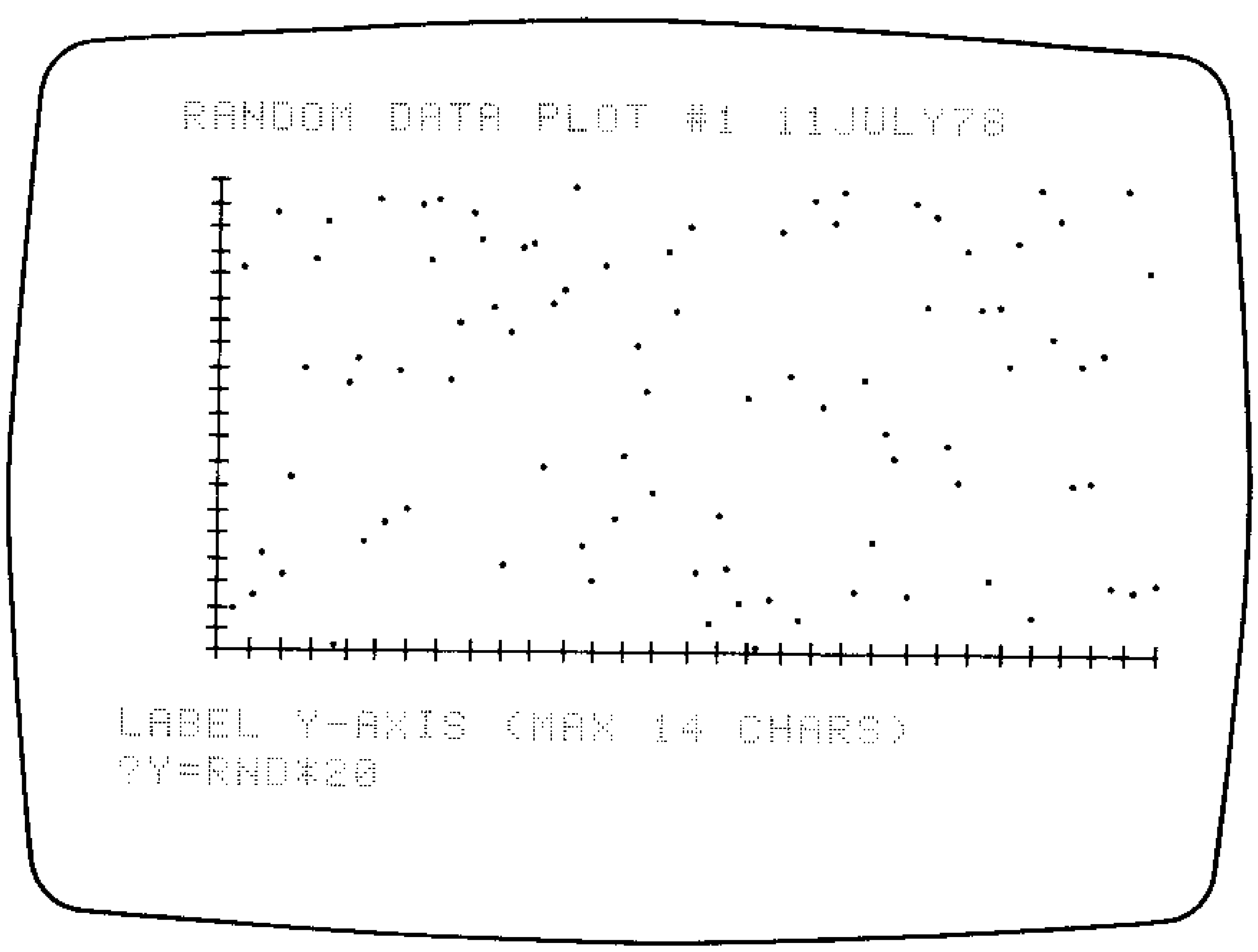
When you have completed typing the heading, press (END LINE). The display will remain in graphics mode while you type.



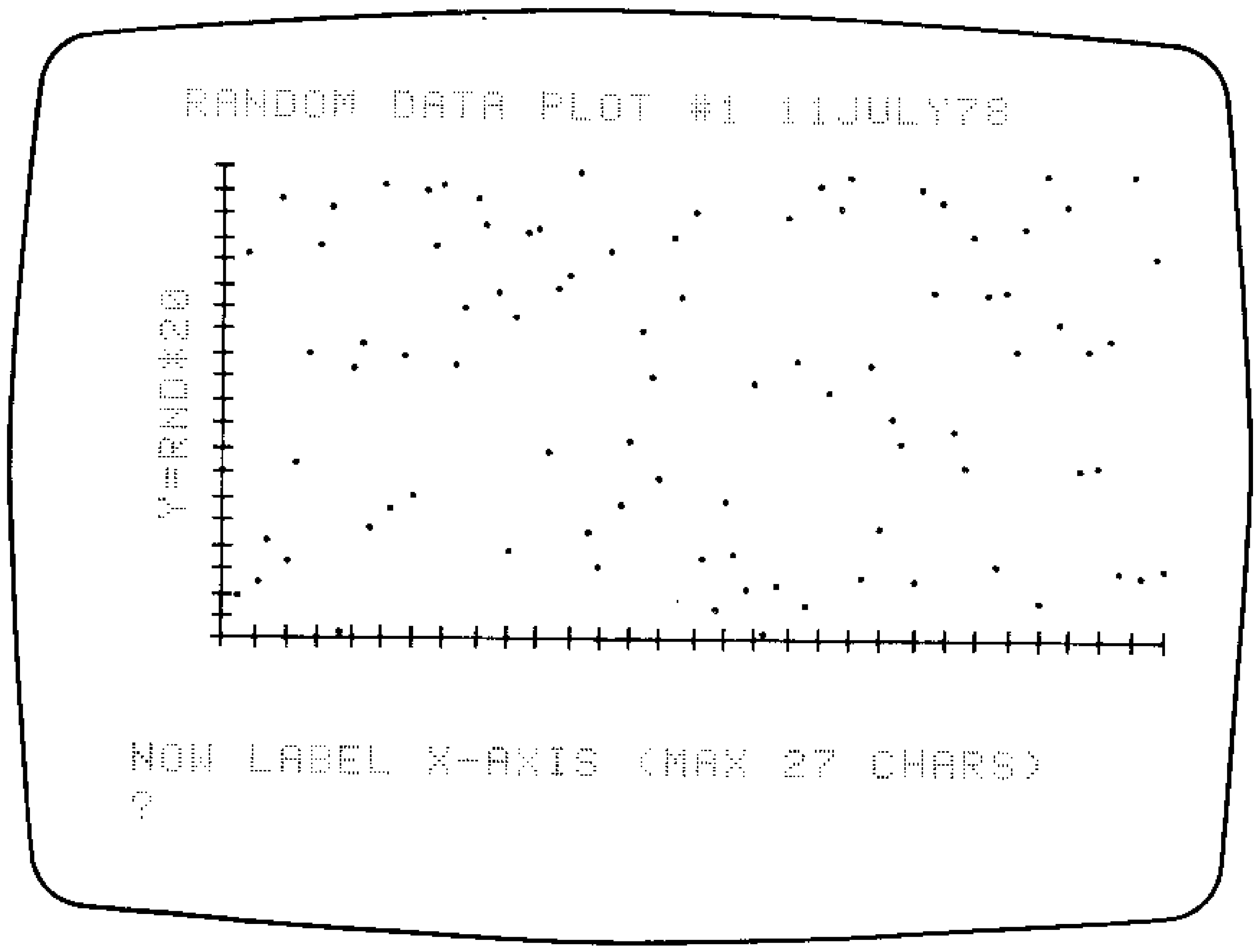
After you enter the heading, the program will prompt for the next label.



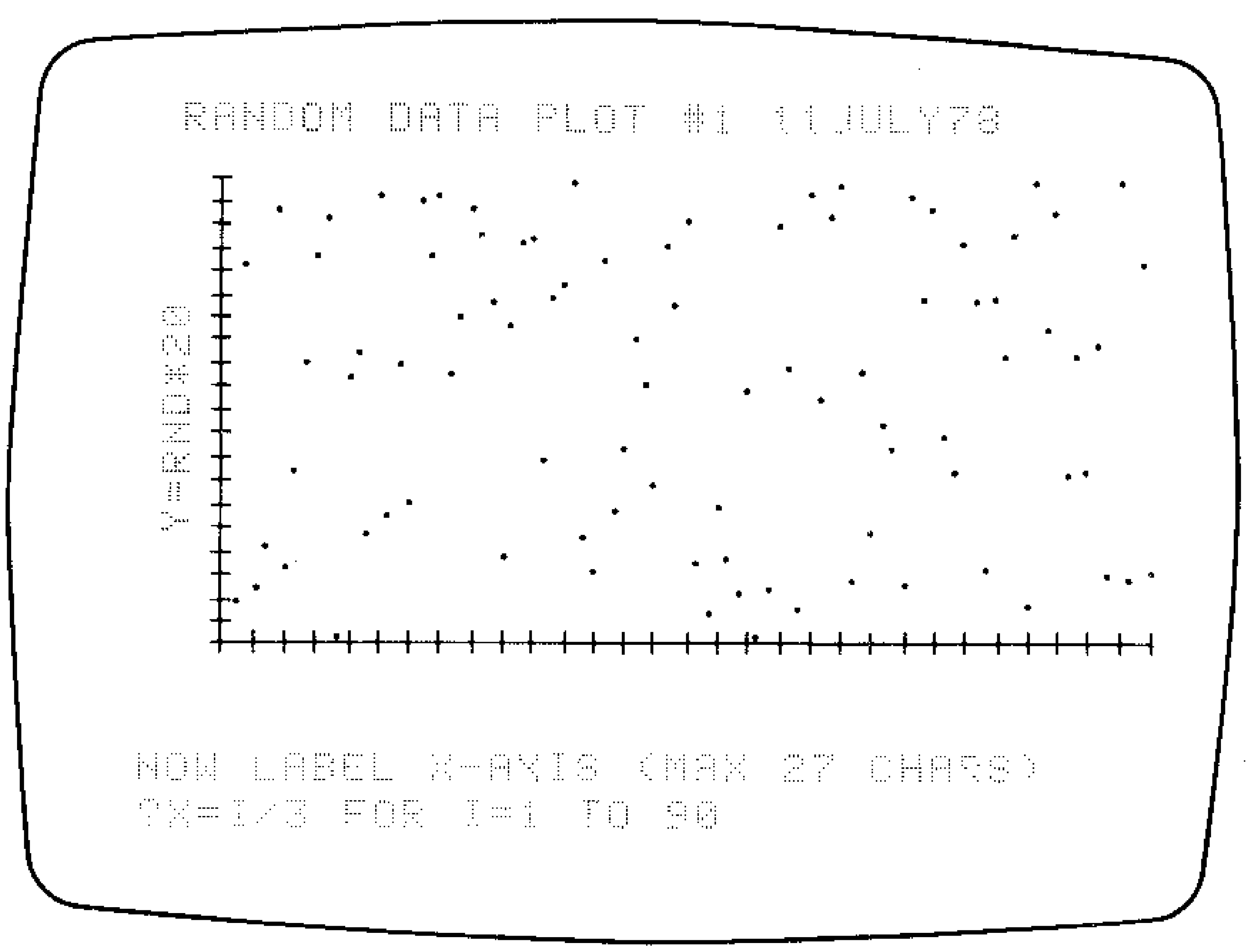
Again, the question mark appeared at point −5, −6 because we moved to that point prior to the `INPUT` statement. Enter the Y-axis label and press (END LINE).



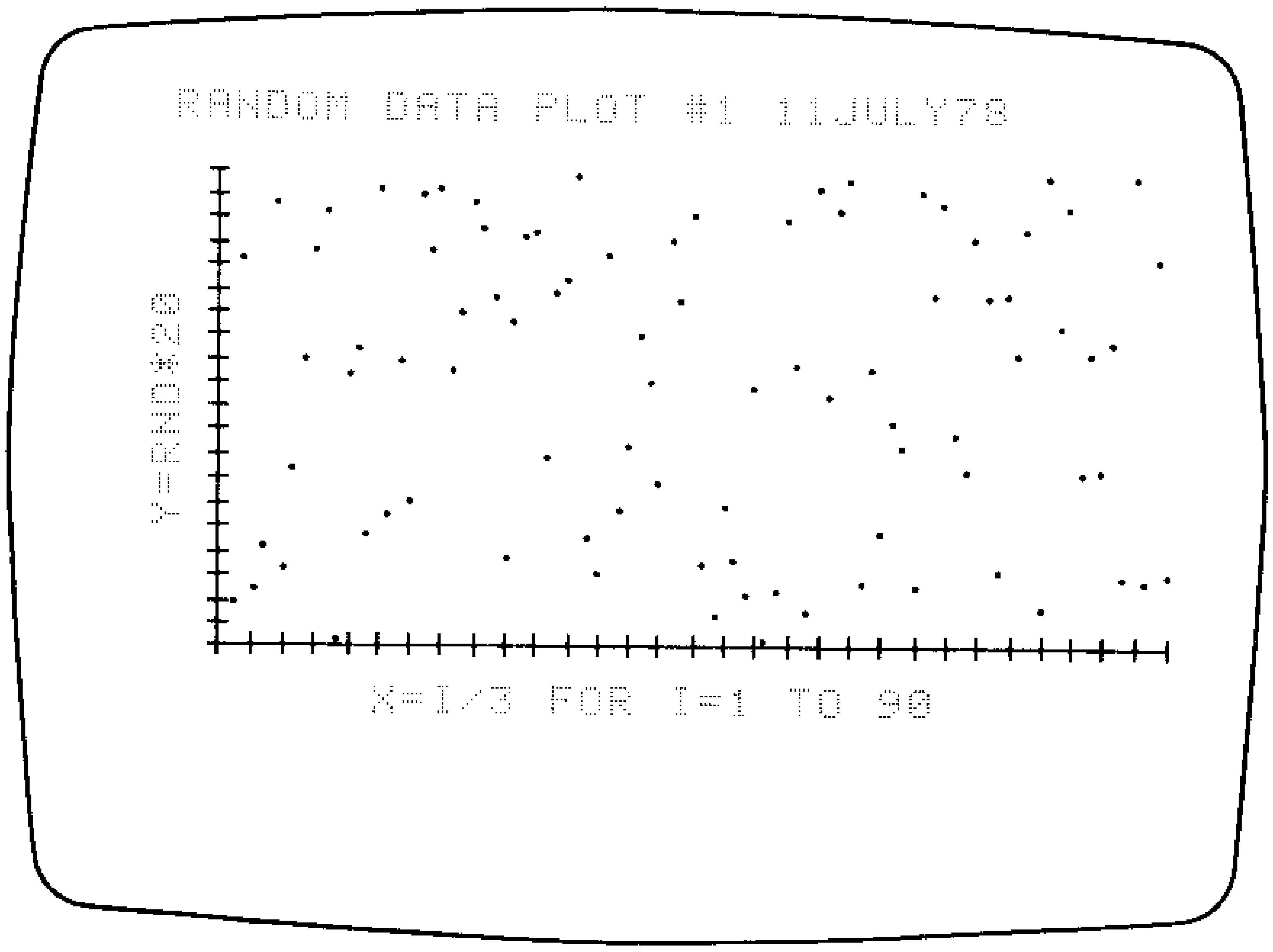
Now the program prompts for the last label.



Enter the label and press (END LINE).



The program centers the last input under the X-axis.

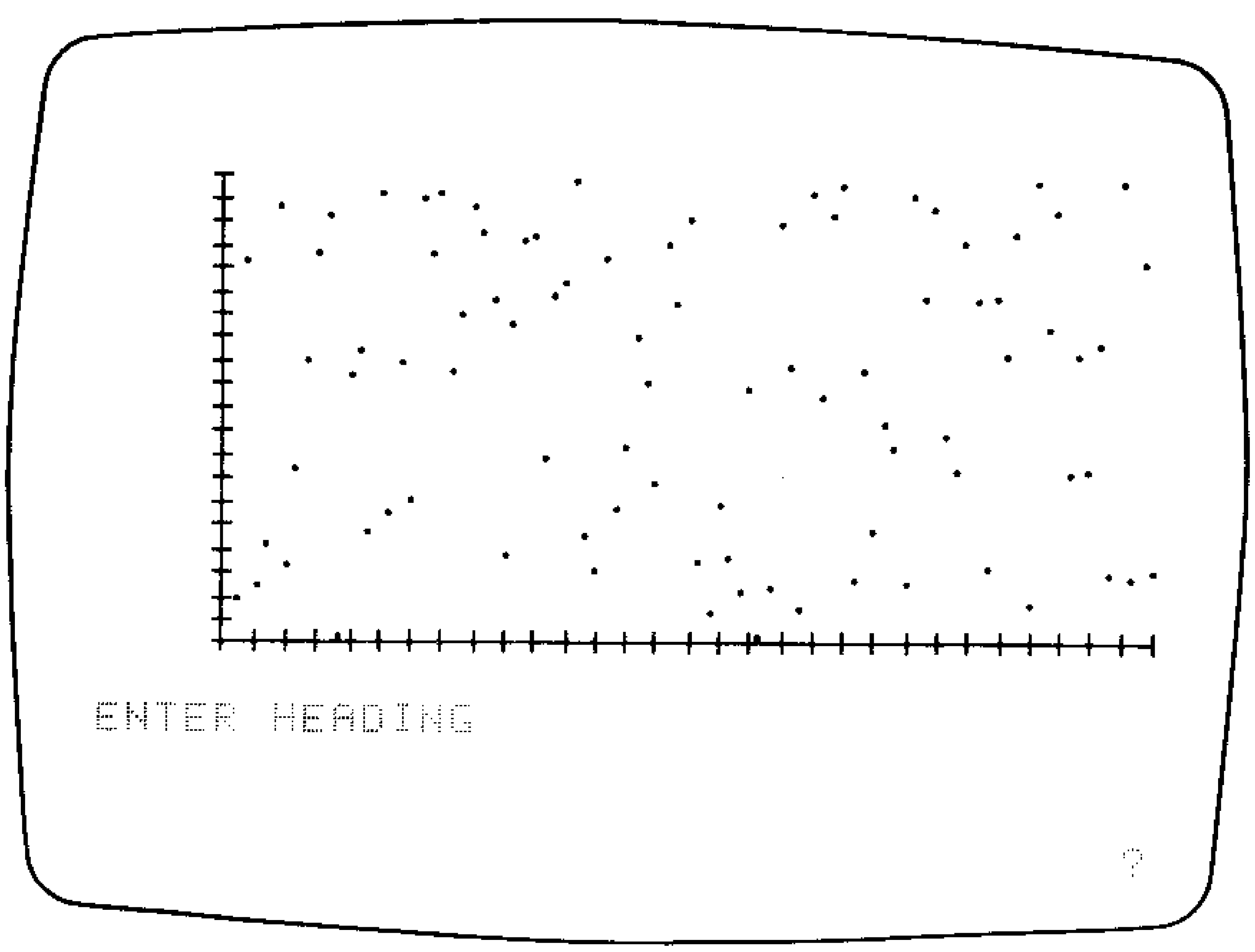


Experiment with the position of the input prompt, ?, to view the results of inputting information to the graphics display.

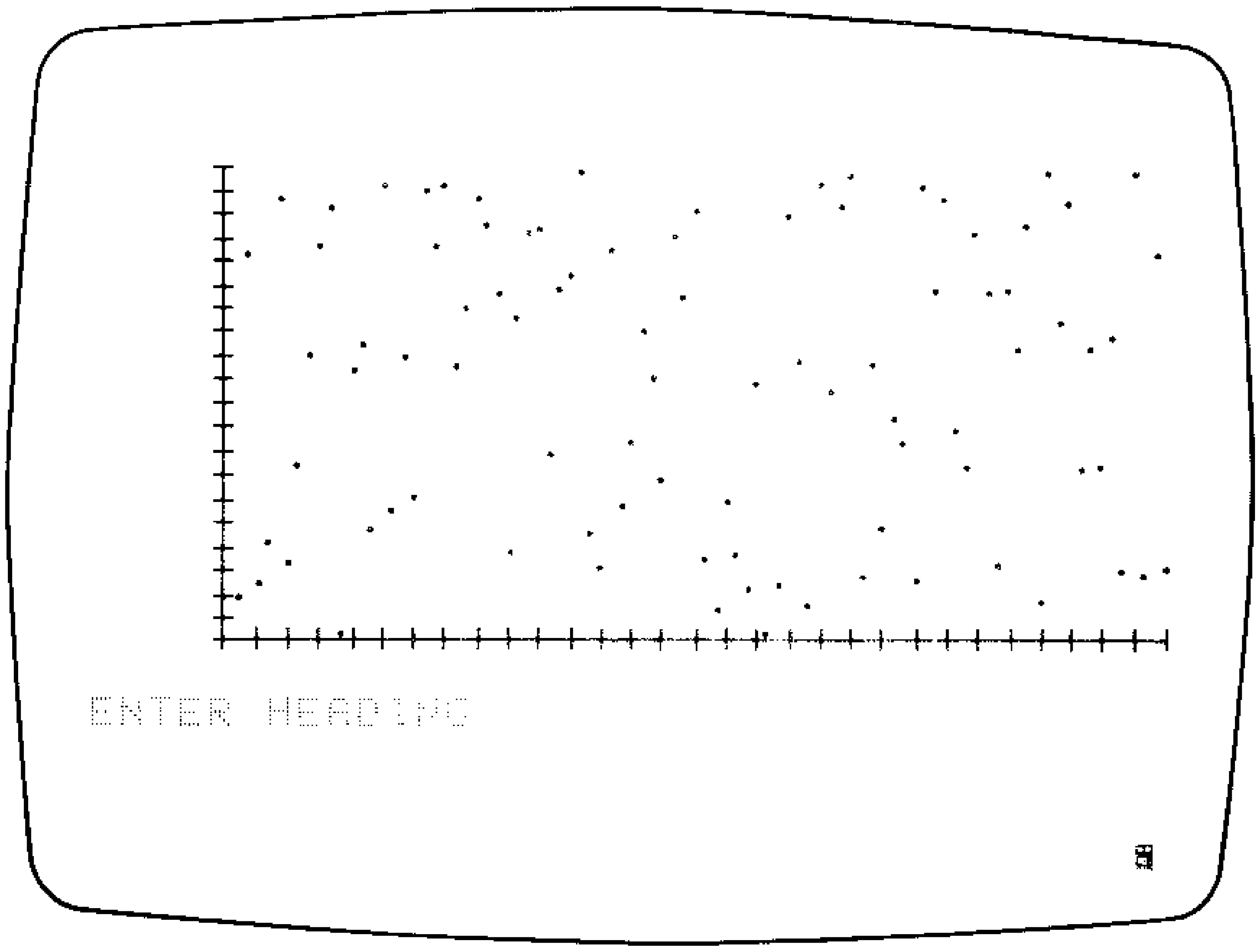
For instance, if you change statement 150 to:

```
150 MOVE 29,-10
```

The first input prompt will be displayed in the lower right corner.

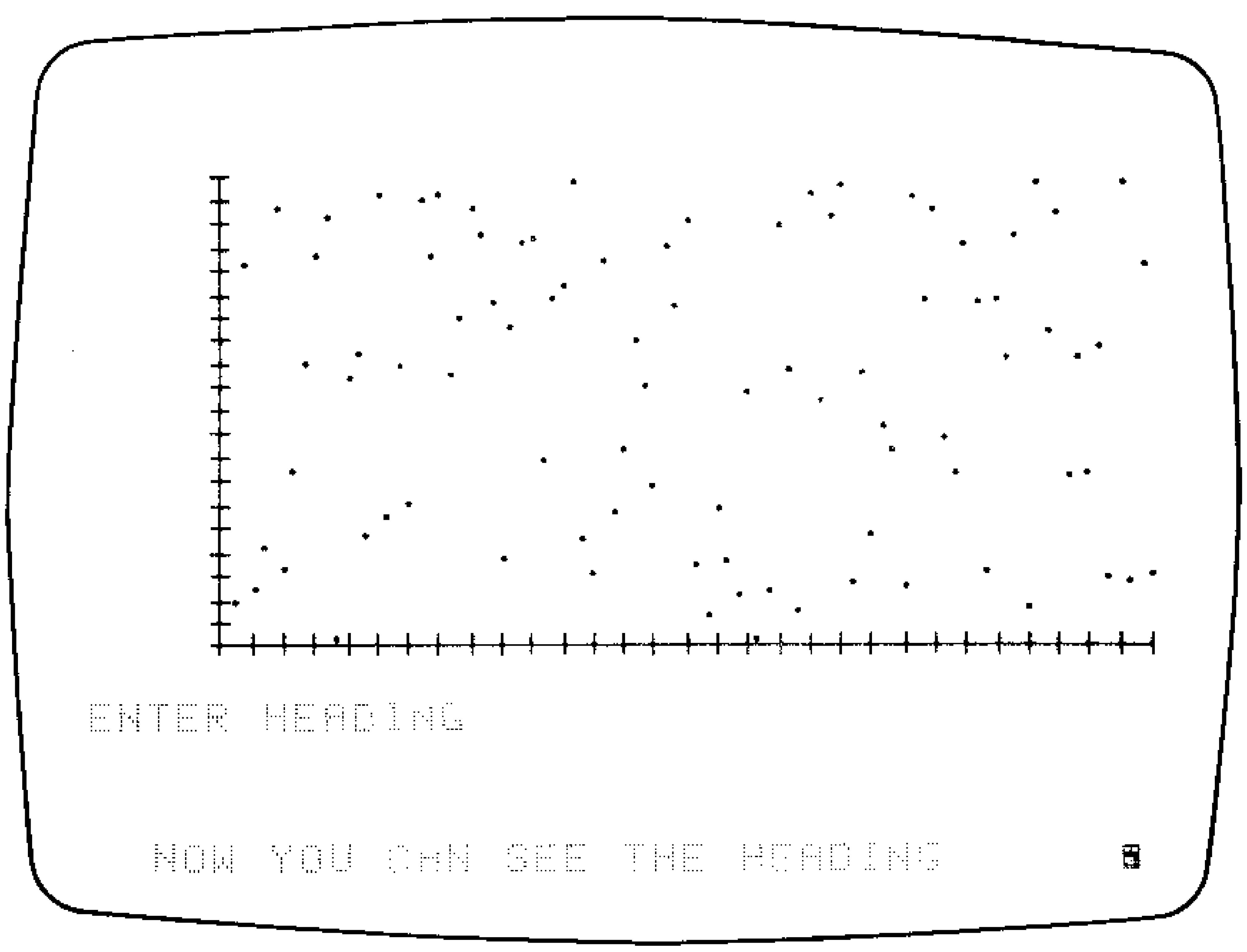


You can enter up to 95 characters (in our program 32 characters) in an input statement, but you won't be able to see what you have entered since the graphics display does not scroll up like the alpha display when characters are typed. Thus, the message would be typed on top of itself in the lower right corner.



Even though you cannot distinguish the characters that have been keyed in, the system remembers up to 95 characters. So, you can still backspace to correct a character if you've made a mistake.

Since there is space on the graphics display to backspace, the system allows you to backspace past the question mark. After you backspace, you can enter the desired input and view the message as you input it.



An INPUT statement in graphics mode always resets the label direction to the horizontal position so that input messages can be read easily. The input prompt, ?, always appears on the graphics display if the CRT is in graphics mode when an INPUT statement is executed. If you really want to input in alpha mode, be sure to execute the ALPHA statement prior to the INPUT statement.

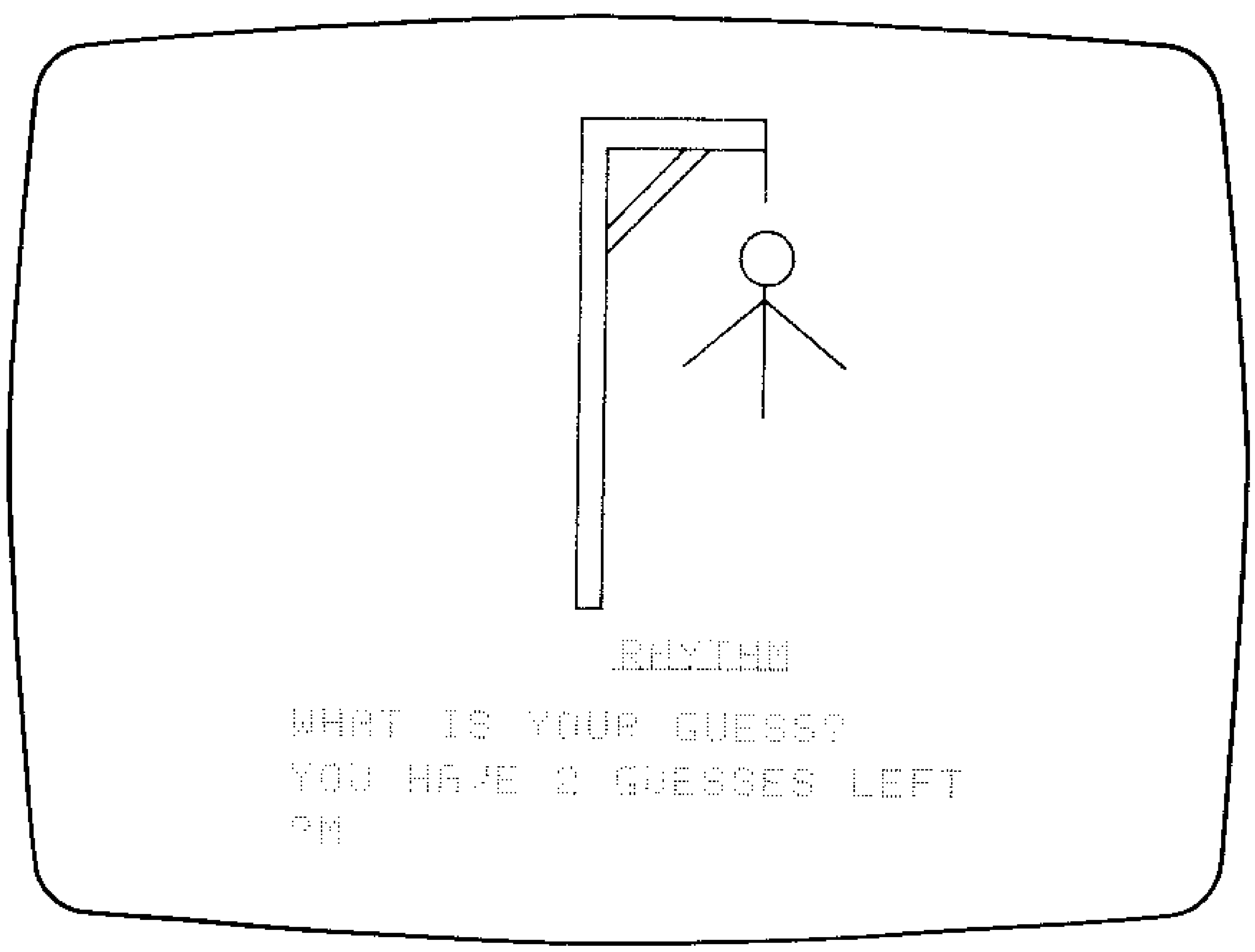
Problem

12.6 "Hangman" is a game commonly played by youngsters (and oldsters, alike) in which one person chooses a word and another must guess it, one letter at a time, given the length of the word. The word-chooser writes a dash to represent each letter in the word. Whenever a letter is guessed correctly, all occurrences of the letter in the word are written above the dash that represents the letter's position in the word. Whenever an incorrect guess is made, a part of the hangman's body is drawn.

If the word is guessed before the hangman is completed, the guesser wins. If the hangman is completed before the word is discovered, the word-chooser wins.

Write a program to simulate this game on the HP-85. So that you do not have to create string data files, write it in such a way that one person inputs a word, the display is cleared, and another person must guess the word. Write one subroutine to draw the scaffold and another subroutine that includes a computed GOTO statement to determine the part of the body that is to be drawn in the case of an incorrect guess. Include six body parts (head, left and right arms, trunk, left and right legs) and allow the guesser six incorrect guesses.

Here's a sample graphics display where the guesser won with two guesses left and the hangman body 2/3 complete.



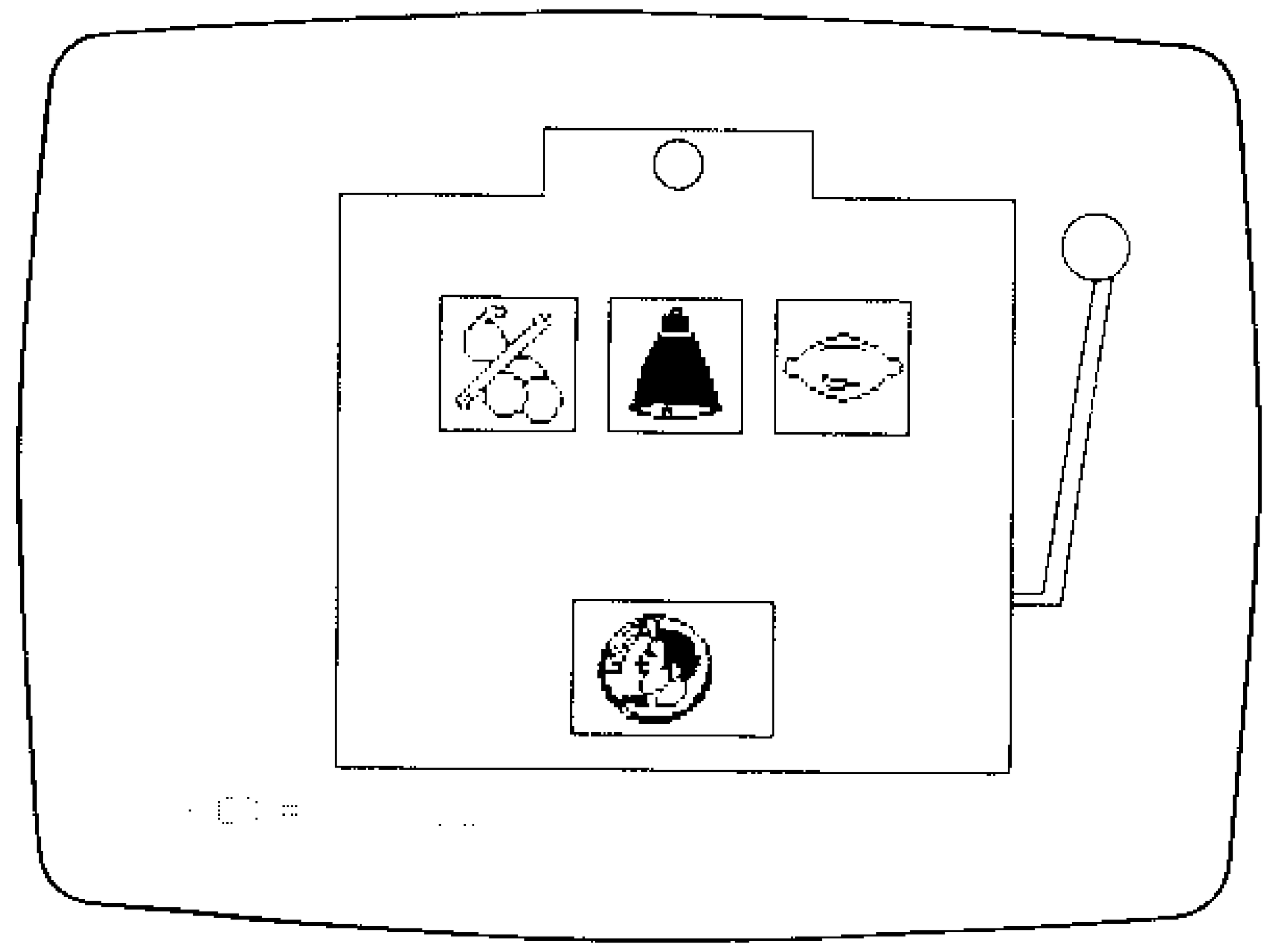
With the hangman program, it is essential to accept inputs in graphics mode.

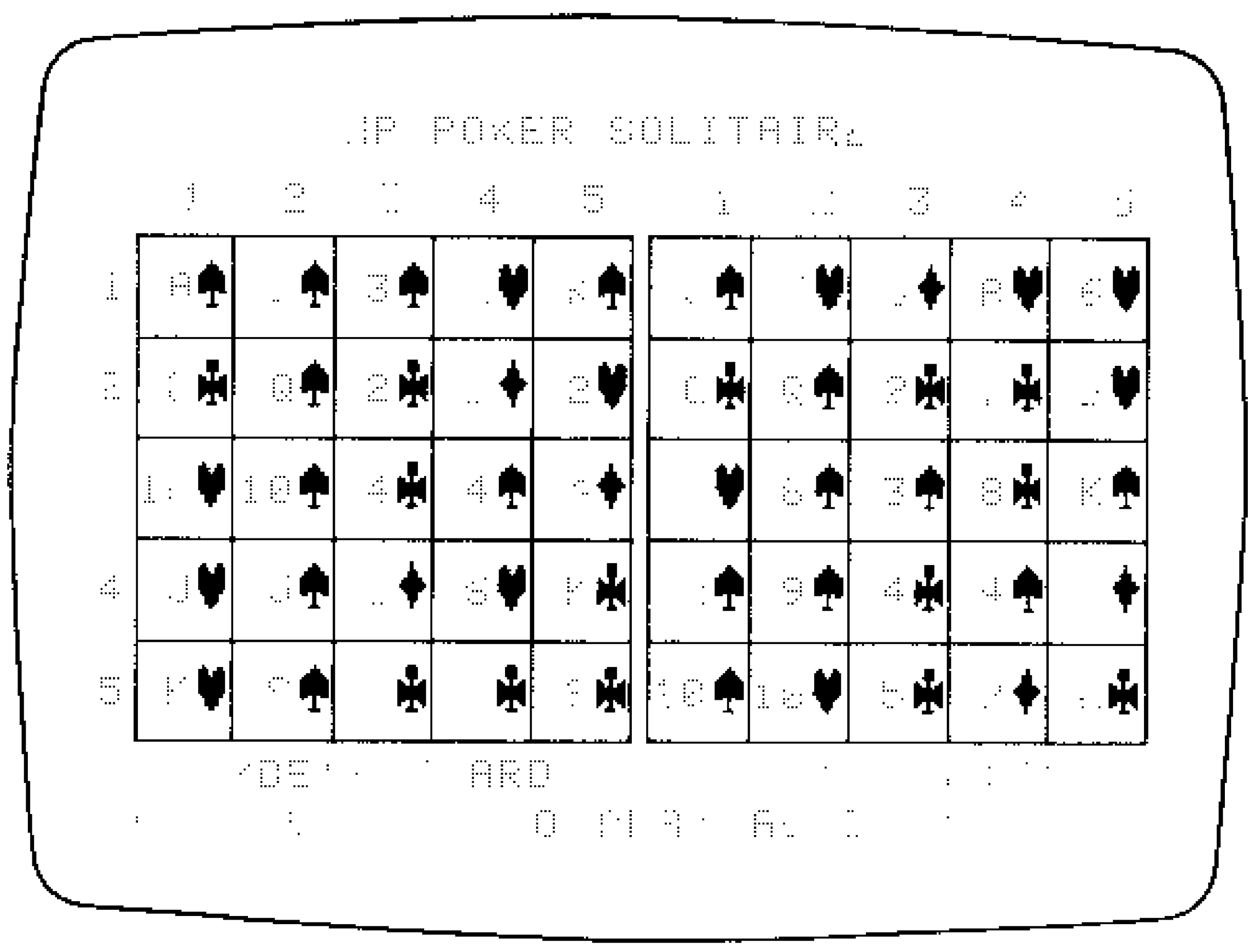
Advanced Plotting With BPLOT

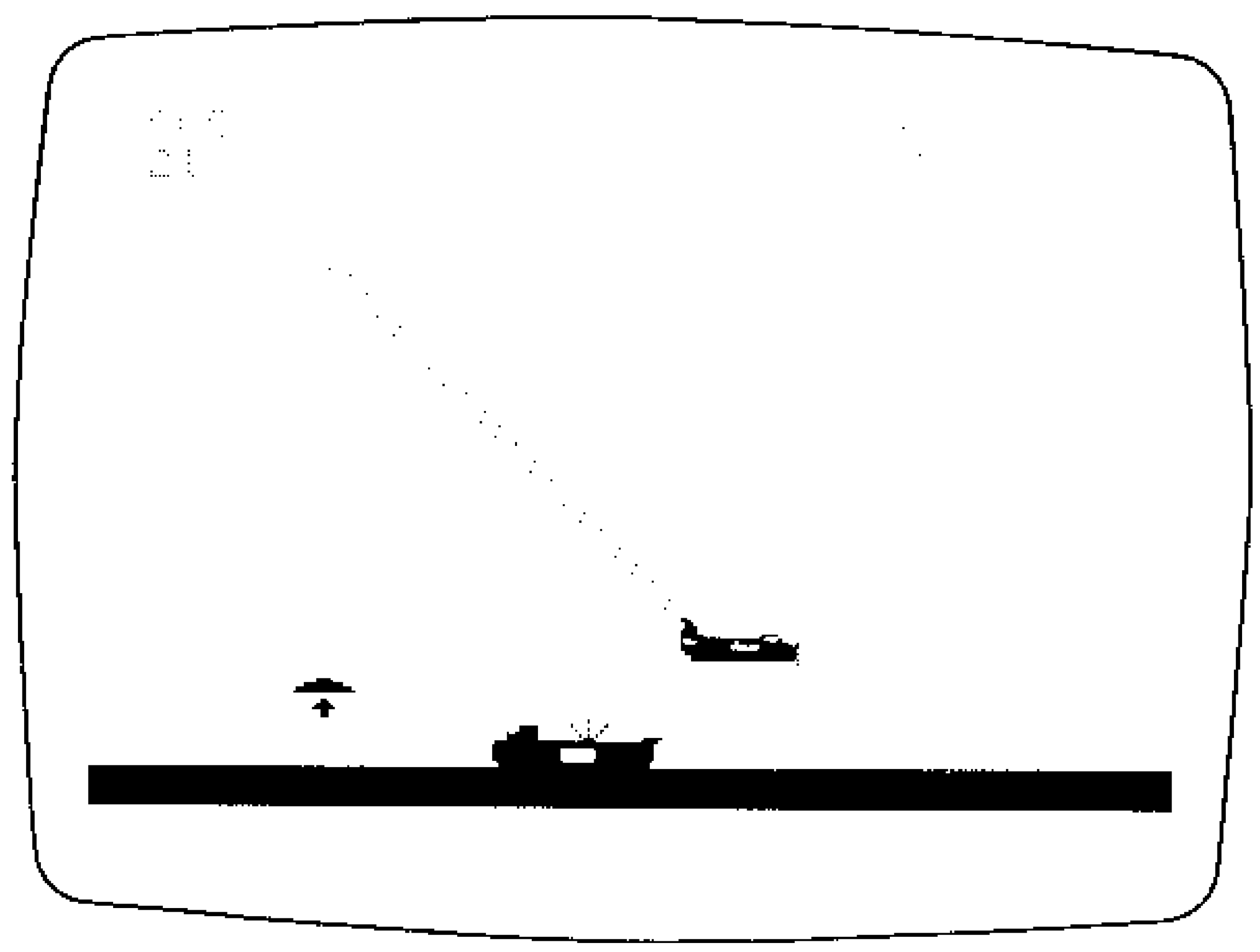
The BPLOT *(byte plot)* statement enables you to plot groups of dots on the graphics display by creating a string of characters that specify those dots. Each character in the string specifies one byte (eight bits or dots) of information which determines whether dots are on or off on the graphics display.

BPLOT string expression , number of characters per row

The BPLOT statement is not difficult to use, but it does take some time to figure out the precise dot configurations of a design or pattern. If you have played any of the games in the HP-85 Games Pac, you've probably seen BPLOT in action. Soon you'll be generating figures like these using BPLOT:







First we will outline the procedure for building a character string for BPLOT, then we will discuss some examples and byte plotting peculiarities.

Procedure for Building the String

1. Draw the figure you wish to plot.
2. Then redraw the figure in matrix form, using dot patterns instead of lines. Graph paper is useful at this point; let each square equal one dot, block, or bit of information.
3. Divide the dot figure into columns of dots and spaces, eight squares wide. View each eight blocks as a byte of information where each block specifies a bit. If a dot is specified, the value of the block is one; if no dot is specified, the block's value is zero. Thus, each group of eight dots or spaces specifies a binary number that determines a particular character.
4. Convert each binary number to its decimal equivalent. This can be done in a variety of ways; the easiest of course, is to use a conversion table. (You can use the table of characters and binary/octal/decimal equivalents in appendix C.) If a table is not available, convert each eight-digit binary number to its three-digit octal equivalent and then convert the octal number to its decimal equivalent (since most of us don't easily convert binary numbers to decimal equivalents). You may wish to use (or modify) the Base Conversions program, listed on page 257.

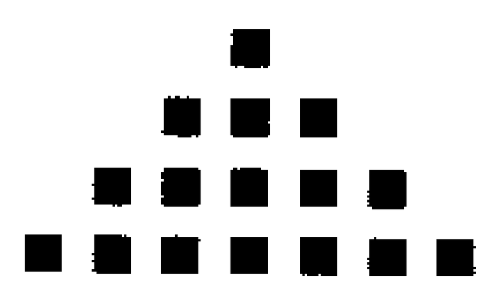
5. Build the character string by assigning the character of the specified decimal value (using the `CHR$` function) to the appropriate character position in the string; the easiest way to build the string is to write a program that accepts and appends the character to the string through `INPUT` statements or `READ` and `DATA` statements.
6. Use this string with the `BPLOT` statement to plot the figure. Examples of the statement are:

`BPLOT T$,1`	Plots T$; 1 character per row of dots.
`BPLOT S$,5`	Plots S$; 5 characters per row of dots.

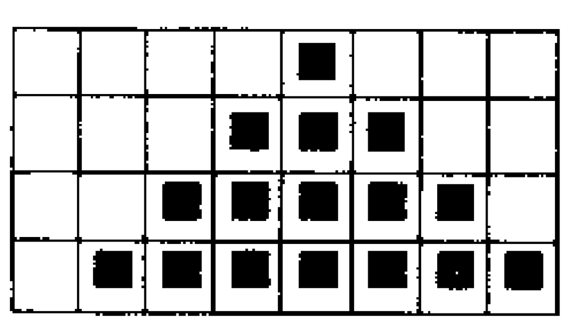
Let's take a simple example to illustrate the first five steps of the procedure. Suppose you wish to plot a solid triangle:

Step 1. Draw the figure.

Step 2. Represent the figure with dots or blocks.

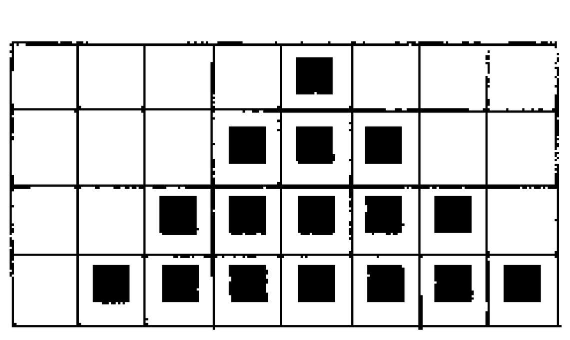


Step 3. Since the base of the triangle is only seven dots wide we need only to place it in a four by eight dot matrix.



Each row of this dot matrix specifies a byte (eight bits) of information.

Step 4. Convert each row of the matrix to a decimal value.



Binary Representation	Octal Value	Decimal Value
0 0 | 0 0 1 | 0 0 0	0 1 0	8
0 0 | 0 1 1 | 1 0 0	0 3 4	2 8
0 0 | 1 1 1 | 1 1 0	0 7 6	6 2
0 1 | 1 1 1 | 1 1 1	1 7 7	1 2 7

Step 5. Build the string using the CHR$ function:

```
T$=CHR$(8)&CHR$(28)&CHR$(62)&CHR
$(127)
T$
△œ>⊢
```

This is a short string so we have manually entered the characters. To see the string, type the variable name, then press END LINE.

You should use the CHR$ function to build this string since some characters cannot be specified from the keyboard (e.g., those with decimal values over 128), and others have special meanings when found in association with strings (e.g., the quotation mark).

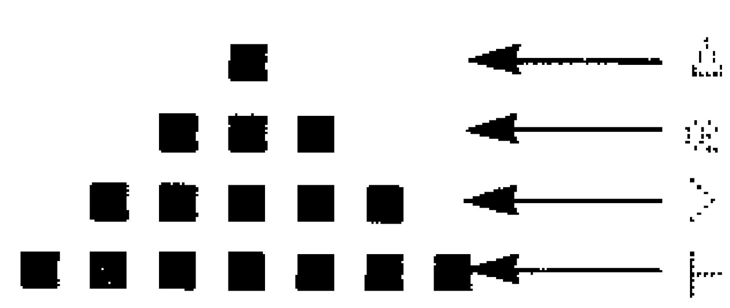
Since T$ is a short string, we built it from the keyboard. With longer strings, you might write a program like this:

```
10 DIM T$[4]
20 FOR I=1 TO 4
30 READ V
40 T$[I,I]=CHR$(V)
50 NEXT I
60 DATA 8,28,62,127
70 END
```

Dimensions the variable.
Uses a FOR-NEXT loop to READ or INPUT the decimal values into the appropriate character position in the string.

Step 6. Use the string with the BPLOT statement to plot the figure. Below we have enlarged the graphics display area around each BPLOT to illustrate the statement. Do not execute these statements now.

`BPLOT T$,1`



Plots one character per line of T$="△œ>⊢", thus producing a triangle.

`BPLOT T$,2`

Plots two characters per line of T$="△œ>⊢".

`BPLOT T$,3`

Plots three characters per line of T$="△œ>⊢".

`BPLOT T$,4`

Plots four characters per line of T$="△œ>⊢".

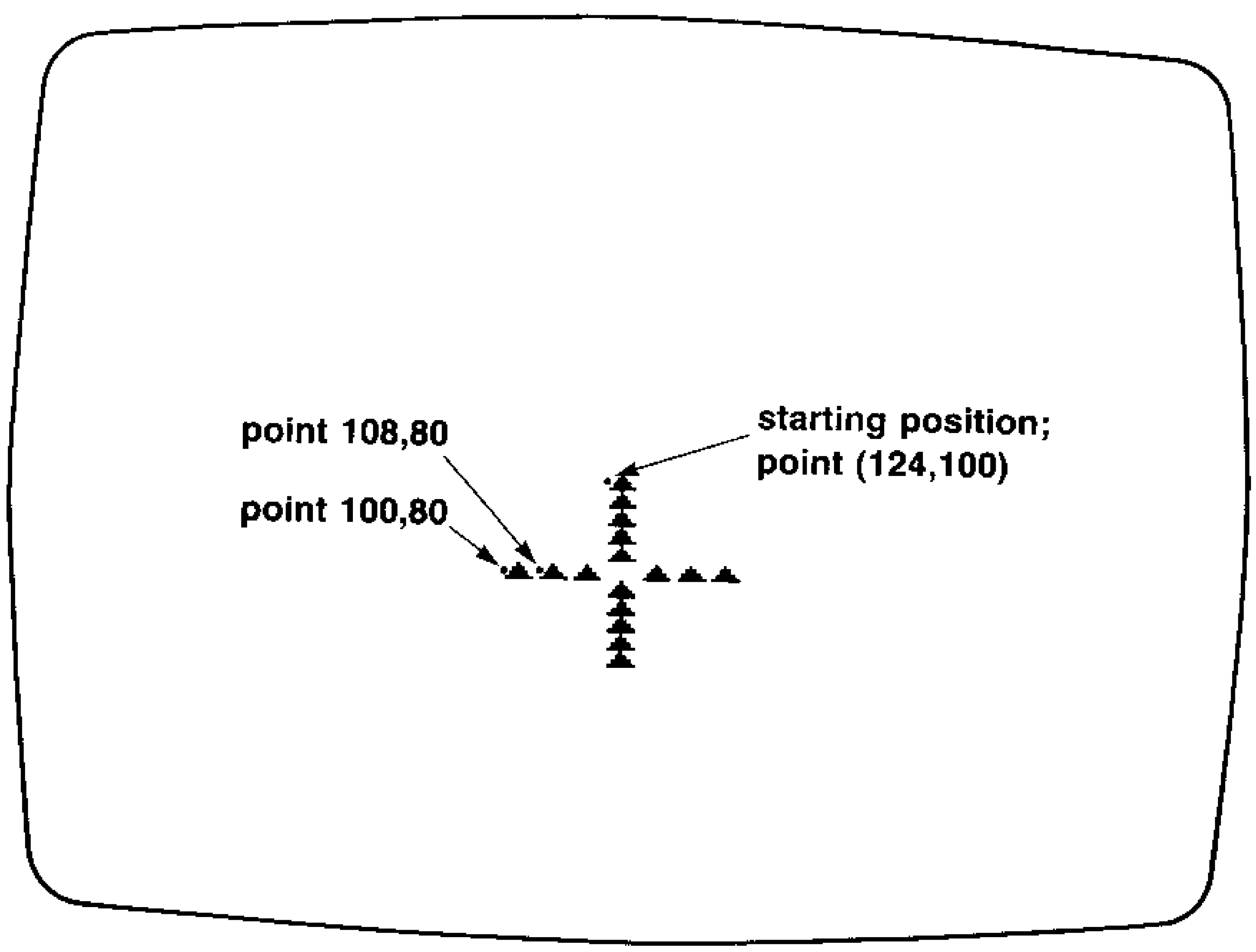
As you can see, BPLOT T$,1 produces a triangle because it plots one character per line. BPLOT T$,4 plots all four characters on the same line.

Using the String With BPLOT

Now that you have composed the string, use the BPLOT statement to plot the figure. Enter and run the following program—use the editing features of your HP-85 to add statements to the last program if you wish and then renumber the program.

```
  10 PEN 1 @ GCLEAR
  20 SCALE 0,255,0,191

  30 FOR I=1 TO 4
  40 READ V
• 50 T$[I,I]=CHR$(V)
  60 NEXT I
  70 MOVE 124,100
  80 FOR I=1 TO 11
• 90 BPLOT T$,1
 100 NEXT I
 110 FOR X=100 TO 148 STEP 8
 120 MOVE X,80
•130 BPLOT T$,1
 140 NEXT X
 150 DATA 8,28,62,127
 160 END
```

Lines	
10	Clears the graphics display.
20	Scales to number of dots on graphics screen.
30–60	Repeats the procedure for building the string.
70	Moves to point 124,100.
80–100	Creates a column of 11 triangles.
110–140	Creates a row of seven triangles.



Note:

1. BPLOT automatically stacks the specified string when only one pen location is specified (see lines 70 through 100 above).
2. BPLOT performs an EXOR (exclusive or) with existing dots on the screen. Thus, we erased the middle triangle by plotting it twice.

The example above illustrates most of the facts you need to know about BPLOT. We enumerate them here:

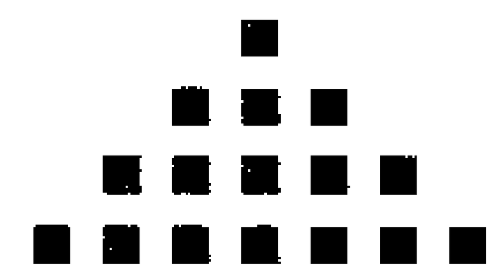
1. For your ease in using BPLOT, scale the display from 0 to 255 (256 dots in the horizontal direction) and from 0 to 191 (192 dots in the vertical direction). With this scale, you always know exactly where the dots will be plotted.

2. The starting position of a byte plot always has an X-coordinate value that is a multiple of four on a horizontal scale of 0 to 255.

   If the current pen location does not have an X-coordinate 0,4,8,...252, the figure will be justified to the nearest four-dot position (multiple of four using the scale above) to the left of the current pen location. The figure is plotted with the upper left corner at the specified pen position. In our example, the statements on the left produced the figure on the right.

```
70 MOVE 124,100
90 BPLOT T$,1
```

point (124,100) ——► •



3. If the BPLOT statement is executed several times without changing the original pen location, the second figure is plotted immediately below the first figure, the third below the second, etc. Notice that lines 70 through 100 produced a column of 11 triangles with the upper left bit of the first triangle at point 124,100. When more than one BPLOT statement is executed, one after the other, it's as if the graphics display performs a carriage return and tabs to the original horizontal position, to begin plotting the figure immediately below the first.
4. BPLOT performs an EXOR (exclusive or) operation between the character string you specify and the existing dots on the display. As you have seen, the middle triangle above was erased because we plotted it twice. Let's discuss how this occurred.

The table below illustrates all possible conditions and outcomes of the EXOR operation between a dot on the screen and the same dot specified by a BPLOT string. The third column gives the resulting dot condition; 0 means the dot is off and 1 means the dot is on.

Dot before BPLOT	Same dot specified by BPLOT string	EXOR: Resultant dot condition	
1 (on)	1	0 (off)	If a dot is on, specifying 1 for the same dot in the BPLOT string turns the dot off.
1 (on)	0	1 (on)	If the dot is on, 0 in the string keeps it on.
0 (off)	1	1 (on)	If the dot is off, 1 turns it on.
0 (off)	0	0 (off)	If the dot is off, 0 keeps it off.

Here's what happened to the middle triangle:

GCLEAR point(124,80)	first BPLOT T$,1 at point (124,80)	display after second BPLOT T$,1 at point (124,80)
0 0 0 0 0 0 0 0	0 0 0 0 1 0 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 0 1 1 1 0 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 0 1 1 1 1 1 0	0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
GCLEAR with PEN 1 turns all dots off.	Since 0EXOR1=1 and 0EXOR0=0; plots triangle.	Since 1EXOR1=0 and 0EXOR0=0; erases triangle.

This aspect of BPLOT becomes very important when you wish to simulate a figure moving across the graphics display. You must know whether a 0 or a 1 will turn the dot on or off of the current display.

Condensing the String Assignment Program

Once you have defined the string, as we have done in the last two programs, you can create one assignment statement that specifies the string for use in future programs. This will shorten your programs by at least four or five statements. More important, for complicated figures, it will eliminate the long set-up time.

For instance, while the last program (page 241) is still in computer memory, type:

```
DISP "30 T$="&CHR$(34)&T$&CHR$(3
4)
```

Execute the statement, by pressing (END LINE), to display:

```
30 T$="△œ>⊢"
```

Now use the (↑) key to move the cursor back to this line and then press (END LINE); the new statement 30 will be stored.

Notice that we have used `CHR$(34)` to specify quotes. If your `BPLOT` character string also contains quotes, you must concatenate them to the string using `CHR$(34)`.

If your `BPLOT` string contains underlined characters (characters with decimal values above 128), you must be careful to avoid the underlined character with the cursor. The cursor will always erase an underline, thus changing the character value.

Since the new statement 30 has been stored, you can delete the unnecessary statements from the program. Do so now by executing:

`DELETE 40,60` (END LINE)	Removes rest of `FOR-NEXT` loop.
`150` (END LINE)	Deletes `DATA` statement.

Renumber the program and list it on the display:

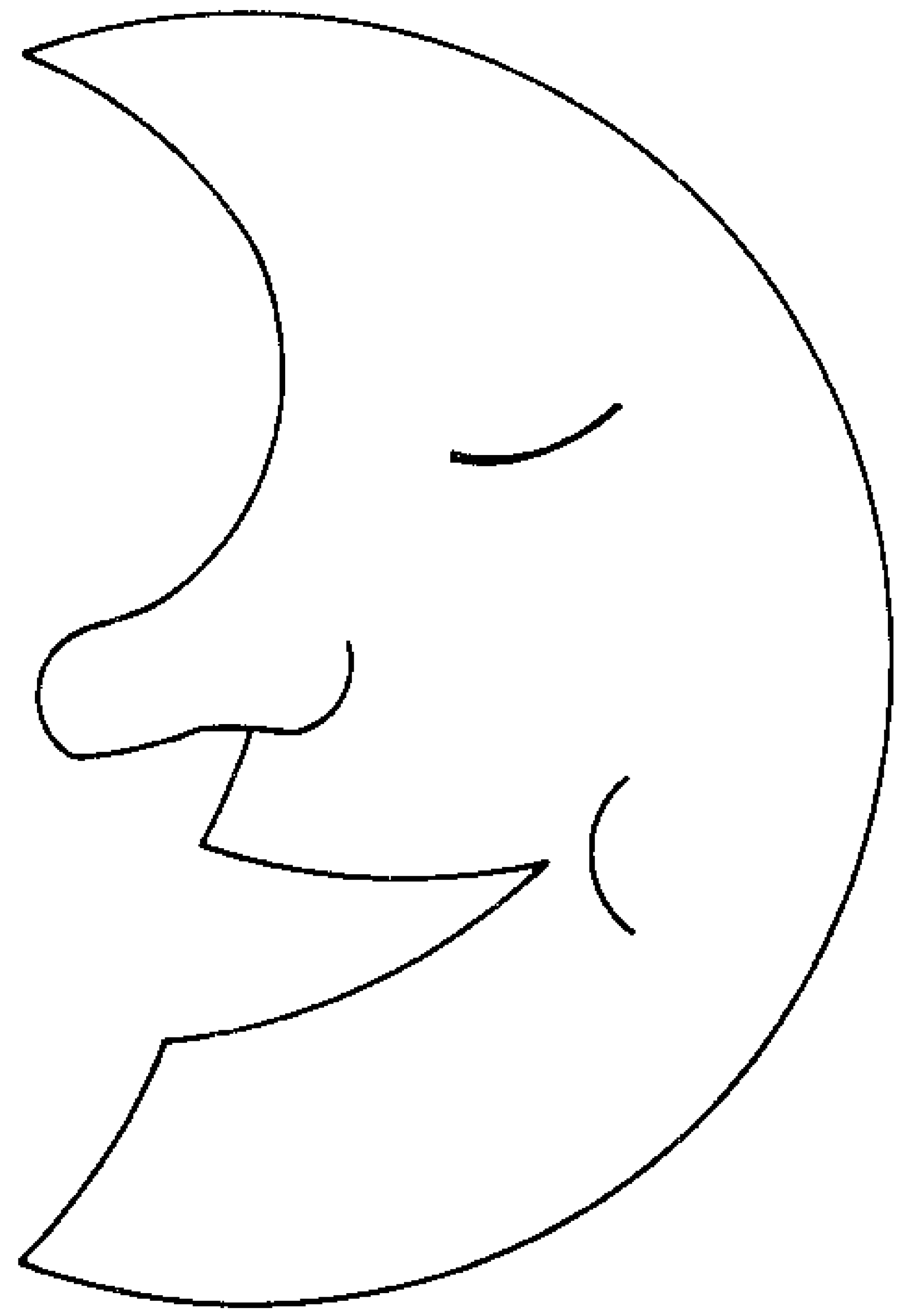
```
10 PEN 1 @ GCLEAR
20 SCALE 0,255,0,191
30 T$="△œ>⊢"
40 MOVE 124,100
50 FOR I=1 TO 11
60 BPLOT T$,1
70 NEXT I
80 FOR X=100 TO 148 STEP 8
90 MOVE X,80
100 BPLOT T$,1
110 NEXT X
120 END
```

This program, shortened by four program lines, performs exactly the same `BPLOT` as the program on page 241. Try it! It will also be faster for longer strings!

Let's look at a more difficult example to illustrate the remaining features of the `BPLOT` statement.

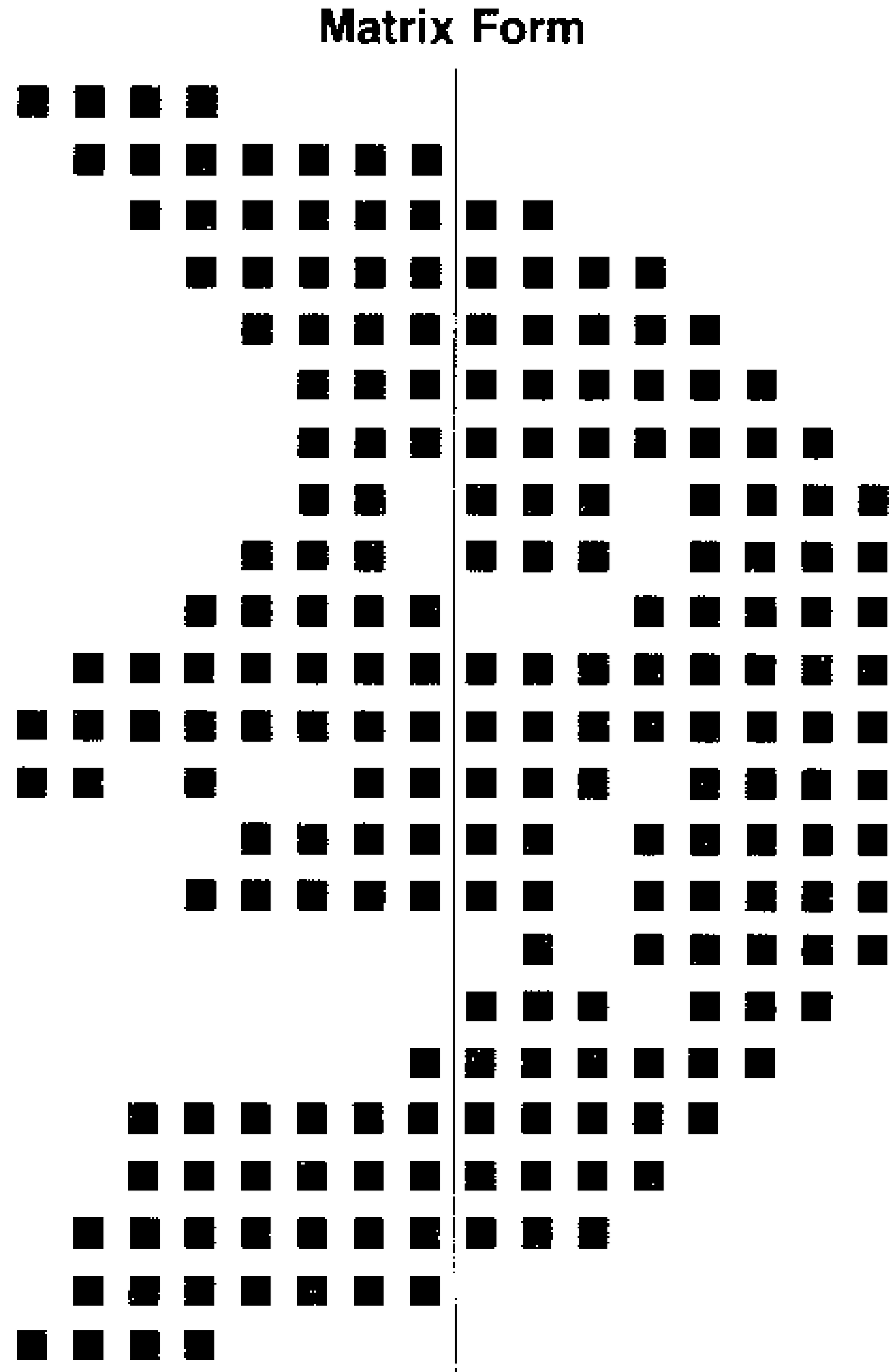
Example: Use `BPLOT` to plot a man in the moon. Then create another `BPLOT` character string to make the man in the moon move across the screen, one byte each move.

1. First draw the figure:



2. Now redraw the figure on graph paper in matrix form. Since the moon is white, we want to light all of the dots inside the figure except the lines drawn for the eye, nose, and cheek.

Matrix Form



Binary Representation

1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0
0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0
0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0
0	0	0	0	0	1	1	0	1	1	1	0	1	1	1	1
0	0	0	0	1	1	1	0	1	1	1	0	1	1	1	1
0	0	0	1	1	1	1	1	0	0	0	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	1	0	0	1	1	1	1	1	0	1	1	1	1
0	0	0	0	1	1	1	1	1	1	0	1	1	1	1	1
0	0	0	1	1	1	1	1	1	1	0	1	1	1	1	1
0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1
0	0	0	0	0	0	0	0	1	1	1	0	1	1	1	0
0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0
0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0
0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

3. Divide the figure into columns of dots and spaces, eight squares wide. Our figure is 16 squares wide, so we divided it into two columns. Each line of each column represents one byte of information. Above, we also converted the figure to its binary representation. For the two columns of eight squares, we also have two columns of eight-digit binary numbers.

4. Convert each eight-digit binary number to its decimal equivalent.

Binary Representation		Octal Representation		Decimal Value	
1 1 1 1 0 0 0 0	0 0 0 0 0 0 0 0	3 6 0	0 0 0	2 4 0	0
0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0	1 7 7	0 0 0	1 2 7	0
0 0 1 1 1 1 1 1	1 1 0 0 0 0 0 0	0 7 7	3 0 0	6 3	1 9 2
0 0 0 1 1 1 1 1	1 1 1 1 0 0 0 0	0 3 7	3 6 0	3 1	2 4 0
0 0 0 0 1 1 1 1	1 1 1 1 1 0 0 0	0 1 7	3 7 0	1 5	2 4 8
0 0 0 0 0 1 1 1	1 1 1 1 1 1 0 0	0 0 7	3 7 4	7	2 5 2
0 0 0 0 0 1 1 1	1 1 1 1 1 1 1 0	0 0 7	3 7 6	7	2 5 4
0 0 0 0 0 1 1 0	1 1 1 0 1 1 1 1	0 0 6	3 5 7	6	2 3 9
0 0 0 0 1 1 1 0	1 1 1 0 1 1 1 1	0 1 6	3 5 7	1 4	2 3 9
0 0 0 1 1 1 1 1	0 0 0 1 1 1 1 1	0 3 7	0 3 7	3 1	3 1
0 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 7 7	3 7 7	1 2 7	2 5 5
1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	3 7 7	3 7 7	2 5 5	2 5 5
1 1 0 1 0 0 1 1	1 1 1 0 1 1 1 1	3 2 3	3 5 7	2 1 1	2 3 9
0 0 0 0 1 1 1 1	1 1 0 1 1 1 1 1	0 1 7	3 3 7	1 5	2 2 3
0 0 0 1 1 1 1 1	1 1 0 1 1 1 1 1	0 3 7	3 3 7	3 1	2 2 3
0 0 0 0 0 0 0 0	0 1 0 1 1 1 1 1	0 0 0	1 3 7	0	9 5
0 0 0 0 0 0 0 0	1 1 1 0 1 1 1 0	0 0 0	3 5 6	0	2 3 8
0 0 0 0 0 0 0 1	1 1 1 1 1 1 0 0	0 0 1	3 7 4	1	2 5 2
0 0 1 1 1 1 1 1	1 1 1 1 1 0 0 0	0 7 7	3 7 0	6 3	2 4 8
0 0 1 1 1 1 1 1	1 1 1 1 0 0 0 0	0 7 7	3 6 0	6 3	2 4 0
0 1 1 1 1 1 1 1	1 1 1 0 0 0 0 0	1 7 7	3 4 0	1 2 7	2 2 4
0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0	1 7 7	0 0 0	1 2 7	0
1 1 1 1 0 0 0 0	0 0 0 0 0 0 0 0	3 6 0	0 0 0	2 4 0	0

5. Build the character string using the CHR$ function.

```
10 REM *BUILD MOON STRING*
20 DIM M$[46]

30 FOR I=1 TO 46
40 READ M1
50 M$[I,I]=CHR$(M1)
60 NEXT I
70 DATA 240,0,127,0,63,192,31,2
   40,15,248,7,252,7,254,6,239,
   14,239,31,31,127,255,255,255
   ,211,239
80 DATA 15,223,31,223,0,95,0,23
   8,1,252,63,248,63,240,127,22
   4,127,0,240,0
90 END
```

Dimensions string to number of decimal values.

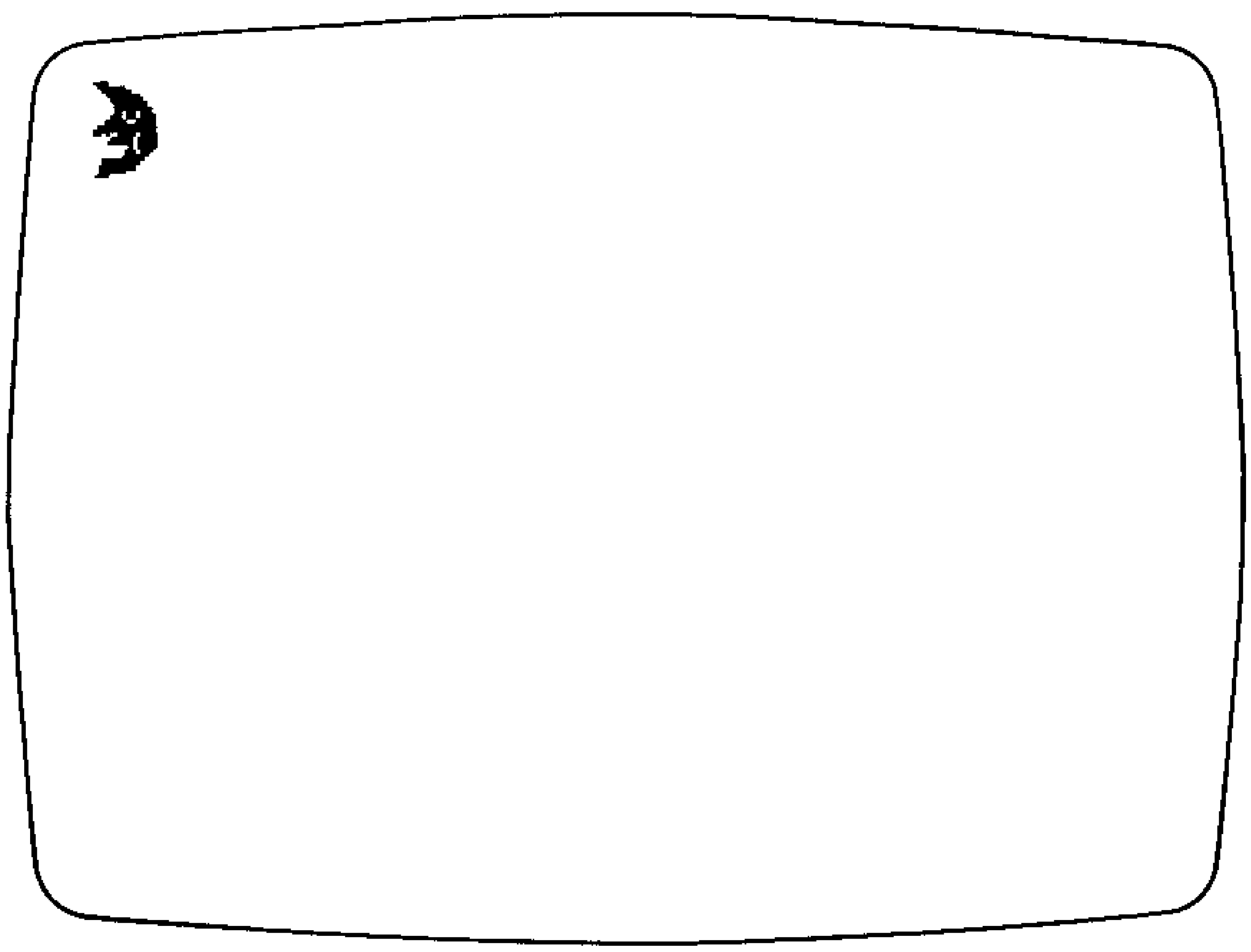
Uses FOR-NEXT loop to READ or INPUT the decimal values and assigns them to the appropriate position in the character string.

Data for moon string read from decimal value table from left to right.

6. Use this string with the BPLOT statement to plot the man in the moon. Append the following statements to the end of the string building program above and then press (RUN).

```
 90 SCALE 0,255,0,191
100 PEN 1 @ GCLEAR
110 MOVE 0,191
120 BPLOT M$,2
130 END
```

Replace END statement with SCALE statement.
Moves to upper left corner of display.
BPLOT the character string, two characters per line.



As you can see, the man in the moon was plotted once in the upper left corner of the graphics display.

To finish the solution to the example, we must move the man in the moon across the display one byte (eight dots) at a time.

What happens when we simply position the pen to point 8,191—eight dots from the original starting position, and then execute BPLOT M$,2 once again? Try it!

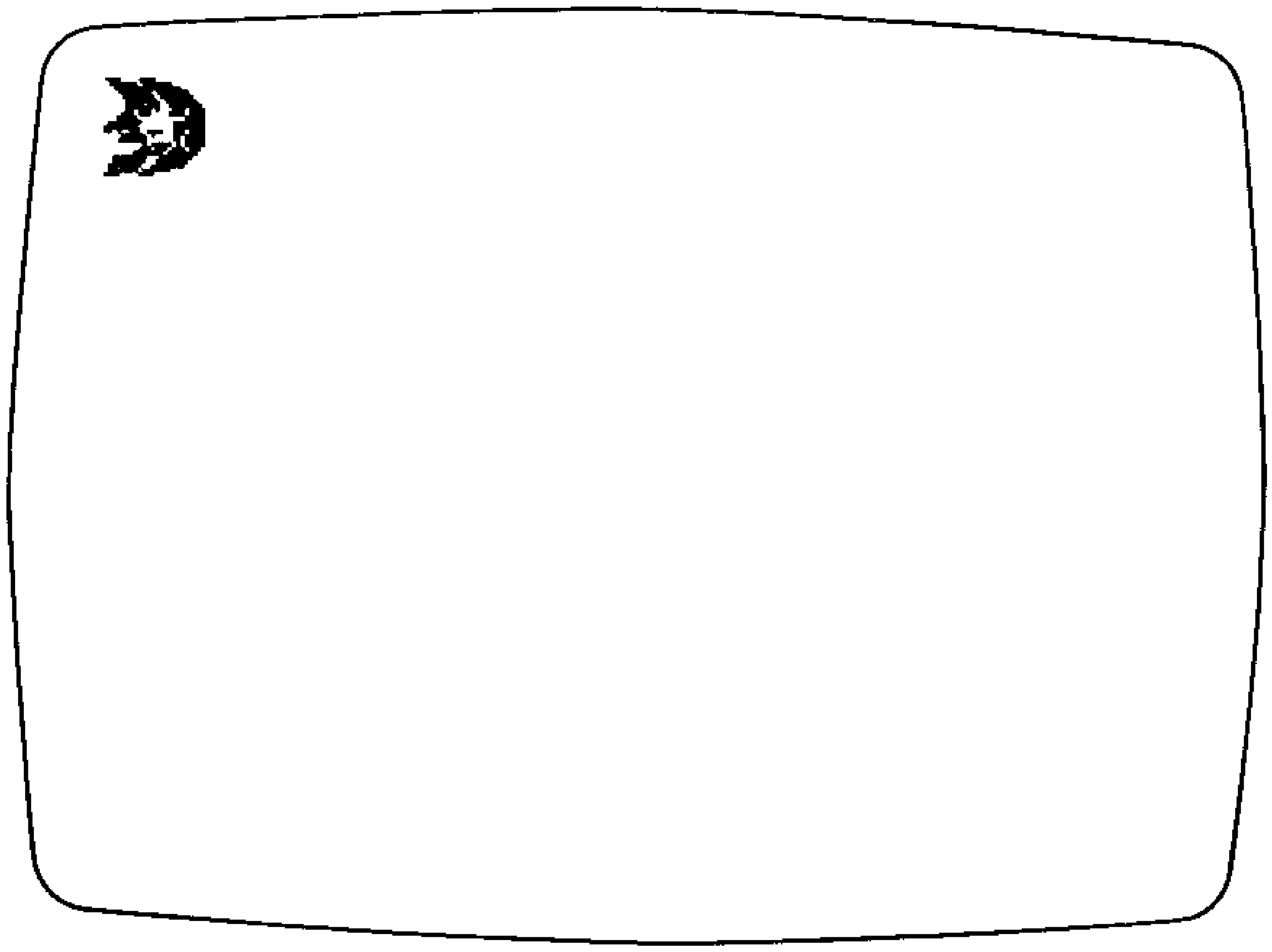
```
MOVE 8,191
```

Moves to point 8,191. Remember the system reverts to alpha mode as you type a statement and reverts back to graphics mode when you execute it.

```
BPLOT M$,2
```

Byte plots the character string for the moon, two characters per line.

After you execute the BPLOT statement, the display shows:



As you can see, BPLOT performs an EXOR operation with existing dots on the graphics display. So, the left half of the first moon remains intact, but the right half of the first moon and the left half of the second moon leave an odd dot configuration on the display. Since the display was clear to begin with (aside from the first moon), the right half of the second moon is plotted correctly.

This should give you an idea of what we must do in order to simulate the moon moving across the display. We must create another character string for BPLOT—three bytes (characters) wide. The first character should erase the left half of the first moon, the second character should plot the left half of the second moon when it is plotted on top of the right half of the first moon, and the third character should plot the right half of the second moon.

The first and third characters are easy enough to compute. Since BPLOT performs an EXOR with existing dots on the display, the first character of each line of our new BPLOT string is the same as the first character of the original string; 1EXOR1=0 and 0EXOR0=0. The third character of each line of the new string is the same as the second character of each line in the original string; 0EXOR1=1 and 0EXOR0=0.

The middle character of each line of our new BPLOT must be computed such that it produces the left half of the moon. Since it is plotted on top of the first moon, you must specify the bit value 0 or 1 so that when an EXOR is performed, you obtain the desired result.

If a dot is on and you want it off, specify 1.
If a dot is off and you want it on, specify 1.
If a dot is on and you want it on, specify 0.
If a dot is off and you want it off, specify 0.

In other words, the middle character is an EXOR between the first half and the second half of the original moon.

Binary Representation of New Moon

Left Half	EXOR	Right Half
1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 0	0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1	0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 0 0 0 0 0 0
0 0 0 1 1 1 1 1	1 1 1 0 1 1 1 1	1 1 1 1 0 0 0 0
0 0 0 0 1 1 1 1	1 1 1 1 0 1 1 1	1 1 1 1 1 0 0 0
0 0 0 0 0 1 1 1	1 1 1 1 1 0 1 1	1 1 1 1 1 1 0 0
0 0 0 0 0 1 1 1	1 1 1 1 1 0 0 1	1 1 1 1 1 1 1 0
0 0 0 0 0 1 1 0	1 1 1 0 1 0 0 1	1 1 1 0 1 1 1 1
0 0 0 0 1 1 1 0	1 1 1 0 0 0 0 1	1 1 1 0 1 1 1 1
0 0 0 1 1 1 1 1	0 0 0 0 0 0 0 0	0 0 0 1 1 1 1 1
0 1 1 1 1 1 1 1	1 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1
1 1 0 1 0 0 1 1	0 0 1 1 1 1 0 0	1 1 1 0 1 1 1 1
0 0 0 0 1 1 1 1	1 1 0 1 0 0 0 0	1 1 0 1 1 1 1 1
0 0 0 1 1 1 1 1	1 1 0 0 0 0 0 0	1 1 0 1 1 1 1 1
0 0 0 0 0 0 0 0	0 1 0 1 1 1 1 1	0 1 0 1 1 1 1 1
0 0 0 0 0 0 0 0	1 1 1 0 1 1 1 0	1 1 1 0 1 1 1 0
0 0 0 0 0 0 0 1	1 1 1 1 1 1 0 1	1 1 1 1 1 1 0 0
0 0 1 1 1 1 1 1	1 1 0 0 0 1 1 1	1 1 1 1 1 0 0 0
0 0 1 1 1 1 1 1	1 1 0 0 1 1 1 1	1 1 1 1 0 0 0 0
0 1 1 1 1 1 1 1	1 0 0 1 1 1 1 1	1 1 1 0 0 0 0 0
0 1 1 1 1 1 1 1	0 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 0	0 0 0 0 0 0 0 0

The binary numbers in this column are the same as the binary numbers in the first column of the original moon.

The binary numbers in the middle column are the results of the EXOR operation performed between digits of the left half and the corresponding digits of the right half of the moon.

The binary numbers in this column are the same as the binary numbers in the second column of the original moon.

We already know the decimal values of the first and third column. Now find the decimal values of the numbers in the middle column.

Binary Representation	Octal Representation	Decimal Value
1 1 1 1 0 0 0 0	3 6 0	2 4 0
0 1 1 1 1 1 1 1	1 7 7	1 2 7
1 1 1 1 1 1 1 1	3 7 7	2 5 5
1 1 1 0 1 1 1 1	3 5 7	2 3 9
1 1 1 1 0 1 1 1	3 6 7	2 4 7
1 1 1 1 1 0 1 1	3 7 3	2 5 1
1 1 1 1 1 0 0 1	3 7 1	2 4 9
1 1 1 0 1 0 0 1	3 5 1	2 3 3
1 1 1 0 0 0 0 1	3 4 1	2 2 5
0 0 0 0 0 0 0 0	0 0 0	0
1 0 0 0 0 0 0 0	2 0 0	1 2 8
0 0 0 0 0 0 0 0	0 0 0	0
0 0 1 1 1 1 0 0	0 7 4	6 0
1 1 0 1 0 0 0 0	3 2 0	2 0 8
1 1 0 0 0 0 0 0	3 0 0	1 9 2
0 1 0 1 1 1 1 1	1 3 7	9 5
1 1 1 0 1 1 1 0	3 5 6	2 3 8
1 1 1 1 1 1 0 1	3 7 5	2 5 3
1 1 0 0 0 1 1 1	3 0 7	1 9 9
1 1 0 0 1 1 1 1	3 1 7	2 0 7
1 0 0 1 1 1 1 1	2 3 7	1 5 9
0 1 1 1 1 1 1 1	1 7 7	1 2 7
1 1 1 1 0 0 0 0	3 6 0	2 4 0

Thus, the decimal values for our second BPLOT character string are:

	Decimal Values	
1st Character	2nd Character	3rd Character
240	240	0
127	127	0
63	255	192
31	239	240
15	247	248
7	251	252
7	249	254
6	233	239
14	225	239
31	0	31
127	128	255
255	0	255
211	60	239
15	208	223
31	192	223
0	95	95
0	238	238
1	253	252
63	199	248
63	207	240
127	159	224
127	127	0
240	240	0

Finally, build the character string for the second moon. With the previous moon program still intact in computer memory, make the following changes:

1. Dimension the second moon for 69 characters in statement 20.
2. Add statements 125 through 200, below, to build the second string, and statements 210 to 9999 to plot the string.

```
 10 REM *BUILD MOON STRING*
 20 DIM M$[46],M2$[69]
 30 FOR I=1 TO 46
 40 READ M1
• 50 M$[I,I]=CHR$(M1)
 60 NEXT I
 70 DATA 240,0,127,0,63,192,31,2
    40,15,248,7,252,7,254,6,238,
    14,239,31,31,127,255,255,255
    ,211,239
 80 DATA 15,223,31,223,0,95,0,23
    8,1,252,63,248,63,240,127,22
    4,127,0,240,0
 90 SCALE 0,255,0,191
100 PEN 1 @ GCLEAR
110 MOVE 0,191
•120 BPLOT M$,2
125 REM *BUILD 2nd MOON*
130 FOR K=1 TO 69
140 READ M2
```

Dimensions the string variable for the second moon, M2$.

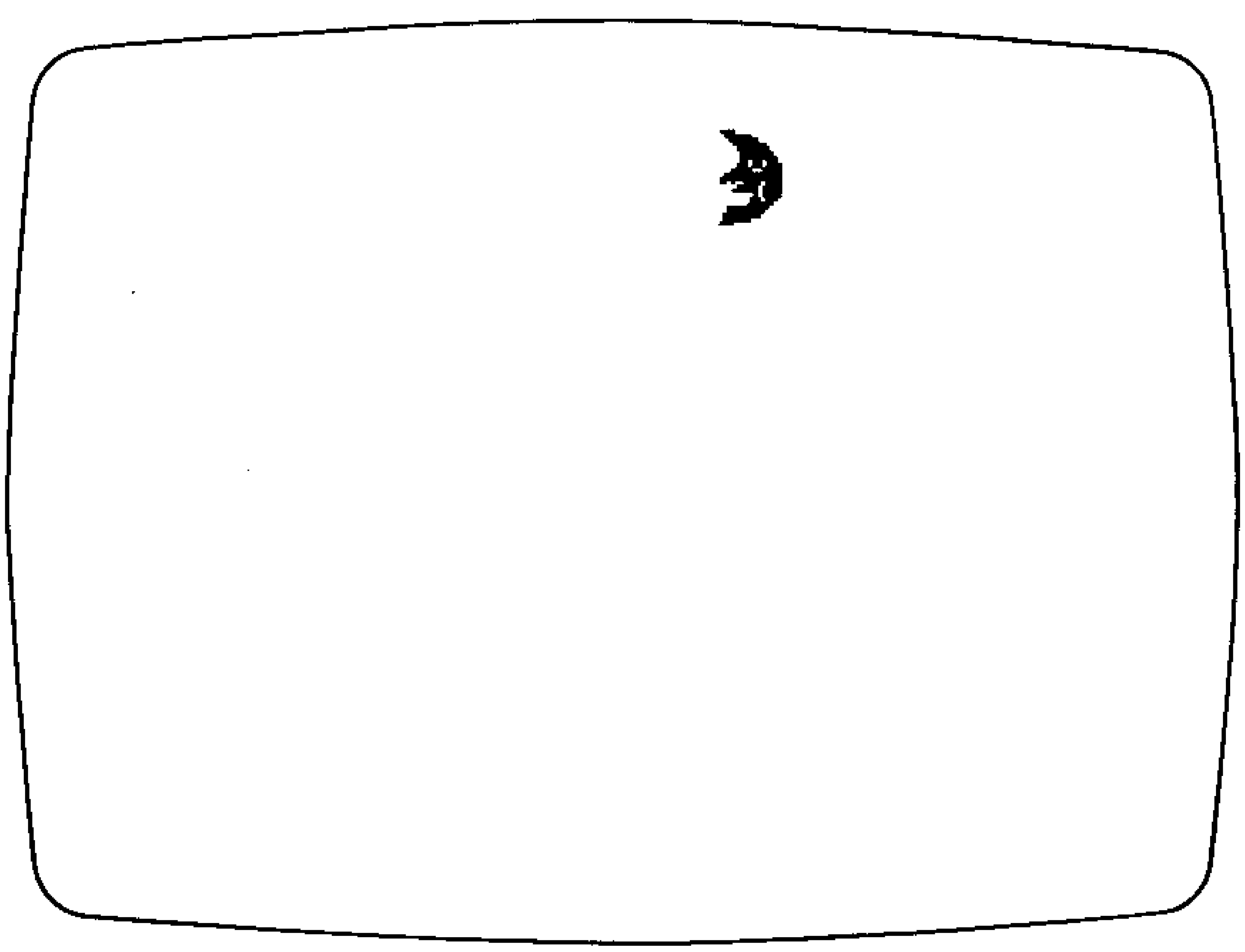
Builds the second string using DATA statements 170 through 200.

```
150 M2$[K,K]=CHR$(M2)
160 NEXT K
170 DATA 240,240,0,127,127,0,63,
    255,192,31,239,240,15,247,24
    8,7,251,252
180 DATA 7,249,254,6,233,239,14,
    225,239,31,0,31,127,128,255,
    255,0,255,211,60,239
190 DATA 15,208,223,31,192,223,0
    ,95,95,0,238,238,1,253,252,6
    3,199,248
200 DATA 63,207,240,127,159,224,
    127,127,0,240,240,0
210 REM *NOW SET UP LOOPS SO THE
        MOON MOVES FROM LEFT
        TO RIGHT ACROSS DISPLAY
220 REM AND FROM TOP TO BOTTOM
        RIGHT CORNER*
230 FOR Y=191 TO 0 STEP -1

240 FOR X=0 TO 255 STEP 8
250 MOVE X,Y
260 BPLOT M2$,3
270 WAIT 1000
280 NEXT X
290 NEXT Y
9999 END
```

Outer loop (Y) moves string from top to bottom.
Inner loop (X) moves string from left to right.

Wait statement pauses to display the moon in its current location.

Now run the program to see the man in the moon move across the display from left to right.



We chose to move the moon eight dots at a time in the horizontal direction. But we could have moved any number of dots at a time. For instance, if we had moved it four dots at a time, the outer four squares of the second moon would be the same as the first. But each of the middle four dots would have to be computed as the EXOR of that set of four dots and the four dots preceding it.

You can stop the moon program anytime by pressing (PAUSE).

Finally, you may wish to condense the moon program in the same manner that we condensed the triangle program earlier.

Since you have just executed the moon program, variable M$ contains character data to build the first moon and variable M2$ contains character data to build the moving moon.

After pressing (PAUSE) to stop the program, create one assignment statement for each moon variable.

Create an assignment statement for M$ by executing:

```
DISP "1000 M$="&CHR$(34)&M$&CHR$
(34)
```

When you execute this statement, an assignment statement numbered 1000 will appear on the display.

Do not enter the new assignment statement into computer memory until you have displayed the characters of each variable you wish to enter. In other words, do not press (END LINE) after statement 1000—the statement you've just created—now. Doing so would deallocate all program variables so that M2$ would be undefined once again.

First, create an assignment statement for M2$ by executing:

```
DISP "2000 M2$="&CHR$(34)&M2$&CH
R$(34)
```

When you executed the statements above, the system displayed the character strings composing each variable. Now you must enter the program lines that you have created into computer memory. Many of the characters are underlined, so the best way to approach the statement with the cursor is from the top.

Thus, press the (↖) key so that the cursor moves directly to the "home" position of the display. Then, continue to press the (↓) key until the cursor rests under 1000 in the statement you just created. Then press (END LINE). Now move the cursor with the (↓) key until it rests under 2000 in the second statement you created and press (END LINE).

```
DISP "1000 M$="&CHR$(34)&M$&CHR$
(34)
1000 M$="ᴇ╣├╣?@▒ᴇ‡xΠ⊥Π∑Γoτo▒▒├├├
├So‡_▒_╣_╣n¿⊥?x?ᴇ├`├╣ᴇ╣"
```

Move the cursor here, then press (END LINE).

```
DISP "2000 M2$="&CHR$(34)&M2$&CH
R$(34)
2000 M2$="ᴇᴇ╣├├╣?├@▒oᴇ‡wxΠπ⊥Πx∑Γ
⊥oτao▒╣▒├╣├├╣├S<o‡P_▒@_╣__╣nn¿÷⊥
?Gx?Dᴇ├▒`├├╣ᴇᴇ╣"
```

Then move the cursor here and press (END LINE).

Since you have now added variable assignment statements for M$ and M2$ to the program, you can delete all of the READ and DATA statements. Execute:

```
DELETE 30,80
DELETE 125,210
```

Now add the following lines to your program:

`115 GOSUB 1000`	Go to 1000 to assign values to variables m$ and M2$.
`300 STOP`	Add `STOP` so that you do not inadvertantly access the subroutine.
`3000 RETURN`	Add return statement at end of subroutine.

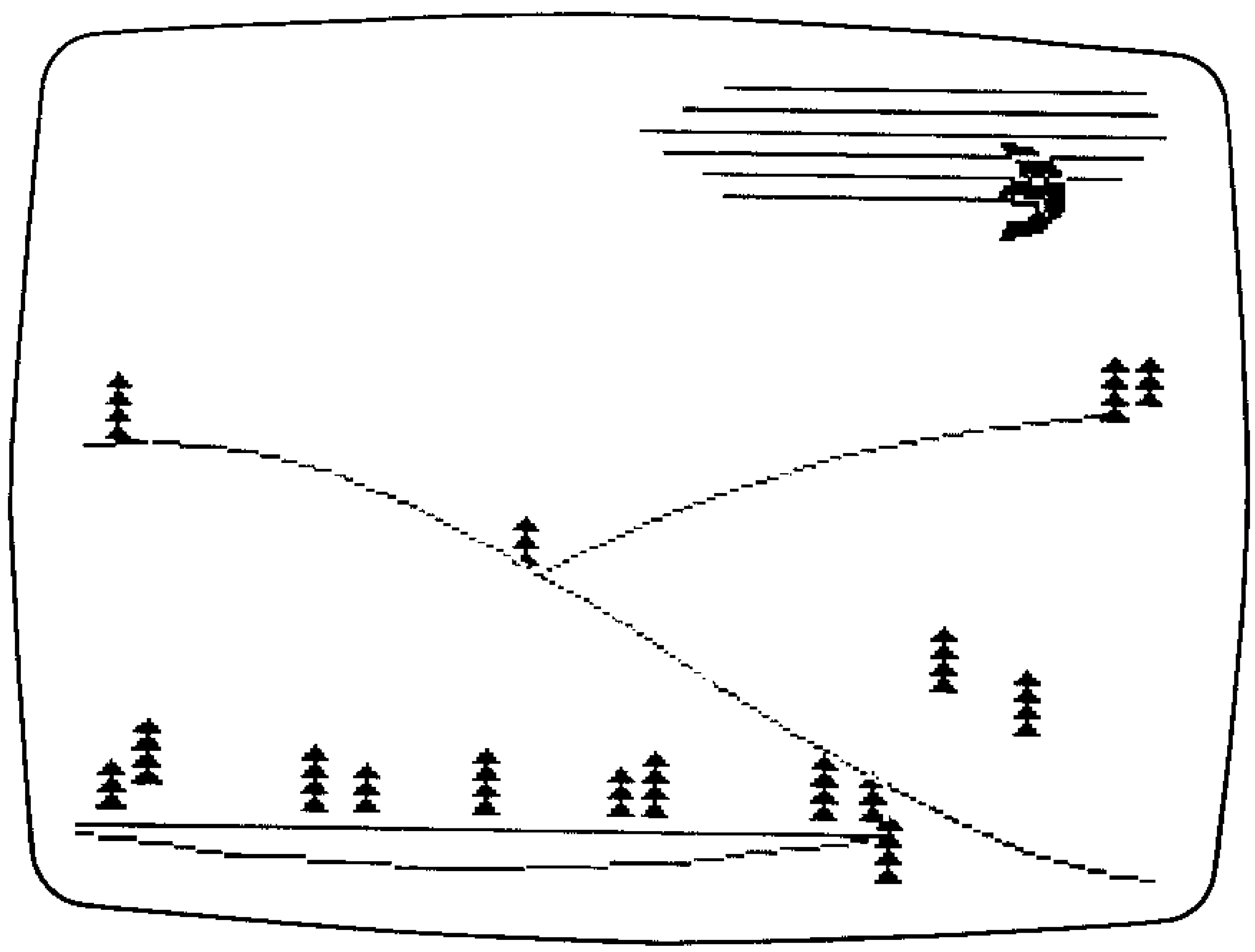
Now list your program; it should match the program listed below:

```
  10 REM *BUILD MOON STRING*
  20 DIM M$[46],M2$[69]
  90 SCALE 0,255,0,191
 100 PEN 1 @ GCLEAR
 110 MOVE 0,191
 115 GOSUB 1000
• 120 BPLOT M$,2
 210 REM *NOW SET UP LOOPS SO THE
         MOON MOVES FROM LEFT TO
         RIGHT ACROSS THE DISPLAY
 220 REM AND FROM TOP TO BOTTOM
         RIGHT CORNER*
 230 FOR Y=191 to 0 STEP -1
• 240 FOR X=0 TO 255 STEP 8
 250 MOVE X,Y
 260 BPLOT M2$,3
 270 WAIT 1000
 280 NEXT X
 290 NEXT Y
 300 STOP
•1000 M$="ε◂├ ◂?@▒ε‡xΠ_Π∑Γα†α▒▒├├├
         ├Sα‡_▒_ ◂_ ◂n¿_!?x?ε├`├ ◂ε◂"
•2000 M2$="εε◂├├ ◂?├@▒αε‡ωxΠπ!Πx∑Γ
         ⅰα†aα▒◂▒├◂├├ ◂├S<α‡P_▒@_ ◂__ ◂
         nb¿±!?Gx?Dε├▒`├├ ◂εε◂"
3000 RETURN
9999 END
```

If you run the program once again, the man in the moon will move across the display as it did earlier with the program on pages 249 and 250.

Problem

12.7 Create a scene for the man-in-the-moon using trigonometric curves for mountains, a segment of a circle for a lake, XAXIS statements for clouds, and our first triangle BPLOT for trees. Or better yet, create your own scene, by creating some new BPLOT character strings. Here's our scene; a sample program exists in appendix F.



Debugging and Error Recovery

Even the most experienced programmers find errors in their programs. These errors range from mistakes in the original formulas to mistakes in the logical flow of the program. Whenever they occur, they need to be found and corrected, and your HP-85 is designed to make various error-checking processes as easy and convenient as possible.

One of the easiest ways to find out if your program is working properly is to work a test case in which you either know the answer or the answer can easily be determined. In lengthy, complex programs, a wrong test-case answer will seldom pinpoint a mistake.

Tracing Program Execution

A convenient method of debugging logic errors in a program is to trace the order of statement execution and variable assignments in a program. The HP-85 provides three tracing statements; `TRACE`, `TRACE VAR`, and `TRACE ALL`, which includes `TRACE` and `TRACE VAR`.

The trace statements can be programmed or executed manually from the keyboard. The trace statements are independent of each other; thus, one or all of the trace statements can be in effect at any time, but `TRACE ALL` includes `TRACE` and `TRACE VAR`.

Tracing Branches

The `TRACE` statement is used to trace the order of statement execution in all or part of a program.

`TRACE`

If the order of program execution proceeds sequentially from the lowest numbered statement to the next higher numbered statement, nothing is printed. But whenever a branch occurs in a program, both the statement number where the branch occurred and the number of the statement to which it branched are printed in the form:

`TRACE LINE` statement number `TO` statement number

Tracing the Values of Variables

You can trace changes in the values of program variables without an output statement by using the `TRACE VAR` *(trace variable)* statement. Calculator mode variables cannot be traced and any attempt to do so will produce an error. Thus, you should always execute `INIT` or `RUN` prior to executing `TRACE VAR`.

`TRACE VAR` variable list

The variable list can contain simple numeric variables, string variables, and references to entire arrays. You can trace as many variables in a program as you wish. The variable names must be separated by commas in the list.

Simple numeric variables and string variables are specified by name; subscripted (array) variables are specified by a name followed by a set of parentheses. A comma may be included within the parentheses to specify a two-dimensional array if desired for documentation purposes.

For example, suppose your program contains simple numeric variables A and B, arrays C(4) and D(25,3), and string E$. To trace all of the variable assignments in your program, execute:

`TRACE VAR A,B,C(),D(,),E$`	Use the variable name followed by `()` to denote an array. The use of the comma for two-dimensional arrays is optional.

Whenever a change occurs in the value of the variable(s) you are tracing, the printed trace output indicates the statement number in which the change occured and:

- The name and new value of a simple numeric variable.
- The name, subscript(s), and new value of a particular array element.
- The name of a string variable.
- The name in the form A() or A(,) and the new value of the first element of the array for statements that operate on complete arrays (e.g., `READ#`).

The form of `TRACE VAR` output is:

`TRACE LINE` statement number variable name [`(`subscripts`)`] [`=` value]

New values are given only for simple numeric variables and array elements. Only the statement number of the variable change and the name of the variable that changed are given for strings. When an entire array is traced, only the new value of the first element is given, along with the statement number of the array change and the array name.

Trace All Statements and Variable Assignments

The `TRACE ALL` statement enables you to trace the order of program execution for every statement in a program and the value change of every variable in the program.

```
TRACE ALL
```

The `TRACE ALL` statement is most often used with the (STEP) key, as we'll see shortly.

Since the tracing operations output all information to the printer, you can save paper by executing the `PRINTER IS 1` statement before executing the program that you are tracing.

Cancelling Trace Operations

All trace operations are cancelled by executing `SCRATCH` or the `NORMAL` statement:

```
NORMAL
```

Example: Load and run the following Base Conversions program to find the octal representation of the binary number 10101010. Execute the TRACE statement before you run the program to follow the order of program execution.

```
10  REM *NUMBER BASE CONVERSION
20  DIM B$[16],I$[24],O$[24]
30  B$="0123456789ABCDEF"
40  DISP "INPUT BASE, OUTPUT BAS
    E";
50  INPUT B1,B2
60  DISP "NUMBER IN BASE";B1;
70  INPUT I$
80  N=0
90  FOR I=1 TO LEN(I$)
100 P=POS(B$[1,B1],I$[I,I])
110 IF P=0 THEN 270
120 N=B1*N+P-1
130 NEXT I
140 O$=""
150 N=N/B2
160 P=B2*FP(N)+1
170 O$=O$&B$[P,P]
180 N=INT(N)
190 IF N#0 THEN 150
200 PRINT I$;"BASE";B1
210 FOR I=LEN(O$) TO 1 STEP -1
220 PRINT O$[I,I];
230 NEXT I
240 PRINT " BASE";B2
250 PRINT
260 STOP
270 BEEP
280 PRINT I$[I,I];"NOT ALLOWED
    IN BASE";B1
290 PRINT
300 GOTO 60
310 END
```

Assigns check string.
Inputs number bases.

Inputs number in first base.

Initializes base 10 value.
Checks for illegal character.
If illegal character go to 270.
Accumulate equivalent in base 10.

Initializes output string.
Shifts one digit to the right.
Gets character.
Build string in reverse order.

Checks if done.
Prints input.
Prints output string.

Illegal character input.

We executed the PRINT ALL command to get the listing below:

```
TRACE
RUN
INPUT BASE, OUTPUT BASE?
2,8
NUMBER IN BASE 2?
10101010
 Trace line 130 to 100
 Trace line 130 to 100
 Trace line 130 to 100
 Trace line 130 to 100
 Trace line 130 to 100
 Trace line 130 to 100
 Trace line 130 to 100
 Trace line 190 to 150
 Trace line 190 to 150
10101010 BASE 2
 Trace line 230 to 220
 Trace line 230 to 220
252 BASE 8
```

As you can see, the program executes the same set of branching operations, from 130 to 100, seven times in order to convert the original number to its decimal equivalent.

Then the program executes line 150 through 190 three times (two branches from 190 to 150) to build the character string of its octal equivalent.
Finally, the string is output with the FOR-NEXT loop.

The program exits the FOR-NEXT loop in statements 90 through 130 when the decimal equivalent of the original value has been accumulated and stored in the variable N. Regardless of the base of the original number, or the base that you convert it to, you can find the decimal equivalent of the original number by using the TRACE VAR statement with this program.

Using the TRACE VAR statement, trace variable N to find the decimal equivalent of 1321 base 4 in the process of converting it to base 3.

First, execute NORMAL to cancel our previous TRACE statement.

```
NORMAL                                  NORMAL also cancels PRINT ALL.
PRINT ALL                               Resets print all mode.
TRACE VAR N                             Traces variable N.
RUN
INPUT BASE, OUTPUT BASE?
4,3
NUMBER IN BASE 4?
1321
 Trace line 80 N=0
 Trace line 120 N=1
 Trace line 120 N=7
 Trace line 120 N=30
 Trace line 120 N=121                   Decimal equivalent of 1321₄;
 Trace line 150 N=40.3333333333         The largest value of N.
 Trace line 180 N=40
 Trace line 150 N=13.3333333333
 Trace line 180 N=13
 Trace line 150 N=4.33333333333
 Trace line 180 N=4
 Trace line 150 N=1.33333333333
 Trace line 180 N=1
 Trace line 150 N=.333333333333
 Trace line 180 N=0
1321 BASE 4
11111 Base 3
```

You can use any of the TRACE statements in a program. For instance, add the following statements to the Base Conversions program.

```
85 TRACE VAR O$,B$,N
135 NORMAL @ PRINT ALL
```

Now execute NORMAL to cancel our previous TRACE VAR statement and run the program to find the binary equivalent of 86_9.

```
NORMAL
PRINTALL
RUN
INPUT BASE, OUTPUT BASE?
9,2
NUMBER IN BASE 9 ?
86
 Trace line 120 N=8                     Since the values of variables O$ and B$ do
 Trace line 120 N=78                    not change between statements 85 and
86 BASE                                 135, no TRACE VAR output occurs
1001110 BASE 2                          for them.
```

The STEP Key

You can execute a program one line at a time by using the (STEP) key. If a program has been halted by a PAUSE statement or the (PAUSE) key, it can be continued, one line at a time, by pressing the (STEP) key. As soon as (STEP) is pressed, the line designated by the internal program counter is executed, then the program halts again.

You can execute an entire program—one step at a time—by first initializing the program with the (INIT) key (or by executing INIT), and then by pressing (STEP) to execute each program statement.

Since the (STEP) key does not output anything to the printer or display, it is often desirable to execute the TRACE ALL statement so that you know what line is being executed.

Example: Execute the TRACE ALL statement and then execute the Base Conversions program, one statement at a time using the (STEP/PAUSE) key. (Remember to delete statement 135, from out last example, so that the tracing operations are not cancelled.) First initialize the program with the INIT command.

```
PRINT ALL
INIT
TRACE ALL

 Trace line 10 to 20
 Trace line 20 to 30
 Trace line 30 B$
 Trace line 30 to 40
 Trace line 40 to 50
INPUT BASE, OUTPUT BASE?
2,10
 Trace line 50 B1=2
 Trace line 50 B2=10
 Trace line 50 to 60
 Trace line 60 to 70
NUMBER IN BASE 2 ?
11110110
 Trace line 70 I$
 Trace line 70 to 80
 Trace line 80 N=0
 Trace line 80 to 90
 Trace line 90 I=1
 Trace line 90 to 100
 Trace line 100 P=2
 Trace line 100 to 110
 Trace line 110 to 120
 Trace line 120 N=1
 Trace line 120 to 130
 Trace line 130 I=2
 Trace line 130 to 100
 Trace line 100 P=2
 Trace line 100 to 110
 Trace line 110 to 120
 Trace line 120 N=3
 Trace line 120 to 130
 Trace line 130 I=3
 Trace line 130 to 100
 :
```

Initializes the program.
TRACE ALL traces all variables and branches.
Now press the (STEP) key to execute the program one statement at a time.
Be sure to press (END LINE) to enter data requested by an INPUT statement. Then continue stepping through the program by pressing the (STEP) key. If you press the (STEP) key in response to an input statement, the keycode for (STEP), θ, will appear. Simply backspace and enter the data with (END LINE).

```
  :
  :
 Trace line 180 N=0
 Trace line 180 to 190
 Trace line 190 to 200
11110110 BASE 2
 Trace line 200 to 210
 Trace line 210 I=3
 Trace line 210 to 220
 Trace line 220 to 230
 Trace line 230 I=2
 Trace line 230 to 220
 Trace line 220 to 230
 Trace line 230 I=1
 Trace line 230 to 220
 Trace line 220 to 230
 Trace line 230 I=0
 Trace line 230 to 240
246 BASE 10
```

Checking a Halted Program

Various operations that aid in debugging can be performed on a program halted by (PAUSE) or an Error or before pressing (STEP) or (CONT):

- Values of variables can be checked merely by keying in the variable names followed by (END LINE).
- Values of variables can be assigned or changed if statements like A(5,2)=7 or B=0 are executed.
- Most program statements can be executed without statement numbers, like PRINT, DISP or COPY.
- Arithmetic operations can be performed and math functions can be executed.
- Any system command can be executed—PRINT ALL, DEG, GRAD, or RAD, etc.

If the halted program is continued with either the (CONT) or (STEP) key, any of the previously mentioned operations that affect program execution remain intact. For instance, values of variables that were changed retain their new values; a DEG statement causes the program to calculate angles in degrees, etc.

If, however, the halted program is restarted with a RUN command, or INIT is executed before you continue, then the program is initialized so that all variables start with undefined values until reassigned a value in the program (or from the keyboard). (Refer to appendix C to see what is affected by RUN and INIT.) You cannot continue a program with (STEP) or (CONT) after editing a line of the program; the program must first be initialized with RUN or INIT.

Error Testing and Recovery

Run time errors are those that occur only when a program is running. Division by zero is an example. These errors normally halt execution of a running program. The DEFAULT ON statement (which is set at power on) provides default values for some error-causing operations, displays a warning message, and thus allows your program to continue. There is more than one way to control program errors or catch a bug. Through use of the ON ERROR statement, routine errors can be recovered so that execution can continue with the specified line after execution of the line in which the error occured.

The ON ERROR statement specifies a branching that takes place after an error occurs.

ON ERROR GOTO statement number
ON ERROR GOSUB statement number

The ON ERROR statement declares what should happen if an error occurs. It need be executed only once in each program segment to establish the ON ERROR condition. Execution of another ON ERROR statement replaces the previous one.

When a run-time error occurs and the ON ERROR condition has been established, execution is transferred to the specified line. Then the ERRN and ERRL functions (discussed next) could be tested and error recovery procedures could be executed. The error is "ignored" if the statement referenced by a GOSUB is a RETURN statement. Execution continues with the statement after the one in which the error occured.

If the recovery routine also contains an error, it is possible to program into an endless loop. The program can be halted, of course, by pressing (RESET).

CAUTION

It is not recommended to ignore service errors. Error numbers 65, 73, and 74 can signify a defective cartridge or defective tape transport. Refer to appendix B.

An ON ERROR statement is disabled with the OFF ERROR statement:

OFF ERROR

It is usually a good idea to turn off the ON ERROR condition as soon as an error has been found so that you don't trap out unexpected errors.

Two numeric functions are associated with ON ERROR:

ERRL The *error line* function outputs the line number in which the most recent program execution error occured.

ERRN The *error number* function outputs the number of the most recent program execution error. Appendix E contains a complete list of error numbers and messages.

Example: Suppose you are concerned that a certain computation will cause a numeric overflow. If it does, you want to skip that segment of the program and go to another part of the program. If the error is not a numeric overflow error, you want to display the number of the statement in which the error occurred and pause so that you can do some program checks. Write the necessary statements that would carry out these functions.

In the following program, we create some obvious errors so that the order of program execution may be followed easily.

```
  10 N=1.E400
  20 DISP N
  30 M=1.E400
  40 DISP M
• 50 ON ERROR GOTO 100        If an error occurs, go to 100.
  60 K=N*M                    We turn off the error overriding condition
  70 OFF ERROR                as soon as the error occurs, or the expected
  80 DISP K                   error-causing statement is executed so
  90 GOTO 510                 that we don't trap out errors we're not
 100 OFF ERROR                prepared to handle.
•110 IF ERRN=2 THEN 500       If error number 2 (overflow), then go to
                              500.
•120 DISP "ERRN=";ERRN;"ERRL=";E   If not an overflow, display error number
     RRL                      and error line.
 130 PAUSE                    Then pause.
 500 DISP "ARRIVED AT STATEMENT    If an overflow condition occurs, will skip
     NUMBER 500"              to this statement.
 510 END
```

Now run the program:

```
RUN
 1.E400
 1.E400
ARRIVED AT STATEMENT  NUMBER 500
```

As you can see, the program checked for an overflow condition, then skipped to statement 500.

Now change statement 60 to read:

```
60 K=N/0                      Creates a division by zero error.
```

And run the program again:

```
RUN
 1.E400
 1.E400
ERRN= 8 ERRL= 60              Displays error number for division by zero
                              and the line number in which the error
                              occured, then pauses.
```

Since the program has paused, you may perform various program checks, or change the values of variables before you continue.

Some Hints About the System

Occasionally, your program may not work the way you think it should, perhaps by giving erroneous results, and yet the system does not detect any errors. The following list of things to remember about the HP-85 system may help you in detecting program errors or user-misunderstandings.

- Be sure to close data files if the program halts because of an error in the midst of printing to a file. Remember that the system uses buffers to "write" data to the tape and that the buffers are "dumped" only under specific conditions (refer to page 184).

- If an assignment (e.g., `J=5`) is made before a program is initialized with `RUN` or `INIT`, it is a calculator mode variable, not known to the program. However, when an assignment is made after a program is allocated (initialized) to a variable that is known to the program, then the assignment is made to that program variable. If the variable is not known to the program (i.e., not referenced in the program), then it is a calculator mode variable.

- Error messages report the first error that occured; there may be others. Remember that the system tries to interpret an expression as a statement and then as an expression. If you get an error with a calculator mode expression, try executing the same expression in a `DISP` statement. Then the system will know that it is looking at an expression in a `DISP` statement and you'll get a better error message. A bad expression is considered a bad statement if typed as [expression] (END LINE).

- The three programmable timers are extremely useful, but be aware that they interrupt the system at the frequency you specify until they are turned off by executing (SCRATCH), (RESET), or `OFF TIMER#`. Thus, for small interrupt intervals, timers can have an adverse effect on the execution speed of the system.

- The `XAXIS` and `YAXIS` statements are interruptable during tic-generation by pressing (RESET)—just in case you specify very small tic spacing that might take hours to complete.

- **Programs are stored allocated (including the space required for dedicated variables) unless they contain variables in common (with a `COM` statement) or have allocation errors (e.g., missing a reference line). If your program contains dimensioned variables, you may want to add a token `COM` statement (e.g., `1 COM Z9`) in order to deallocate the program before it is stored.**

- If you reference a multiple-line function in a `PRINT` or `DISP` list, and the function contains a `PRINT` or `DISP` statement, you may not get the output you expect.

- Should you get a memory overflow error when attempting to read a long string from a data file, break the string into shorter substrings and write the substrings into smaller logical records. Then read the substrings back, one at a time, into computer memory.

Memory Conservation Hints

- Remarks and comments (!) take one byte per character. Use enough to document your program but don't be excessively wordy.

- Use `INTEGER` and `SHORT` data types for arrays whenever possible.

- Use `INTEGER` constants instead of `REAL` constants whenever possible (e.g., 4 instead of 4.).

- Explicitly dimension *all* arrays if the upper bound is not 10.

- Explicitly dimension all strings if the maximum length is 10 or less.

- Use `OPTION BASE 1` if you don't plan to use the zero'th element of your arrays.

- Use multistatement lines (using @) when it doesn't detract from program readability and if the program is not intended to be run on other BASIC computers.
- Use a variable assignment for program constants that occur more than once. Variable names take up less space.
- Use subroutines or functions for program sequences that occur more than once.
- Try to reuse variables when possible, rather than declaring new variables.

Accessories

Standard Accessories

Your HP-85 comes equipped with one each of the following standard accessories:

Accessory	Part Number
• *HP-85 Owner's Manual and Programming Guide*	00085-90002
• *HP-85 Pocket Guide*	00085-90040
• *HP-85 BASIC Reference Card*	00085-90039
• Standard Pac, including: Instruction Manual Preprogrammed Tape Cartridge	00085-13001
• Registration Card	
• Service Card	
• Accessory Data Sheet	
• Users' Library Form	
• Roll of Thermal Printer Paper	
• Power Cord	
• Fuses and Fuse Cap Holders 750 milliamperes fuse (for 115 Vac-nominal line voltage) and U.S. style fuse cap holder T400 milliamperes fuse (for 230 Vac-nominal line voltage) and European style fuse cap holder	
• Three-Ring Binder and Dividers	

Optional Accessories

In addition to the standard accessories shipped with your HP-85, Hewlett-Packard also makes available the following optional accessories. These have been created to help you maximize the usability and convenience of your personal computer.

HP-85 Applications Pacs

Each pac offers one or more BASIC programs in a particular field or discipline prerecorded on a tape cartridge. Each pac comes complete with a detailed instruction manual and handy pac binder that carries up to four cartridges and the instruction manual.

HP-85 Plug-in Modules

- HP 82903A 16K RAM (Random Access Memory)
  The 16K memory module approximately doubles the amount of space for programming and data storage.
- Enhancement ROMs (Read Only Memory)
  A ROM drawer and several plug-in ROMs will be released in the near future that actually enhance the capabilities of your HP-85. For more information about HP-85 enhancement ROMs call the following toll-free number: 800/547-3400 (except in Alaska and Hawaii).

Owner's Manual and Pocket Guides

Supply members of your staff with personal copies of HP-85 operating information:

- *HP-85 Owner's Manual and Programming Guide* 00085-90002
- *HP-85 Pocket Guide* 00085-90040
- *HP-85 BASIC Reference Card* 00085-90039

HP-85 Carrying Case (HP 82933A)

With its stylish, leather-like exterior, made of durable, easy-to-clean vinyl, the lightweight HP-85 carrying case provides you with a convenient means of transporting your computer safely. Inside the case, molded foam liners conform exactly to the contours of the HP-85, providing maximum shock absorption. Designed to carry the computer, enhancement plug-in modules, and power cord, the carrying case also has an exterior-accessible pouch that will hold two instruction manuals, a roll of paper, and several tape cartridges. The case is secured with a three-sided zipper and is fitted with a double-web handle with keeper, providing a suitcase-type grip. Dimensions:

23 centimeters (9 inches) thick
48 centimeters (19 inches) wide
56 centimeters (22 inches) high

BLANK TAPE CARTRIDGES (HP 98200A)

Hewlett-Packard blank tape cartridges are available in packages of five each.

Tape Cartridge Binder (HP 82932A)

The tape cartridge binder provides you with a convenient way to both store and transport your HP-85 tape cartridges and one instruction book. Available in vinyl, the case measures 29 cm (11.5 in) high, 28 cm (11 in) wide, and 5 cm (2 in) deep, and has space for four Hewlett-Packard tape cartridges.

Thermal Printer Paper (HP 82931A)

Each pack gives you two rolls of special HP-85 thermal printer paper. Roll length: 120 meters (400 feet).

Three-Ring Manual Binder and Dividers (HP 82935A)

Additional HP-85 manual binders are available, enabling you and members of your staff to organize your HP-85 system user's manuals conveniently. The binder, measuring 29 cm (11.5 in) high, 28 cm (11 in) long, and 6.5 cm (2.5 in) wide, includes sheet lifters and a set of dividers.

Owner's Information

The following information covers the initial set-up of your HP-85 Personal Computer and includes other information that is important when you first receive the computer.

Note: Become thoroughly familiar with the information in this appendix before attempting to operate your HP-85.

Inspection Procedure

Your HP-85 is another example of the award-winning design, superior quality, and attention to detail in engineering and construction that have marked Hewlett-Packard electronic instruments for more than 30 years. Each Hewlett-Packard computer is precision crafted by people who are dedicated to giving you the best possible product at an affordable price.

Your HP-85 computer was thoroughly inspected before shipping and should be ready to operate after completing the set-up instructions. Carefully check the computer for any physical damage sustained during shipment. Do not turn the power on if the CRT display shows any cracks. Notify your dealer and file a claim with any carriers involved if there is any such damage.

Please check to ensure that you have received all of the standard accessories included with the HP-85. Review the list of standard accessories in appendix A. If any accessory items are missing, please contact the dealer from whom you purchased the computer. If your computer was purchased directly from Hewlett-Packard, please contact the office through which your order was placed.

Power Supply Information

Power Cords

Power cords supplied by HP have polarities matched to the power-input socket on the machine, as shown below.

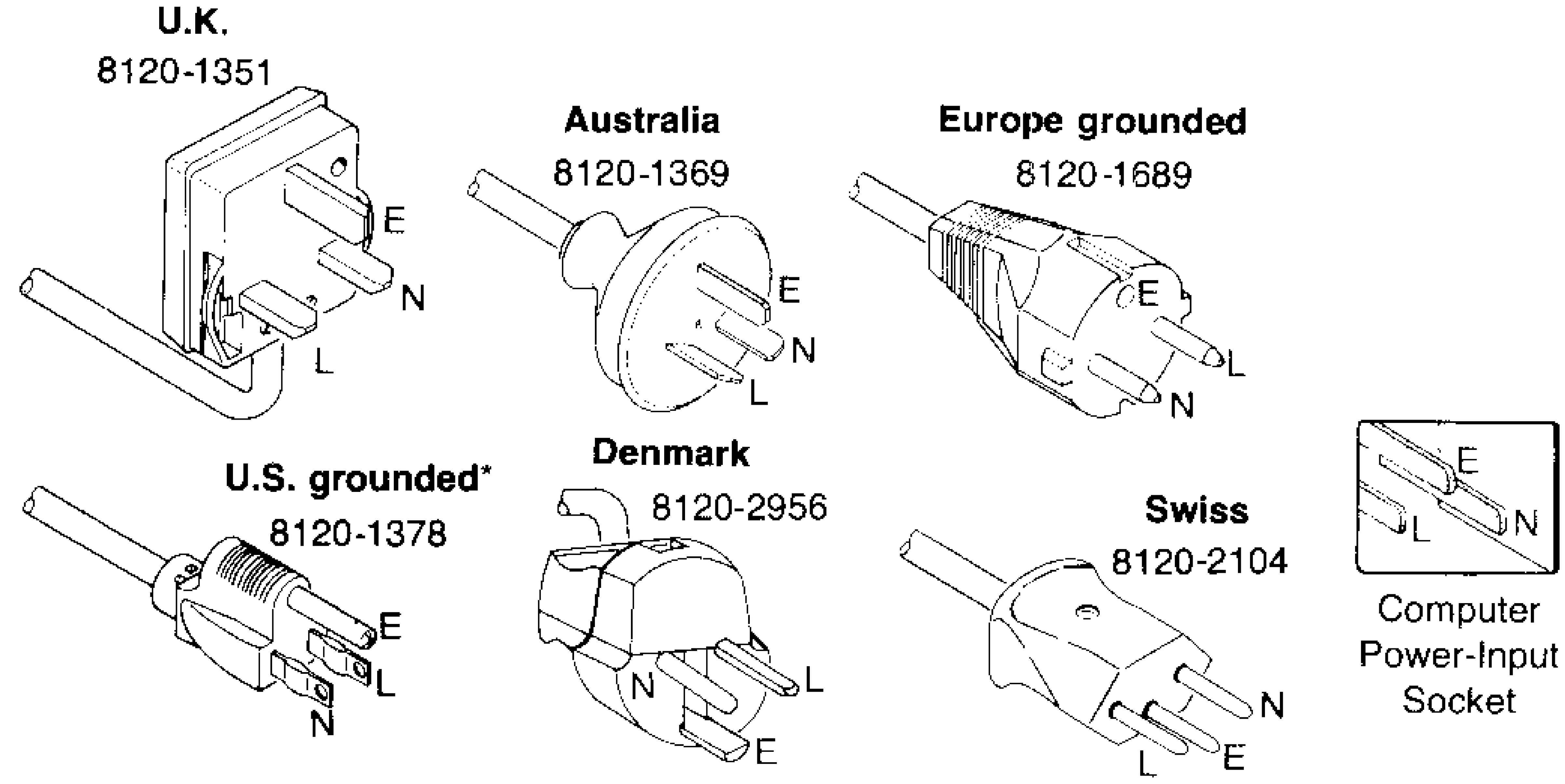
- L=Line or Active Conductor (also called ''live'' or ''hot'')
- N=Neutral or Identified Conductor
- E=Earth ground

WARNING

Use only the HP-85 power cord specified by Hewlett-Packard for your area.

If it is necessary to replace the power cord, the replacement cord must have the same polarity as the original. Otherwise a safety hazard from electrical shock to personnel might exist. In addition, the equipment could be extensively damaged.

Power cords with different plugs are available for the HP-85; the part number of each cord is shown below. Each plug has a ground connector. The cord packaged with the machine depends upon where the machine was delivered. If your equipment has the wrong power cord for your area, please contact your local authorized HP-85 dealer or HP sales and service office for information on how to obtain the proper cord.



Grounding Requirements

To protect operating personnel, the National Electrical Manufactures' Association (NEMA) recommends that all equipment not double insulated be properly grounded. The HP-85 is equipped with a three-conductor power cable which, when connected to an appropriate power receptacle, grounds the machine. To preserve this protection feature, do not operate the machine from a power outlet which has no earth ground connection.

WARNING

To avoid the possibility of any injury, disconnect the ac power cord before installing or replacing a fuse.

Power Requirements

The HP-85 has the following power requirements:

Line Voltage	
115 Vac Nominal	100/117 Vac
230 Vac Nominal	220/240 Vac
Line Frequency	50/60 Hz
Power Consumption	40 Watts Nominal

Fuses

For 100/117 Vac operation, set the voltage selector switch to 115V and use a 750mA fuse; for 220/240 Vac operation, set the voltage selector switch to 230V and use a T400mA fuse.

* UL and CSA approved for use in the U.S. and Canada with machines set for 115 Vac operation.

Service

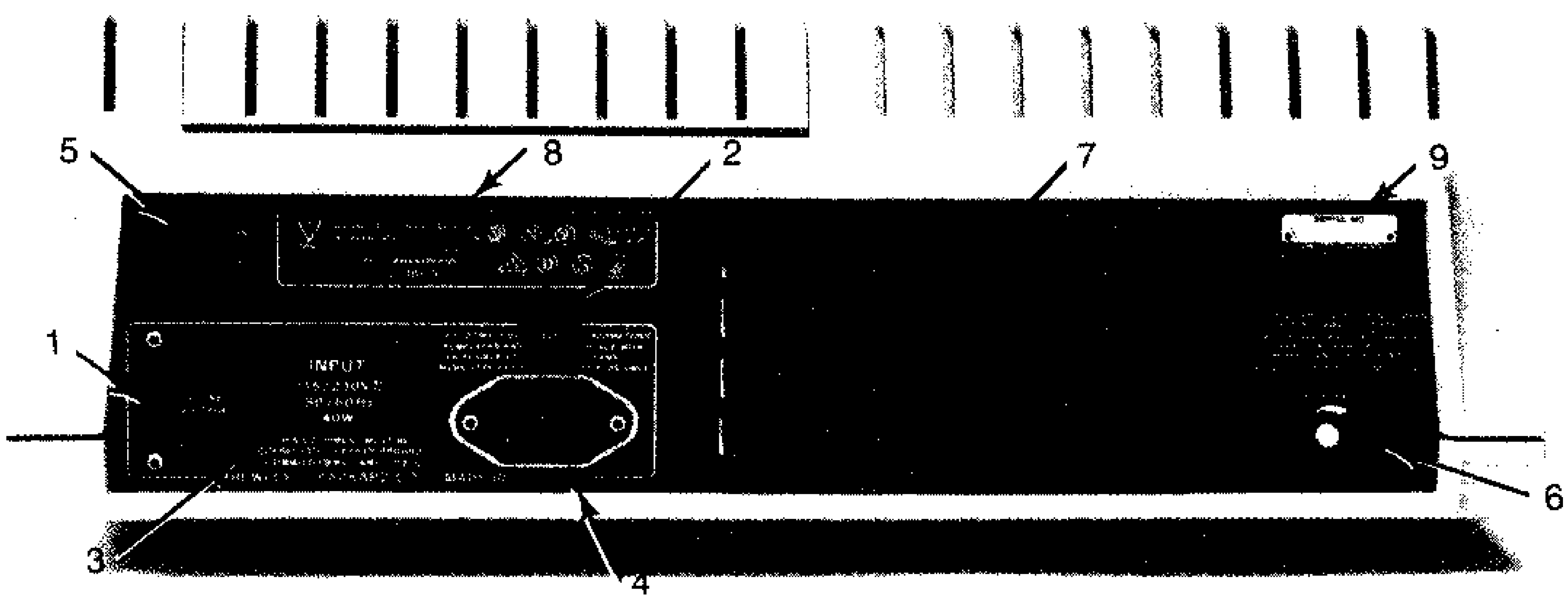
WARNING
High voltages are present inside the HP-85. There are no customer serviceable parts inside the HP-85. In case of any difficulty or malfunction with your HP-85 contact your nearest authorized HP-85 dealer or HP repair facility.

For specific warranty and service information, refer to pages 287 through 289.

Rear Panel

Understanding the rear panel layout and features of your HP-85 computer is important for safe and efficient operation. The rear panel contains the following:

1. Line Voltage Selector Switch.
2. Fuse Receptacle.
3. Ground Information.
4. Power Cord Receptacle.
5. ON-OFF Switch.
6. Display Brightness Control.
7. Module Plug-in Ports with Covers.
8. Worldwide Saftey Approval Nomenclature.
9. Serial Number Plate.

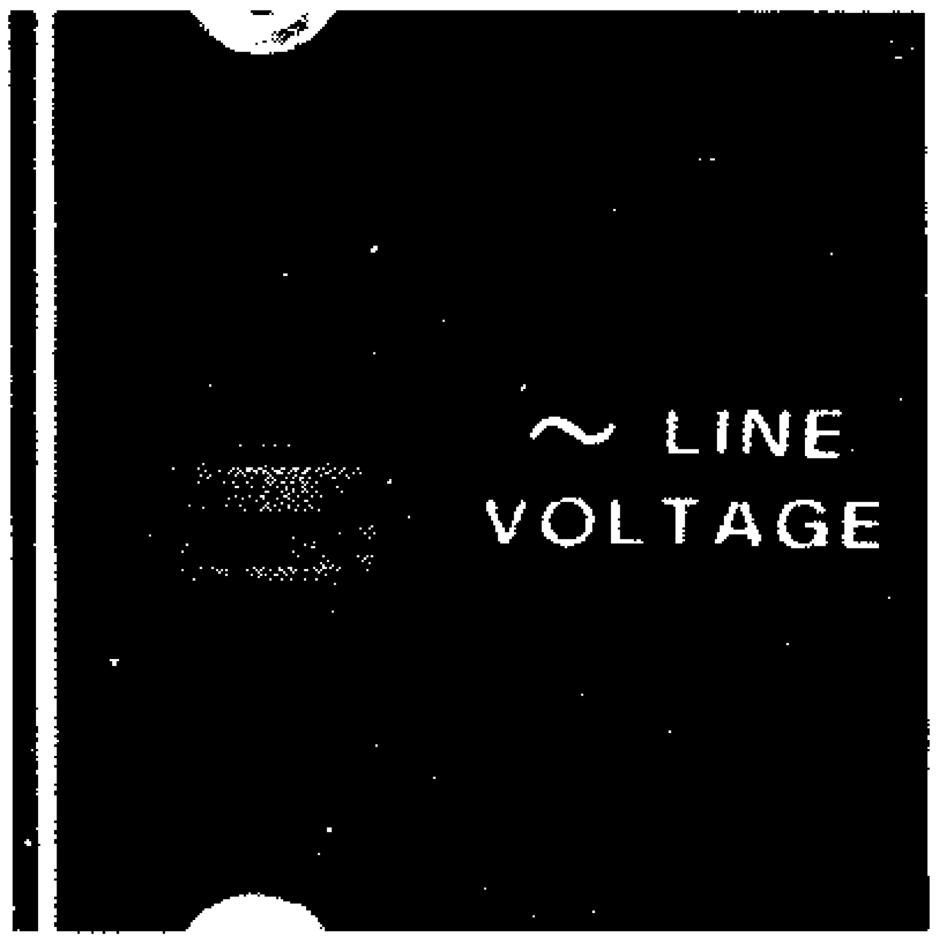


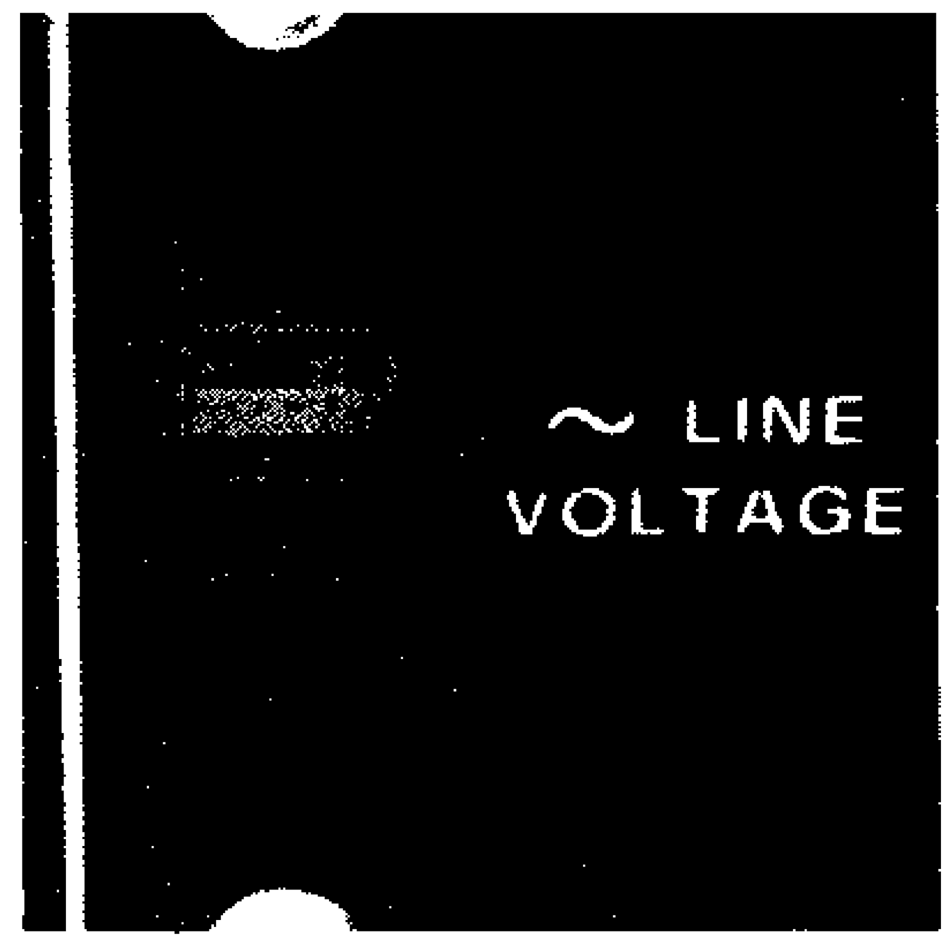
Initial Set-up Instructions

1. Disconnect the power cord and make sure the ON/OFF switch is OFF.
2. Ensure that the voltage selector switch located on the rear panel of the computer is set for the voltage range of the nominal line voltage in your area.

CAUTION
Check the selector switch before applying power. Damage to the computer will occur if the selector switch is set to 115 volts ac, and 230 volts ac is applied to the power input connector.

If it is necessary to alter the setting of the switch, insert the tip of a small screwdriver or coin into the slot on the switch. Slide the switch so that the position of the slot corresponds to the desired voltage as shown below. The computer is shipped with the voltage selector in the 230 Vac position.



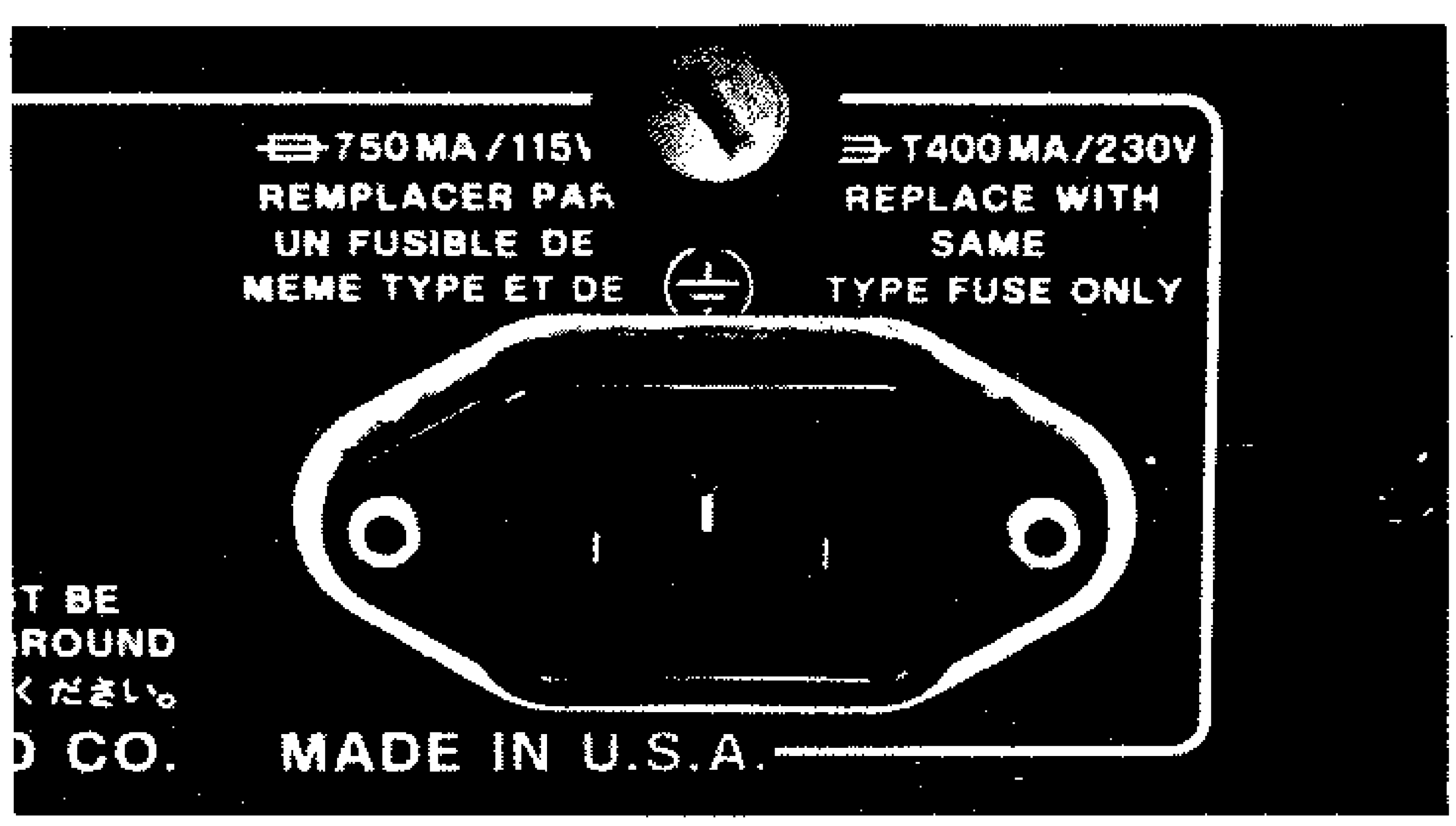


WARNING

Before installing or replacing a fuse, be sure that the computer is disconnected from any ac power source. Otherwise, a chance of electrical shock to personnel exists and the new fuse might be immediately overloaded.

3. Next install the proper fuse.

   The computer's fuse receptacle is located on the rear panel. (See figure below.) A 750 mA fuse is required for 115 Vac operation and a T400 mA fuse is required for Vac operation.



   The photograph shows the location of the fuse receptacle on the rear panel. To install or replace the fuse, first disconnect the power cord from the machine. Install or replace the proper fuse in the fuse cap holder (either end of the fuse can be inserted into the cap). Now, install the fuse and fuse cap into the fuse receptacle by pressing the cap inward and at the same time turning it clockwise until it locks in place.

4. Now, connect the power cord to the power input receptacle on the back of the computer. Plug the other end of the cord into the ac power outlet.

5. Switch the HP-85 on using the switch on the upper left side of the rear panel. A cursor (underscore) should appear in the upper left corner of the CRT display within 2 to 3 seconds. Each time the power is turned on, the system performs a self-test operation. When the cursor appears on the screen, the HP-85 is ready to go to work.

The brightness of the display can be adjusted using the Brightness knob on the lower right side of the rear panel.

Should the cursor not appear or the words `Error 23 : SELF-TEST` appear on the display, turn the machine off, then on again. Should the problem persist, contact your local authorized HP-85 dealer or HP sales and service office.

Installing Plug-In Modules

Your HP-85 is designed with four module ports on the rear panel. The ports are numbered 1 through 4 from the top. Before shipping from the factory, each port is fitted with a removable protective cover. It is recommended that each port be kept covered when not in use.

First we will discuss general module installation and removal, then we will discuss the installation of plug-in ROMs into a special ROM drawer module.

WARNING

Do not place fingers, tools, or other foreign objects into the plug-in ports. Such actions may result in minor electrical shock hazard and interference with pacemaker devices worn by some persons. Damage to plug-in port contacts and the computer's internal circuitry may also result.

General Module Installation and Removal

The HP-85 plug-in modules may be installed or removed as often as your needs require. To install modules, observe the following procedures.

1. Read all documentation accompanying each module for user instructions, warnings, and any limitations.

CAUTION

Always switch off the machine and any peripherals involved when inserting or removing modules. Use only the plug-in modules designed by Hewlett-Packard specifically for the HP-85. Failure to do either could damage equipment.

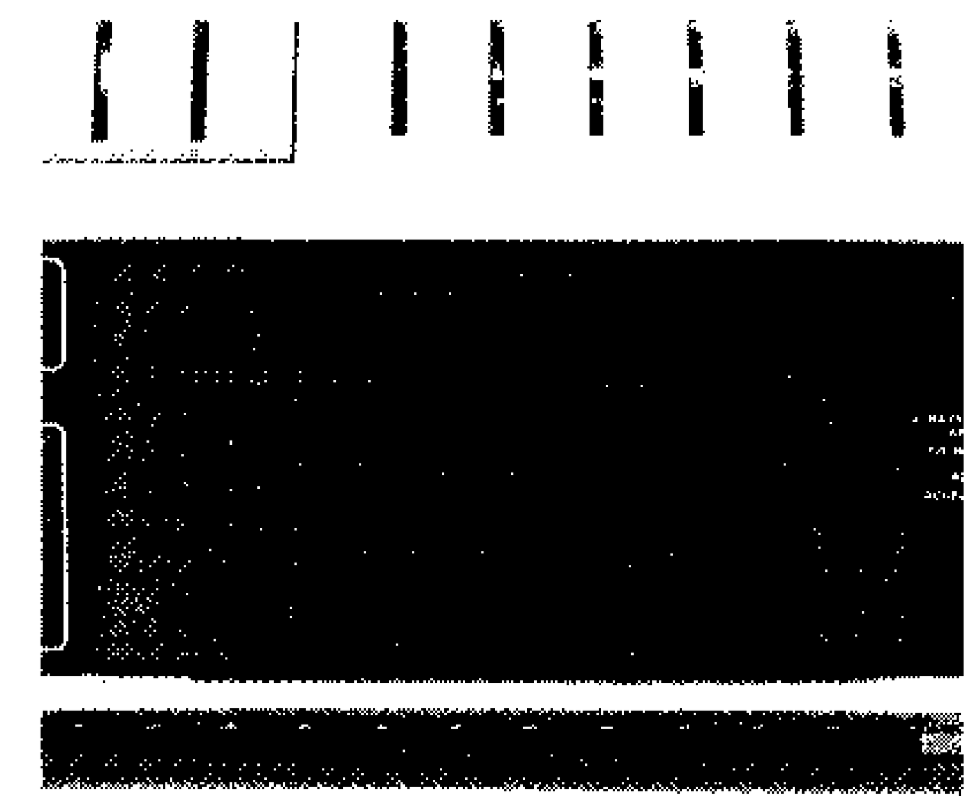
2. Turn off your HP-85 system. If an interface module is to be installed, or is already in use, switch off any peripheral devices involved.

CAUTION

If a module jams when inserted into a port, it may be upside down or designed for another port. Attempting to force it further may result in damage to the computer or the module.

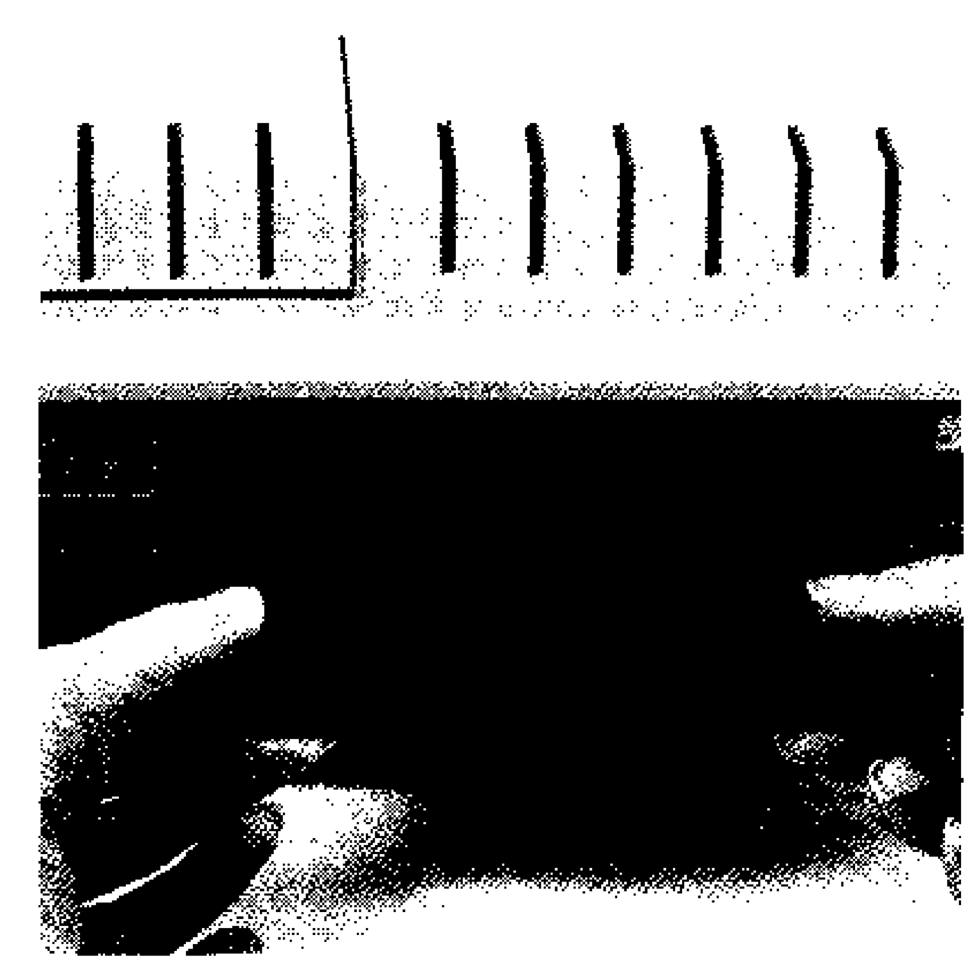
To insert a plug-in module:

1. Remove the protective cover from the plug-in port to be used.



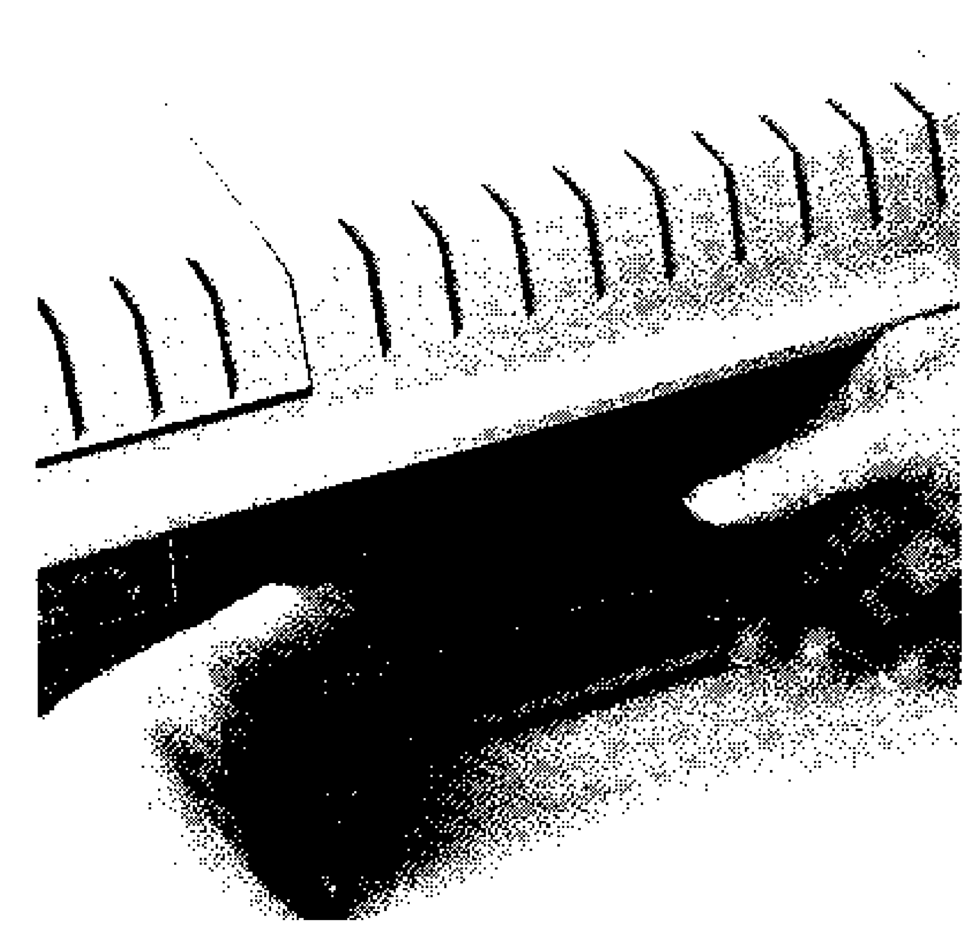
Note: Most plug-in modules can be inserted in any of the four ports. However, examine the documentation included with each module for any instructions regarding the use of a specific plug-in port. If it is intended that a module fit into a particular port, it can be inserted only in that port.

2. With the label right-side-up, insert the contact end of the module into the port and push until the module seats firmly with its stops against the port's edge. A slight up and down motion may be necessary to start the module moving in the tracks of the port. The tracks are keyed to prevent upside-down module insertion.

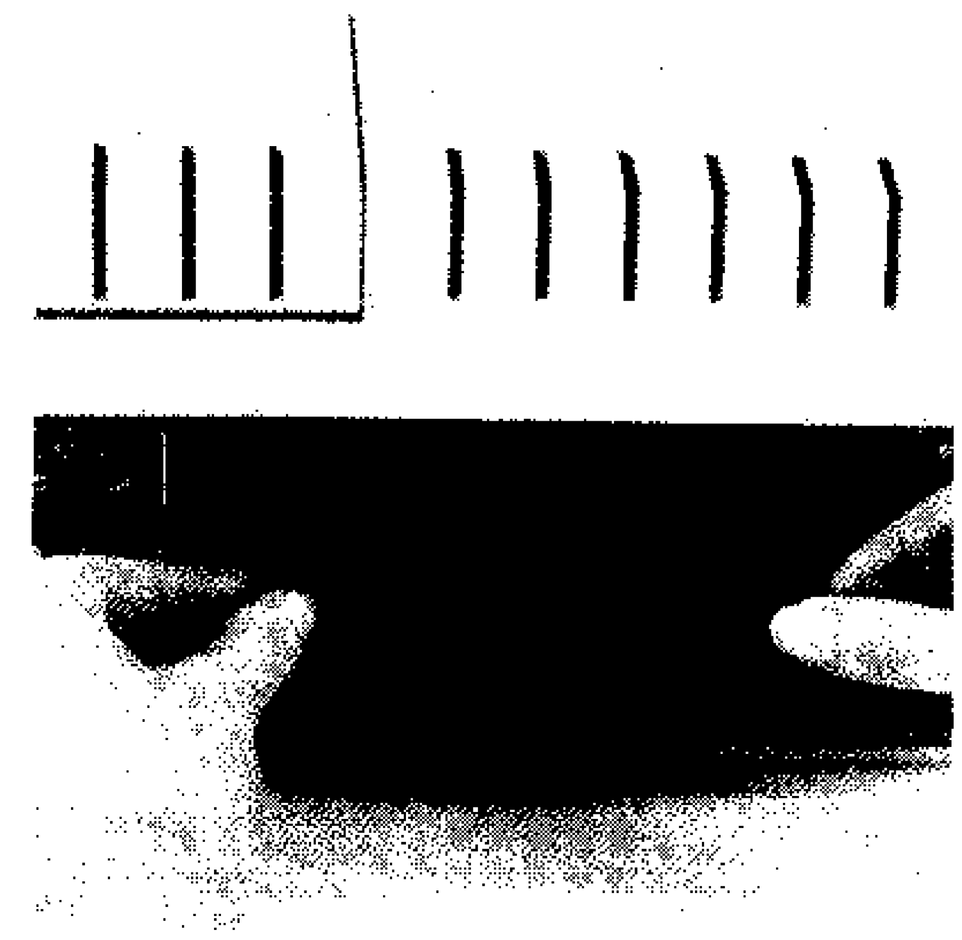


To remove a plug-in module, observe the following procedure:

1. Switch OFF your HP-85 system and any connected peripherals.
2. Firmly grasp and pull the module free of the port. Store the module in its original container or where it will be safe from damage to the contacts.



3. Replace the port cover.



Note: Up to four different modules can be installed in the HP-85 at any time. However, do not install duplicate RAM modules or duplicate ROMs into the ROM drawer. Such duplication can create error conditions and will not increase memory or computing power.

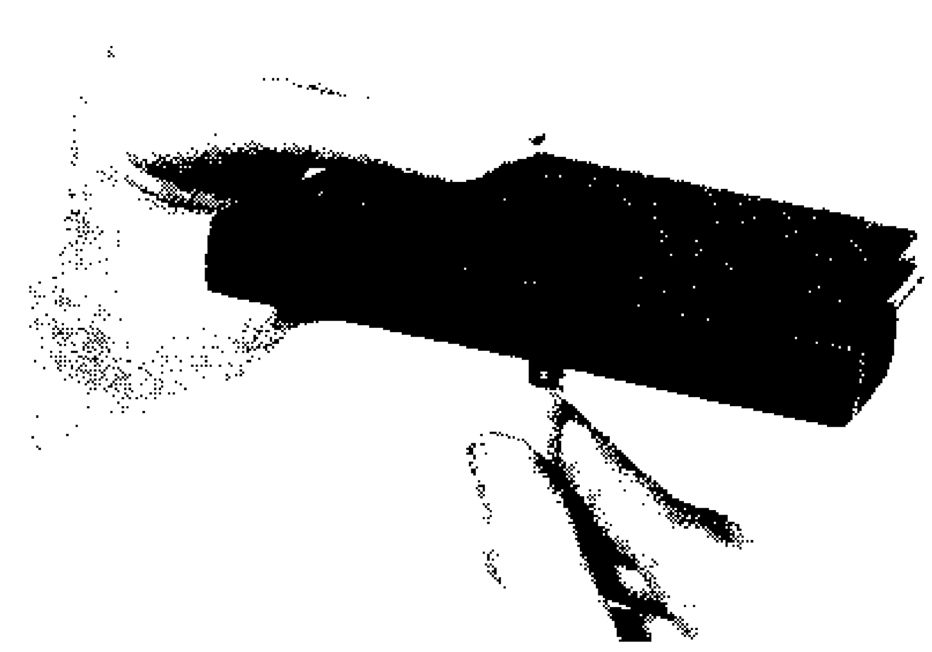
Plug-in ROM Installation and Removal

The ROM drawer is a particular plug-in module that contains six rectangular slots for individual plug-in ROMs, each fitted with its own protective cap.

Any HP-85 plug-in ROM will fit in any of the six positions in the ROM drawer. Be sure to read all documentation accompanying each plug-in ROM for user instructions, warnings, and any limitations. Remember that duplicate ROMs will *not* increase your computing power and may even create error conditions.

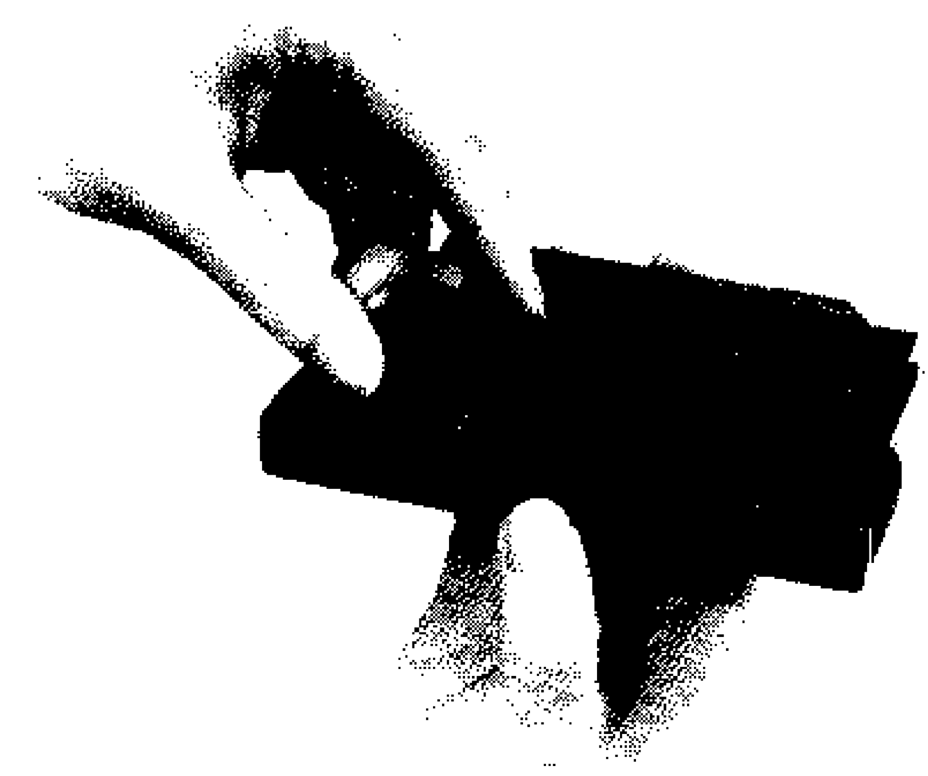
To insert a plug-in ROM into the ROM drawer:

1. Remove the protective cap from the desired plug-in slot in the ROM drawer as follows:
   - Insert the eraser end of a pencil into the circular hole on the underside of ROM drawer.
   - Press with the pencil until the cap snaps off.



CAUTION
Do not touch the spring-finger connectors in the ROM drawer with your fingers or insert tools or other foreign objects. Static discharge could damage electrical components.

2. Inside each plug-in slot in the ROM drawer you can see two rows of spring-finger connectors. These connectors correspond to the two rows of holes on the underside of the ROM plug. ROMs can be inserted in only one direction. Insert the ROM plug into the slot with its label up and its beveled edge toward the connector side of the ROM drawer. Push the ROM into place so that the top of the plug is flush with the top of the ROM drawer.



Note: Leave the cap on any slot in the ROM drawer that is not in use.

3. When all of the desired plug-in ROMs have been inserted into the ROM drawer, the module may be installed into a plug-in port on the rear panel of the HP-85 as described under General Module Installation and Removal.

To remove a plug-in ROM from the ROM drawer:

1. First remove the ROM drawer as described under General Module Installation and Removal.
2. Insert the eraser end of a pencil into the hole on the underside of the ROM drawer corresponding to the ROM you wish to remove, just as you did to remove the protective cap. Push gently with the pencil until the ROM pops out.

3. Replace the protective cap over the slot in the ROM drawer.

Your HP-85 Printer

The printer in your HP-85 is a thermal printer that uses a moving print head to print on a special heat-sensitive paper When the print head is energized, it heats the paper beneath it. The heat causes a chemical reaction in the paper, which then changes color. The printer, designed expressly for the HP-85, prints quickly and quietly at 2.6 lines per centimeter (6.7 lines per inch) at about two lines per second.

Graphics output is uni-directional and, therefore, approximately half the normal print speed.

Paper for Your HP-85

Because the printer in your HP-85 is a thermal printer, it requires special heat-sensitive paper. You should use only the Hewlett-Packard thermal paper available in 400-foot long rolls from your nearest authorized HP-85 dealer or HP sales and service center, or in the U.S., by mail from:

Hewlett-Packard
Corvallis Division
P.O. Box 999
Corvallis, Oregon 97330

Because of the special heat-sensitive requirements of the paper, impact printer paper will *not* work in the HP-85. Also, since different types of thermal paper vary in their sensitivities and abrasiveness, the use of thermal paper other than that available from Hewlett-Packard may result in poor print quality and excessive print head wear.

CAUTION

Use only Hewlett-Packard paper in your HP-85 computer. Failure to do so may result in excessive print head wear.

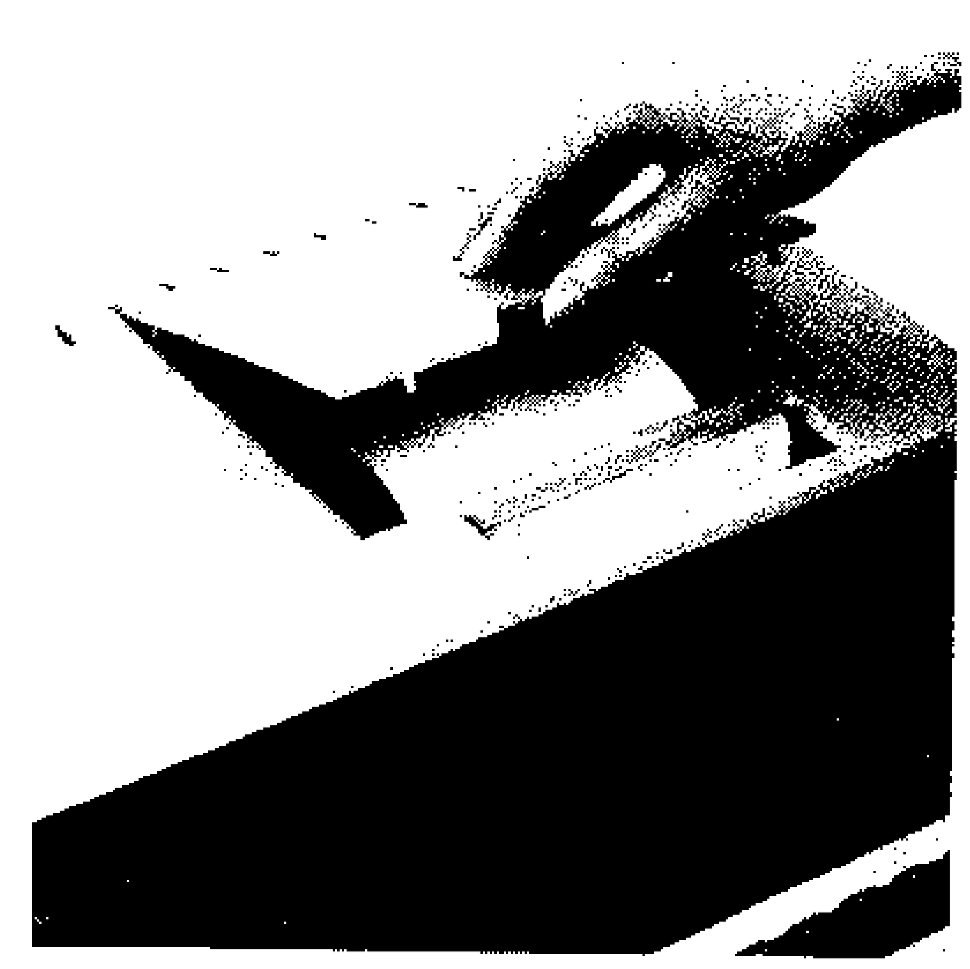
The heat-sensitive paper used in your HP-85 should be stored in a cool, dark place. Discoloration of paper may occur if it is exposed to direct sunlight for long periods of time, if storage temperatures rise above 65°C (149°F), if the paper is exposed to excessive humidity or to acetone, ammonia, alcohols, or other organic compounds, or if you attempt to erase anything on the paper. (Exposure to gasoline or oil fumes will not harm your HP-85 paper supply.)

Printed paper from your HP-85 will last 30 days or more without fading under fluorescent light, but to ensure the permanence of your records, you should store printed paper at room temperature in a dark place away from direct sunlight, heat, or fumes from organic compounds.

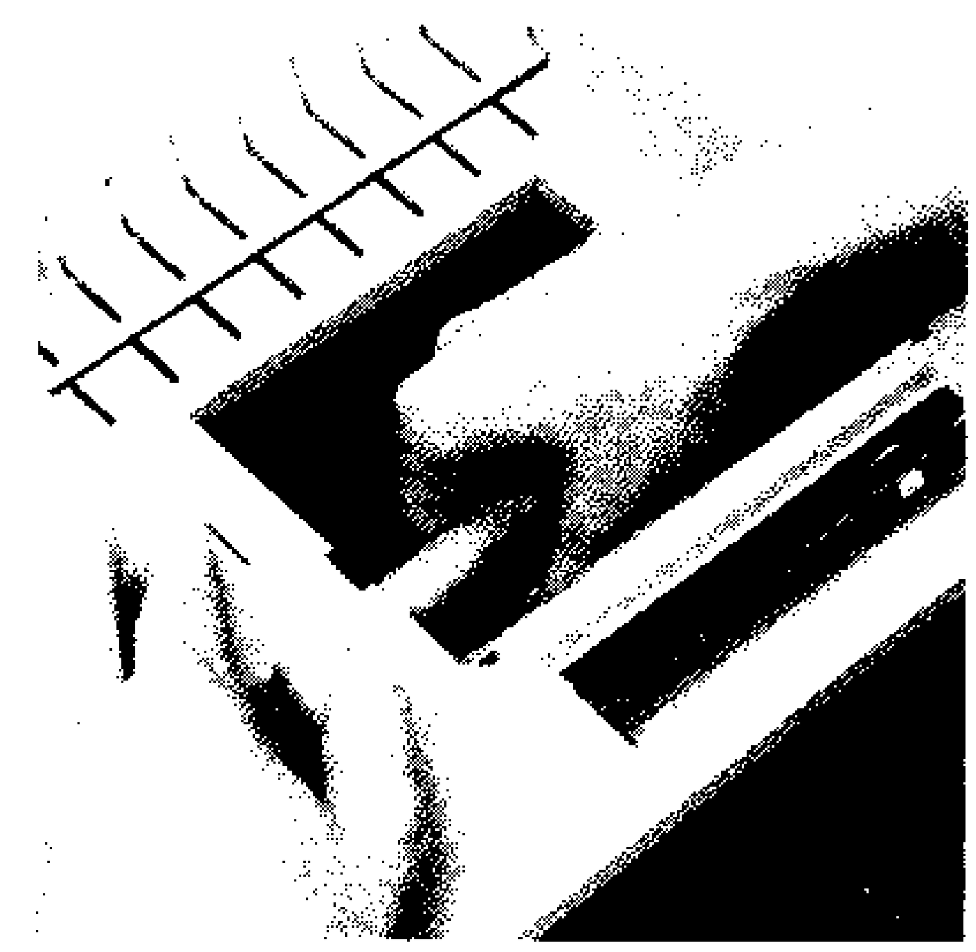
Loading Printer Paper

Printed paper is loaded by using the following procedure. To perform the following steps, the computer must be switched ON.

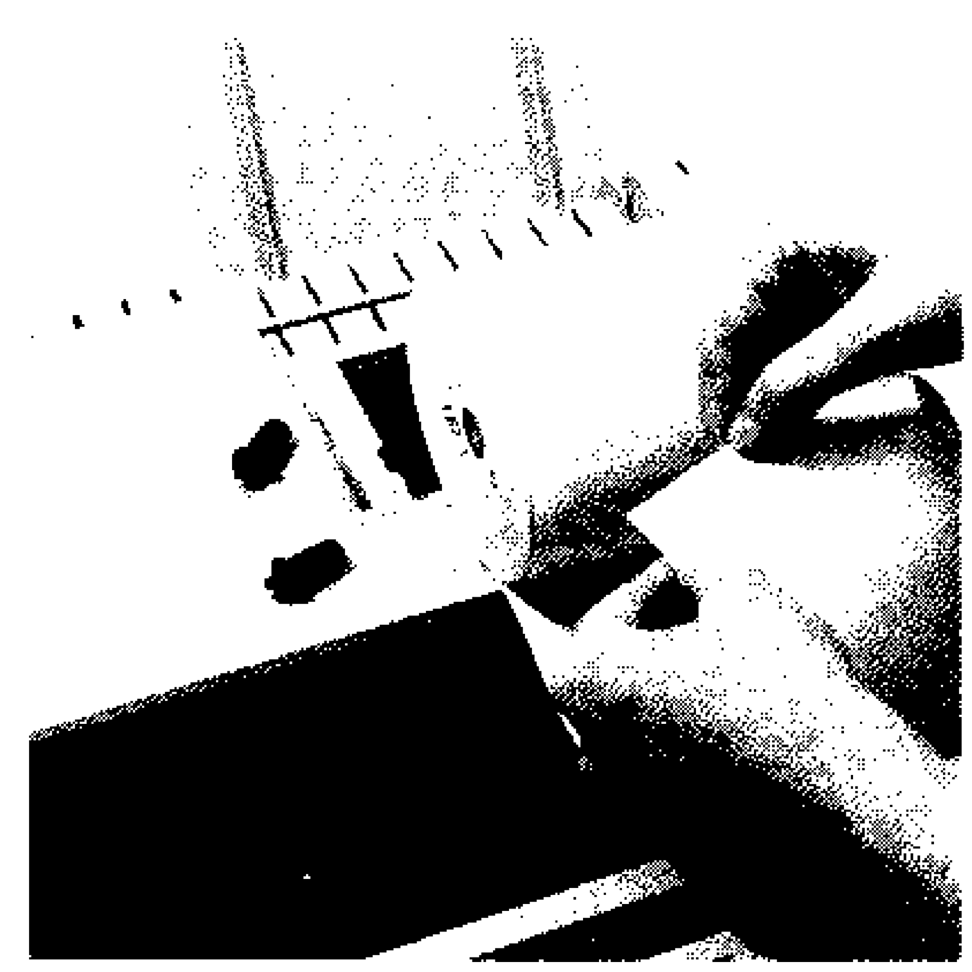
1. Open the hinged access cover by gently lifting the front edge of the cover up and back until it stops.



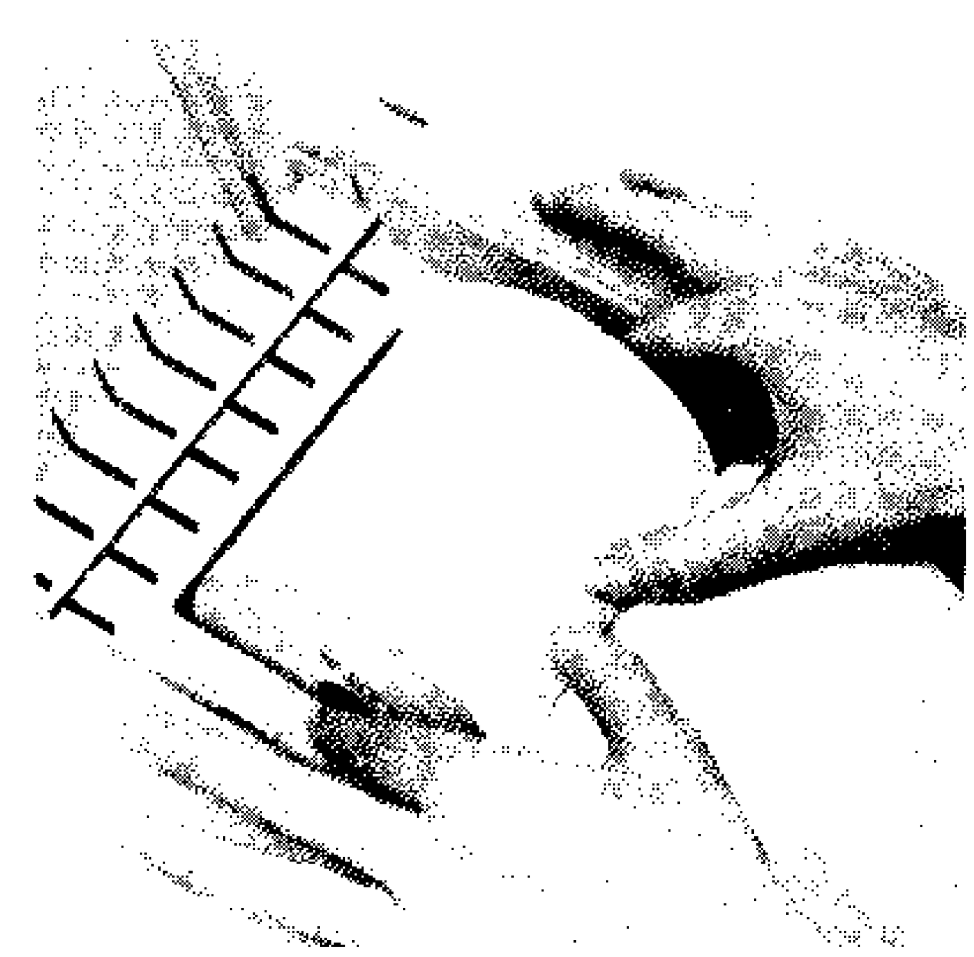
2. Remove the empty paper core with the roll guides from the paper well by pulling gently until the roll guides are released from their sockets. Discard the old paper core but save the roll guides at either end of the paper core. Remove any paper remaining from the previous roll by pressing the PAPER ADV key until the remaining paper stops moving. Then lift the paper out of the printer mechanism.



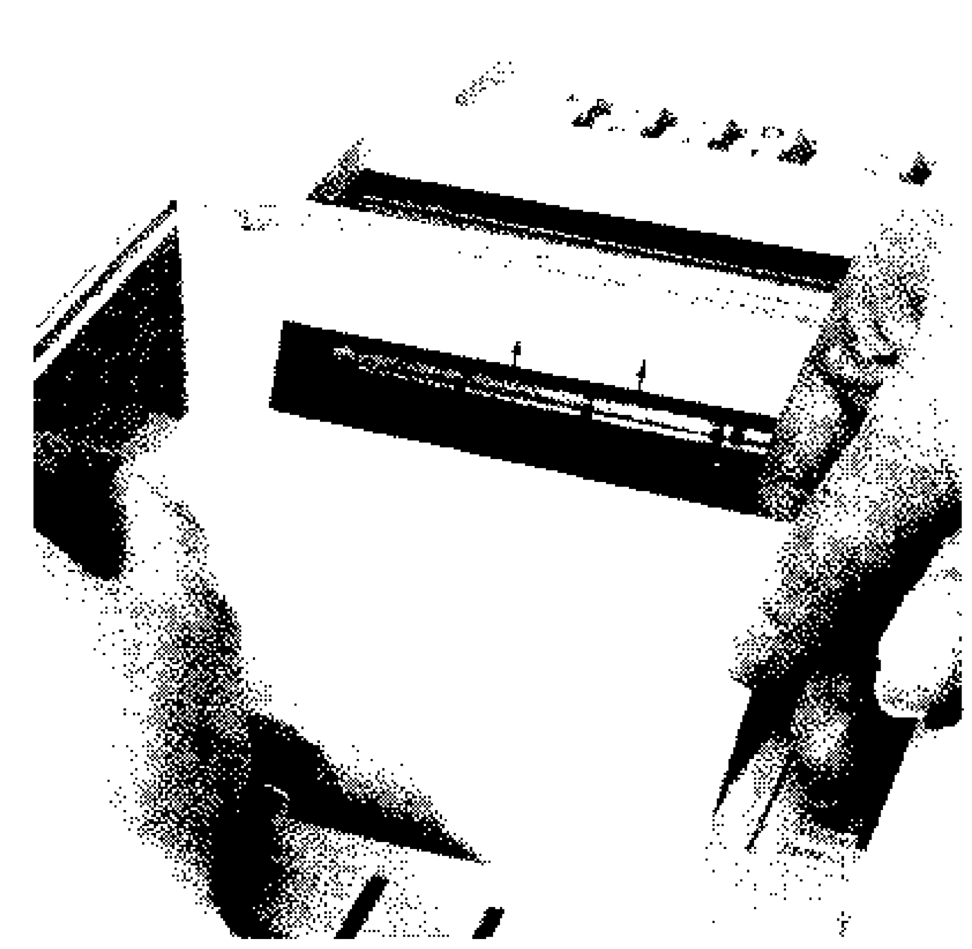
3. Discard the first 1-1/2 turns of the new roll to insure that no glue, tape, or other foreign matter is on the paper. Make sure that the leading edge of the paper is straight and cleanly cut or folded. A crooked or jagged leading edge will not engage properly in the paper advance rollers.



4. Insert the cylindrical ends of the roll guides into the core of the paper roll, aligning the tabs of the roll guides vertically. Using both hands to hold the roll guides in place, rest the paper roll on the paper well. Make sure that the leading edge of the paper is positioned to unroll forward from the bottom. Press inward on the roll guide tabs while pushing down on the paper roll, until the guides snap into place.



5. Pull approximately 6 inches of paper out of the roll and evenly insert the leading edge over and into the grey throat of the paper feed. Continue manually feeding the paper until it halts. Press and hold the paper advance key until the leading edge of the paper passes the top edge of the clear plastic tear bar. Close the hinged access cover, keeping the paper clear.



If the paper feeds properly through the printer mechanism but no printing appears on the tape when the printer is operated, the paper roll is probably inserted backwards. The paper is chemically treated and will print on one side only.

Printer Maintenance

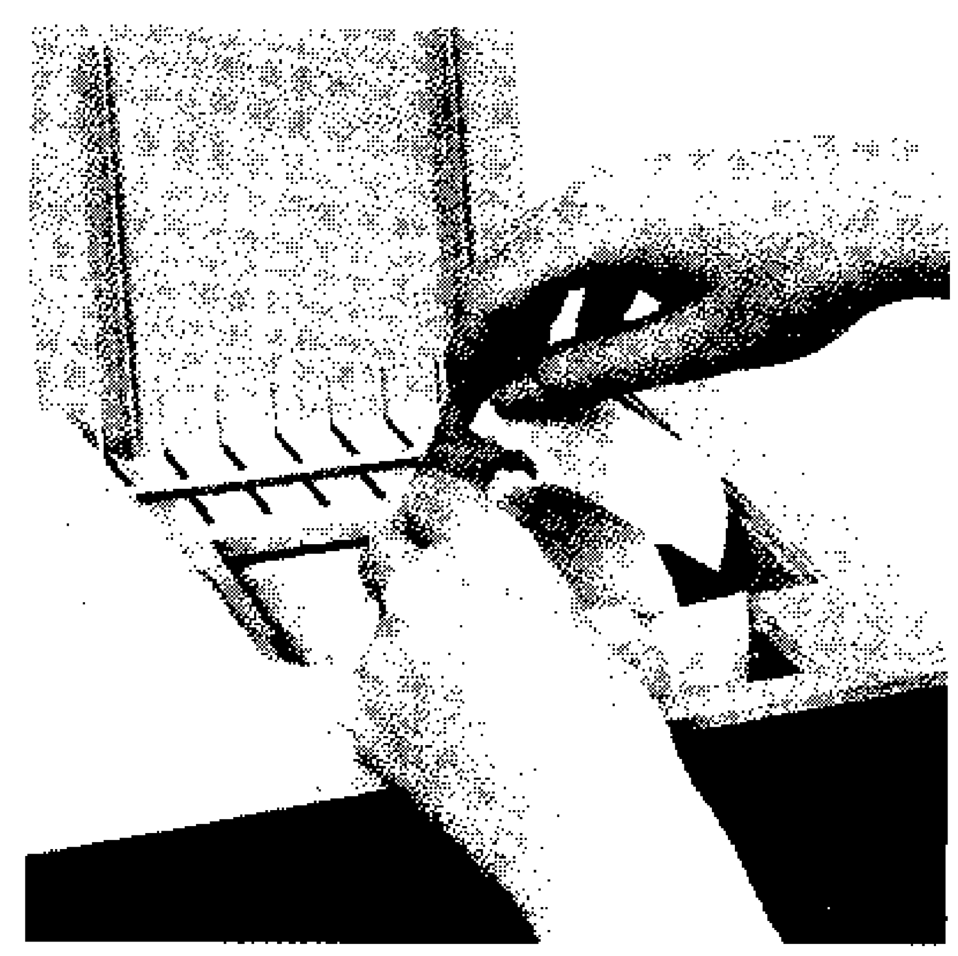
The printer in your HP-85, like the rest of the computer, is crafted for engineering excellence and is designed to give trouble-free operation with a minimum of maintenance. All moving parts in the printer mechanism have self-lubricating qualities. No lubrication, cleaning, or servicing of the mechanism is ever required. Setting the printer intensity dial to 5 or more for long periods of time may affect the long-term performance of the printhead. You can extend the life of the printer by setting the printer intensity dial to 4 or less.

CAUTION

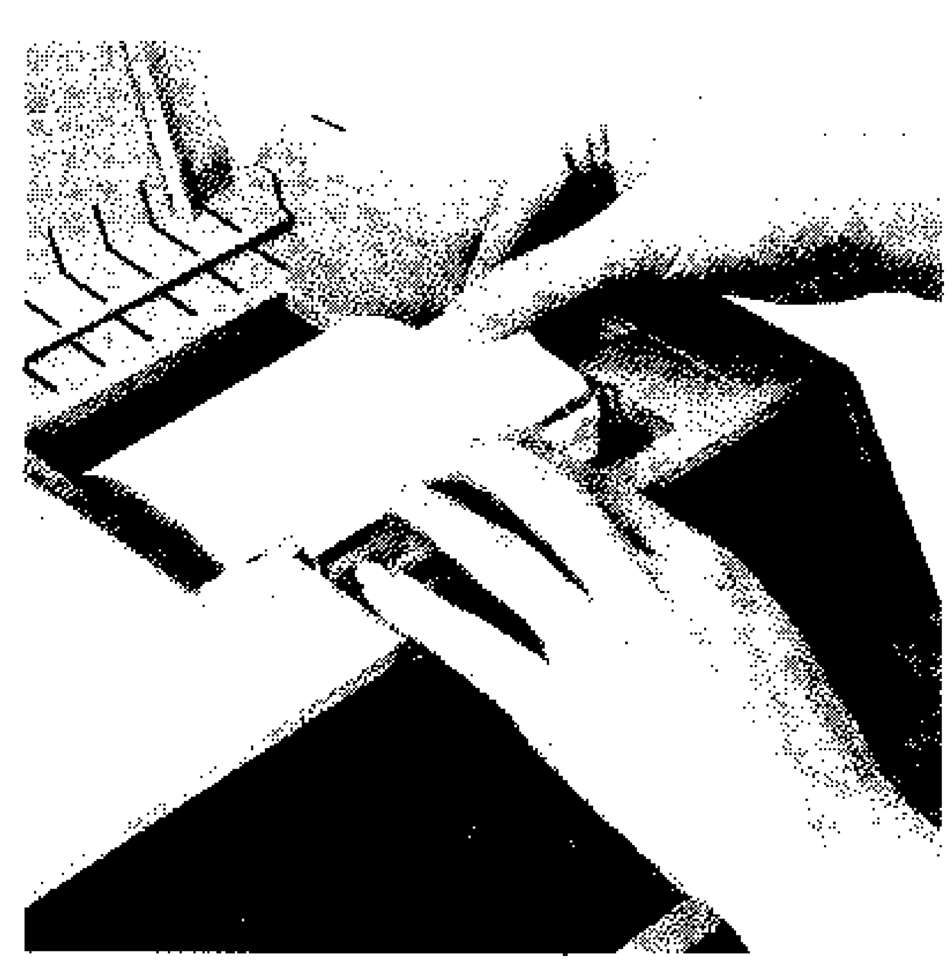
You should never attempt to insert a tool, such as a screwdriver, knifeblade, pencil, or other foreign object into the printer or its mechanism. Such actions can damage the platen, as well as other parts of the printer mechanism and will void your warranty.

If the printer paper should become jammed and fail to feed properly, first tear the jammed paper loose from the rest of the roll, then clear it by grasping the leading edge of the paper and pulling it *forward* through the printer mechanism while holding down the PAPER ADV key. Discard any lengths of paper damaged by tears or creases. Then reload as described above.

If the printer paper should become jammed with the leading edge below the tear bar, remove the clear plastic tear bar to reach the leading edge. Tear the jammed paper loose from the rest of the roll and pull it forward through the printer mechanism.



Before you replace the tear bar, reload the paper. Hold the paper back against the platen to ensure that the platen face will be properly located behind the tear bar. Then slide the tear bar back into place.



The Tape Cartridge

The tape cartridge used with the HP-85 computer is a high-quality digital storage medium. This section covers use, specifications, and care of tape cartridges.

Rewinding the Tape

The REW key or REWIND statement rewinds the tape to its beginning. Press SHIFT REW or type REWIND END LINE to rewind the tape.

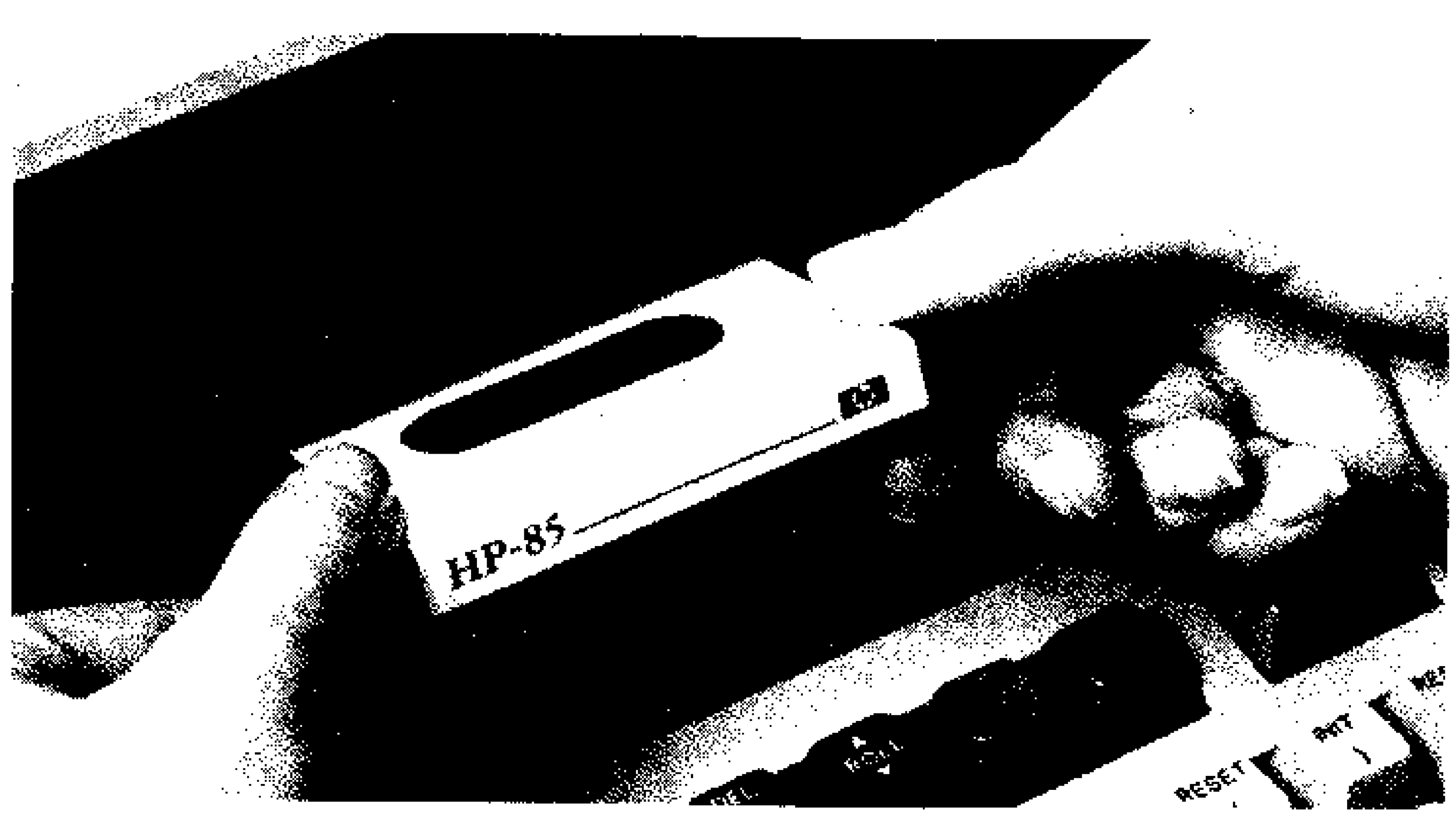
General Information for Use

Rewind time	29 seconds
Initialization time	15 seconds
Search speed	60 inches per second
Read/write speed	10 inches per second
Tape length	43 meters (140 feet)
Number of tracks	2 independent tracks
Typical tape capacity	780 program records (195K bytes) 850 data records (210K bytes)
Tape directory capacity	42 files (directory entries)
Typical access rate (search speed)	7,800 bytes/second
Typical transfer rate	650 bytes/second
Typical tape life (continuous use)	50 to 100 hours
Typical error rate*	<1 in 10^8 bits (that's less than one in every 100 million!)

Inserting the Tape Cartridge

Insert the tape cartridge so that its label is up and the open edge is toward the computer.

The tape drive door opens when the cartridge is pressed against it; the cartridge can then be inserted.



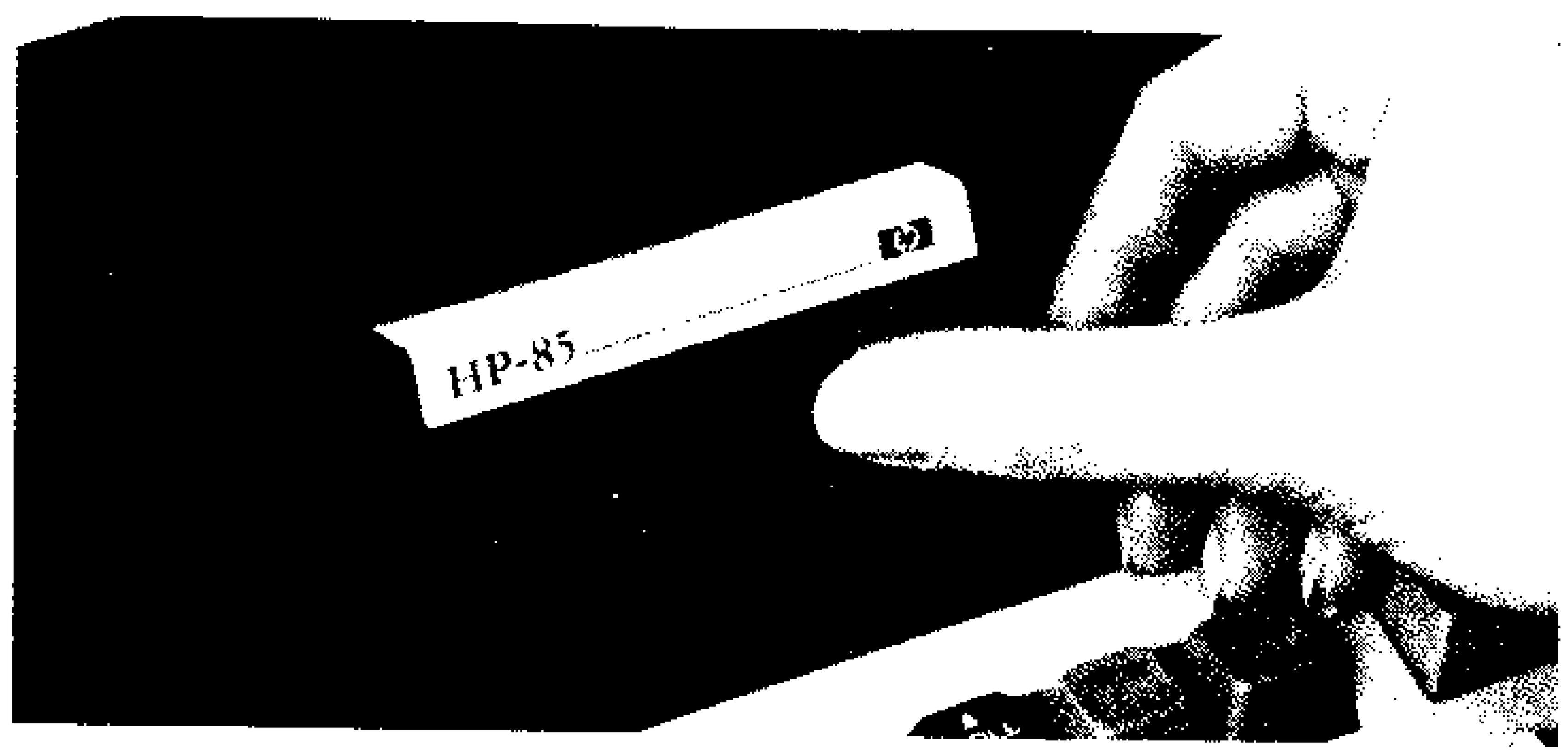
* This is dependent on the cleanliness of the tape head, tape care, and the cleanliness of the environment.

Removing the Tape Cartridge

The cartridge may be removed by pressing the bar below the tape drive. The tape drive will partially eject the cartridge so that you can remove it freely the rest of the way.

CAUTION

Do not attempt to remove the cartridge while the tape is in motion. Damage to the tape may result.



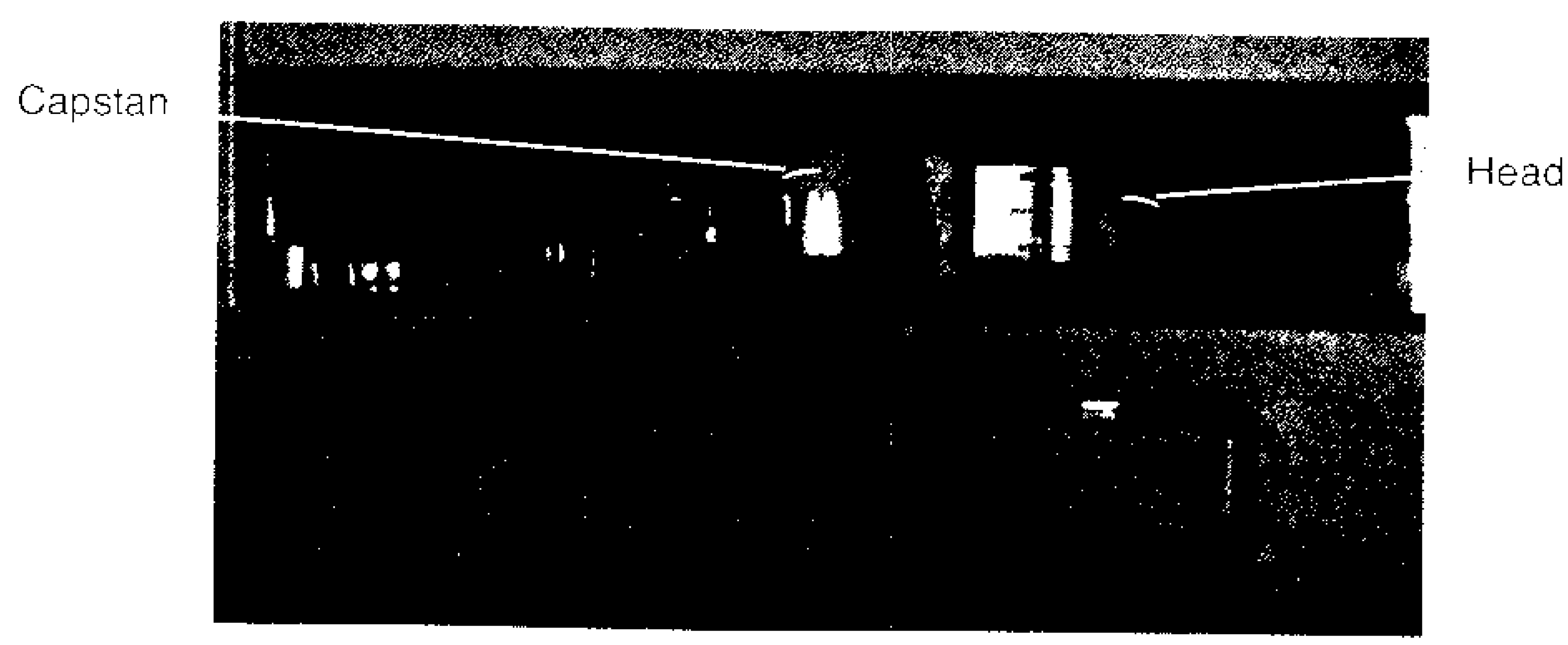
Write Protection

You may protect your cartridge against write (STORE or PRINT#) operations by sliding the RECORD slide tab to the left before inserting the cartridge. To record on the cartridge, the tab must be all the way to the right.

Tape Care

The cartridge tape drive may develop a buildup of oxide on the recording head after extensive use. As dirty tape drives are one of the most common cause of cartridge-related errors, the following basic precautions are aimed at reducing the risk of cartridge problems in your HP-85.

- Clean the tape head and the tape drive capstan at least as often as every 8 hours of cumulative tape use, or more frequently in dirty environments. Use a cotton-tipped swab dampened with isopropyl alcohol, wiping the tape head and the capstan with a light lateral (back-and-forth) motion (not a heavy scrubbing or up-and-down motion).



The head is the shiny surface on the right rear of the drive.

After using the head cleaning solution, wipe the tape head clean of any residue or lint with a dry cotton swab using a lateral motion (not an up-and-down motion). Be sure the head is dry before inserting a cartridge in the drive. It is a good idea to clean the head before making an important recording.

- Remove the tape cartridge when you are not using the computer. If a cartridge is left in, a flat spot may develop on the rubber wheel of the tape drive capstan in the tape drive of your HP-85. This condition will cause errors when using the tape. The dent is only temporary, and may be corrected by "conditioning" the tape, as described below.

As a normal operating guideline, it is a good practice to run tapes through a conditioning process after every 6 to 8 hours of use. "Conditioning" a tape means to run the tape forward to the end of the tape, reverse it, and run the tape backward to the beginning of the tape. This is done by inserting the tape cartridge to be conditioned, and executing the CTAPE *(condition tape)* command. CTAPE will not affect any programs or data on the tape.

CTAPE (END LINE)

Conditioning is necessary for smooth, continuous operation of the cartridge. (By warming up the tape drive capstan, conditioning also helps to remove a dent caused by leaving a cartridge in the drive.) Whenever a cartridge has been subjected to sudden environmental changes (such as being transported by air), you should condition the tape before use. Also if a READ error occurs while reading a particular cartridge, it may be due to uneven tension on the tape. Conditioning restores proper tension, and the tape will operate smoothly. If READ errors still occur after conditioning, try cleaning the tape head as described above.

- Keep the cartridge in the plastic container supplied with it.
- **Never** eject the tape cartridge while it is moving. Damage to information can be severe if a write or directory operation is in progress.

CAUTION

Strong magnetic fields can erase programs and data stored on tape. Where conditions warrant, keeping cartridges in a metal box, such as a card index, will help protect tapes from potential sources of magnetic damage.

Physical damage to tapes, such as wrinkles or folds, can cause recording and reading problems.

Tape Life

The tape cartridge has a typical life span of 50 to 100 hours of cumulative use. Environmental conditions of 25°C (77°F) and 20 to 50% relative humidity are most favorable for long tape life. A high duty cycle (percent of time the tape is accessed during the total time the computer is in use), high turning resistance, and continuous use for long periods of time (1/2 to 3 hours) contribute to heat buildup in cartridges and decrease tape life. Because tape cartridges eventually wear out, it is always a good practice to maintain back-up copies of vital programs and data using cartridges specifically reserved for this purpose.

If READ errors begin to occur frequently when using a tape cartridge, it is advisable that steps be taken to prevent the loss of information stored on the tape. The first step is to clean the tape as discussed previously in this section. If this does not alleviate the problem, the next step is to transfer the information to a new medium and retire the worn tape. Continued use could cause loss of information or damage to the tape drive itself.

STALL errors (signifying tape transport error caused by motor overload) can occur when either the tape drive or the cartridge itself fails. To determine the source of the problem, a different cartridge can be inserted. If STALL errors stop occuring, assume the cartridge itself is bad and replace it. If STALL errors continue to occur, the drive may require servicing. In this case, contact your HP dealer for assistance.

Tape cartridges that have reached the end of their useful life exhibit some specific danger signals:

1. The oxide starts breaking loose from the mylar backing of the magnetic tape.
2. The cartridge drive belt becomes loose, evidenced by the tape winding unevenly on the tape reels. This condition can be seen through the top of the cartridge. (Slight unevenness is common; you should be concerned when the tape is uneven by a quarter of the width of the tape.)
3. The drive pulley of the tape cartridge contains dark spots due to slippage. In severe cases, the cartridge may stall and the capstan will wear a flat spot on the drive pulley.

4. The cartridge rattles rather than making a constant hum when any tape movements occur.
5. You begin to get recurring READ errors or STALL errors.

If any of the above five danger signals occur, you should replace the cartridge at once. If you continue to use a cartridge under these circumstances, there is a chance that you could lose all the information on your cartridge and that you could damage the tape transport itself.

CAUTION

Ignoring READ, STALL, EOF, or SEARCH errors in ON ERROR routines is not recommended. These errors can signify tape transport problems. Overriding any of them could easily damage the transport.

Tape Cartridge Rethreading

If the tape runs off of the cartridge reel, it either signifies a tape transport problem or the light path in the cartridge is being obstructed. Do not block the light window of the cartridge, because the tape will not operate properly. Tape rethreading is difficult and is not recommended unless the data recorded on the runoff tape must be recovered. Instead, if tape runoff occurs, it is recommended to replace the entire tape cartridge. The rethreading procedures contained in this paragraph are for rethreading tape onto the tape cartridge's left tape hub. If a tape runoff condition occurs from the right tape hub, use the left-hand instructions and change all counterclockwise directions to clockwise directions. This procedure requires the use of a small Pozidrive screwdriver. Rethread tape onto the left tape hub as follows:

CAUTION

Whenever the tape cartridge top cover is removed, the spring-loaded door and spring can easily slide off the door pivot post. To prevent loss of parts, ensure that door is always completely seated on its pivot post as long as the tape cartridge top cover and backplate are separted.

a. Remove tape cartridge top cover by removing four screws from backplate with Pozidrive (small Phillips-head) screwdriver.

b. As shown in figure A, rethread loose end of tape around right tape guide, past belt drive pulley, outside front guide pin, and around left tape guide so that approximately1-3/4 inches of tape is clear of guide.

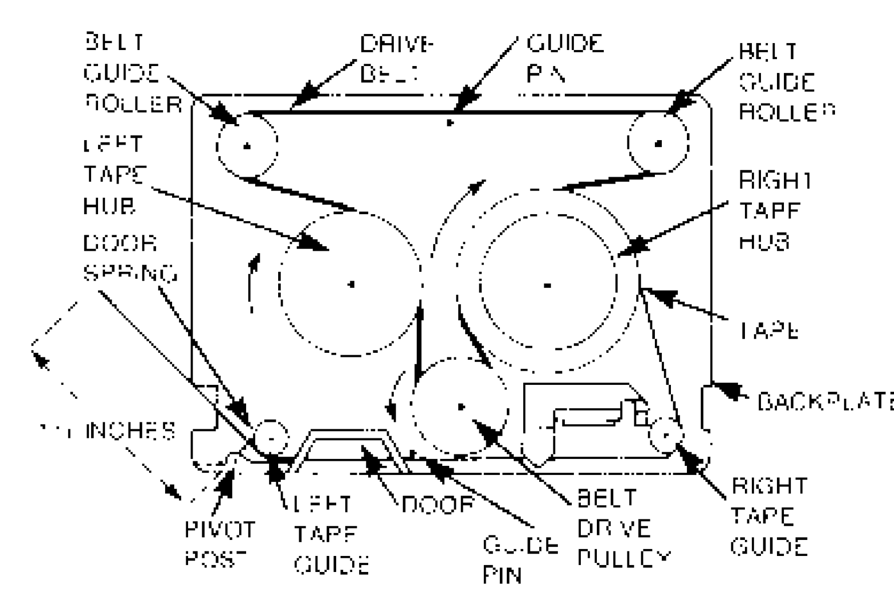


Figure A

c. Hold tape cartridge as shown in figure B, so that right hand can be used to rotate belt drive pulley and left hand can be used to maintain tape tension at tape guide.

d. Moisten inside surface of free end of tape and, while maintaining tape tension at left tape guide, rotate belt drive pulley counterclockwise to wrap free end of tape around left tape hub until tape reaches point where drive belt touches tape hub.

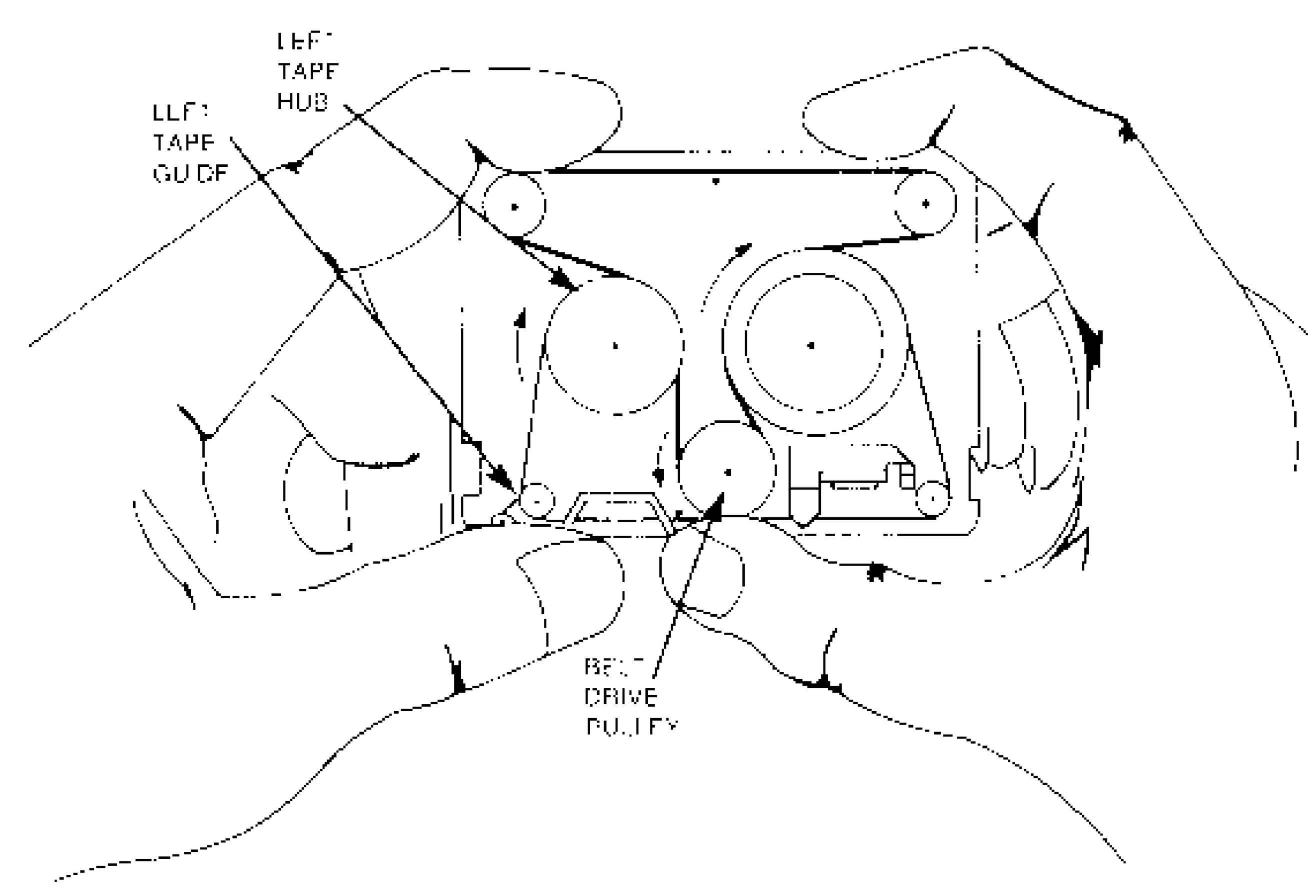


Figure B

e. While maintaining tape tension, use any small round-tipped tool to trap free end of tape between drive belt and left tape hub as shown in figure C.

f. Rotate belt drive pulley counterclockwise until tape is wrapped several times around left tape hub past first set of tape holes (approximately 2 feet). Check the tape pulleys to be sure they are not riding up.

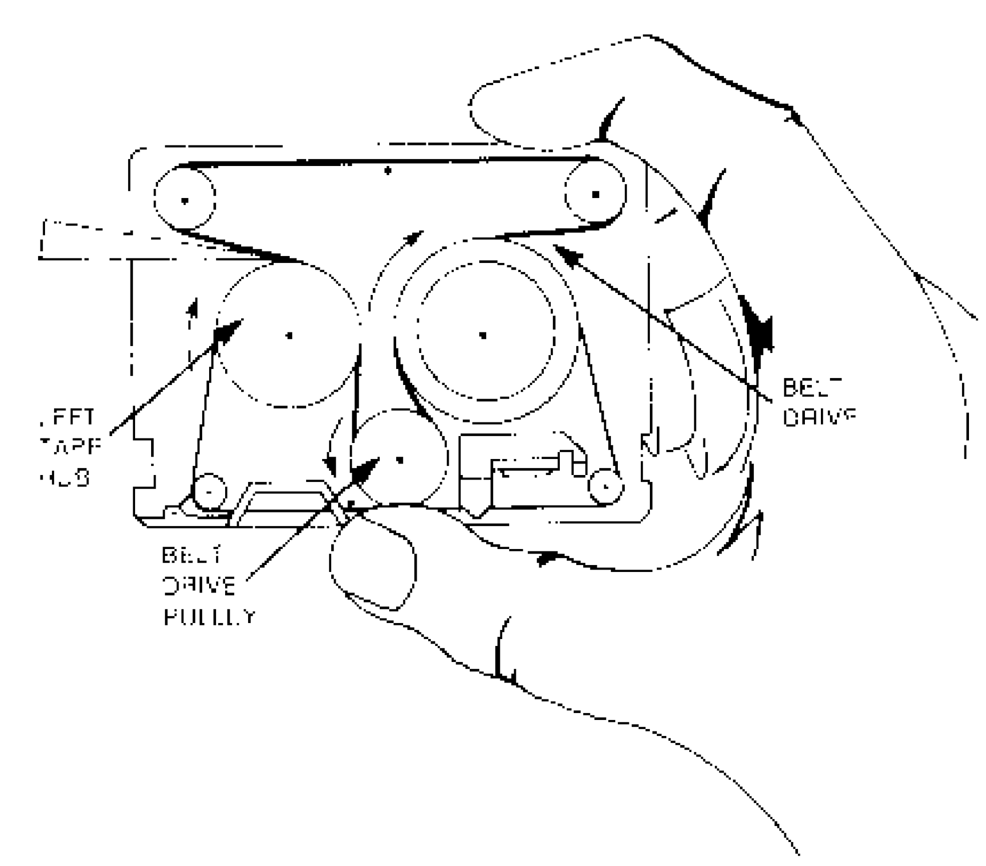


Figure C

g. Replace tape cartridge top cover on backplate and secure in place with four screws.

h. Condition tape in accordance with the instructions contained under Tape Care (page 282).

Optimizing Tape Use

A tape cartridge has two tape tracks with a variable number of records available in consecutively numbered files on each track, depending on the nature of your program and data storage requirements. The first file on track A and the first file on track B are both at the same end of the tape. This can cause a situation in which one file spans two tracks, making access to this file both time-consuming and wearing to the tape.

Track A	DIRECTORY	FILE 1	FILE 2	• • •	FILE J
Track B	FILE J	FILE J + 1	• • •	FILE N − 1	FILE N

When this happens, it is a good idea to first label the file spanning both tracks as a "dummy" and then store the program or data again, using the following procedure:

```
LOAD "file"
RENAME "file" TO "DUMMY"
STORE "file"
```

The file named DUMMY will then span both tracks and will not be accessed. However, the STORE command causes the same program or data to be stored under the desired file name on the second track, immediately after the end of file DUMMY. Now, when accessing the program or data material in this file, the time loss and additional wear to the tape cartridge caused by running back and forth between tape tracks is avoided.

Operational Considerations

General Cleaning

Disconnect the HP-85 from its ac power source before cleaning.

The HP-85 can be cleaned with a soft cloth dampened either in clean water or in water containing a mild detergent. Do not use an excessively wet cloth, nor allow water inside the computer. Do not use any abrasive cleaners, especially on the tape cartridge window or the CRT screen.

The tape head should be cleaned after a maximum of 8 hours of use; refer to Tape Care, page 281.

Selecting a Workspace

HP-85 computers are designed to operate on a flat, hard surface such as a desk or table top. Any workspace you choose for your HP-85 should allow the following minimum clearance dimensions for adequate air circulation around and within the instrument:

15 cm (6 in) both sides
15 cm (6 in) rear panel
15 cm (6 in) overhead

> **CAUTION**
>
> Always keep the top of the computer free of books, papers and other materials to avoid obstructing the air circulation vents built into the cover.

Potential for Radio/Television Interference

The HP-85 generates and uses radio frequency energy and may cause interference to radio and television reception. Your computer complies with the specifications in Subpart J of Part 15 of the FCC Rules for a Class B computing device. These specifications provide reasonable protection against such interference in a residential installation. However, there is no guarantee that interference will not occur in a particular installation. If the HP-85 does cause interference to radio or television reception, which can be determined by turning the computer off and on, you can try to eliminate the interference problem by doing one or more of the following:

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

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

Temperature Ranges

Temperature ranges for the HP-85 computer are:

Operating	5° to 40°C	40° to 105°F
Storage	−40° to 65°C	−40° to 150°F

Service

If at any time you suspect that the computer is malfunctioning, the following information will help you determine whether or not servicing is needed. If you are not familiar with the first part of this appendix, review it before proceeding with this section.

Display

If the CRT display blanks out or becomes erratic, or if the computer fails to respond to keyboard commands, turn the computer off and check the following:

1. Ensure that the voltage selector switch is set to the correct nominal line voltage for your area (115 Vac or 230 Vac).
2. Ensure that the correct fuse is installed for the power supply in your area (750mA for 115 Vac; T400mA for 230 Vac); and that the fuse is intact.
3. Unplug the power cord and inspect the power contacts on both power cord and the computer. Clean them if necessary.
4. Make sure that the power cord is securely plugged into both the computer and an earth-grounded ac outlet.
5. Adjust the brightness control on the computer's rear panel for optimum display clarity.

If, after step 5, the display fails to respond properly, service is required. (Refer to warranty information on the following pages.)

Tape Drive

If a STALL error appears on the display, or if the tape transport fails to operate, check the following:

1. Remove and examine the tape cartridge for defects. If any are found, discard the cartridge.
2. Clean the tape head as described under Tape Operations earlier in this appendix.
3. Test the tape transport using a fresh tape cartridge.

If, after step 3, the tape transport fails to operate properly, servicing is required. (Refer to the following warranty information.)

Printer

If the thermal printer fails to operate properly, follow the procedures outlined under Printer Operation in this appendix. If the printer continues to malfunction, servicing is required. (Refer to the following warranty information.)

Internal Timer

The HP-85 internal timer was checked at the factory to meet an initial accuracy of within 1 second per hour. Because of the effects of temperatures variations, aging, shocks, and vibrations on its quartz-crystal time standard, the HP-85 timer accuracy may vary slightly.

Warranty Information

The complete warranty statement is included in the information packet shipped with your HP-85. Please retain this statement for your records.

If you have any questions concerning this warranty, **please contact the authorized HP-85 dealer or the HP sales and service office from which you purchased your HP-85.** Should you be unable to contact them, please contact:

In the U.S.:

Hewlett-Packard
Corvallis Division • Customer Support
1000 N.E. Circle Blvd.
Corvallis, Oregon 97330
Tel. (503) 758-1010

Toll Free Number (6 a.m. to 6 p.m., Pacific Time):
Call 800/547-3400 (except in Alaska and Hawaii).

In Europe:

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

Tel. (022) 82 70 00

Other countries:

Hewlett-Packard Intercontinental
3495 Deer Creek Rd.
Palo Alto, California 94304
U.S.A.

Tel. (415) 857-1501

For world-wide HP sales and service offices, please refer to the back of the manual.

How to Obtain Repair Service

Not all Hewlett-Packard facilities offer service for the HP-85. For information on obtaining service in your area, consult the service information included in the information packet shipped with your HP-85. Or contact your authorized HP dealer or the nearest Hewlett-Packard sales and service facility. (Addresses are listed in the back of the manual.)

If your HP-85 malfunctions and repair is required, you can help assure efficient servicing by following these guidelines:

1. Leave configuration of the HP-85 exactly as it was at the time of the malfunction, i.e., any plug-in modules and tape cartridges in use at that time should remain intact.
2. Write a description of the malfunction symptoms for Service personnel.
3. Save printouts or any other materials that illustrate the problem area.
4. Have on hand a sales slip or other proof of purchase to establish warranty coverage period.

Serial Number

Each HP-85 computer carries an individual serial number on the plate in the upper right-hand corner of the rear panel. It is recommended that owners keep a separate record of this number. Should your unit be lost or stolen, the serial number is often necessary for tracing and recovery, as well as any insurance claims. Hewlett-Packard does not maintain records of individual HP-85 owner's names and unit serial numbers.

General Shipping Instructions

Should you ever need to ship your HP-85, be sure it is packed in a protective package (use the original shipping case), to avoid in-transit damage. Such damage is not covered by the warranty. Hewlett-Packard suggests that the customer always insure shipments.

Further Information

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

Should other problems or questions arise regarding repairs, please call your nearest Hewlett-Packard sales and service facility or your authorized HP-85 dealer.

Note: Not all Hewlett-Packard repair facilities offer service for HP-85 computers. However, you can be sure that service may be obtained in the country where you bought your computer.

If you happen to be outside of the country where you bought your computer, contact the nearest authorized HP-85 dealer or the local Hewlett-Packard center. All customs and duties are your responsibility.

Reference Tables

Reset Conditions

The following table shows the status of specific functions when the indicated commands are executed. Parentheses in the POWER ON column indicate the values when the system is turned on. "R" designates a function restored to POWER ON values. "—" designates a function unchanged from its status prior to executing the command. "U" designates that variables are assigned undefined values, except those in COMmon.

	POWER ON	RESET	SCRATCH	RUN	CHAIN	INIT	CONT
Program variables	R(none)	—	R	U	U	U	—
Calculator variables	R(none)	R	R	R	R	R	R
Result	R(zero)	R	R	—	—	—	—
Trigonometric Mode	R(RAD)	R	—	—	—	—	—
Typing Mode	R(BASIC)	—	—	—	—	—	—
PRINT ALL mode	R(off)	R	—	—	—	—	—
Output device	R(PRINTER IS 2 CRT IS 1)	R	—	—	—	—	—
Special function key definitions	R(none)	R	R	R	R	R	—
DATA pointers	R(none)	R	R	R	R	R	—
Default values	R(DEFAULT ON)	R	—	—	—	—	—
System timer							
TIME	R(zero)	—	—	—	—	—	—
DATE	R(zero)	—	—	—	—	—	—
Random Number Seed	R	R	—	—	—	—	—
KEY LABEL	R(none)	R	R	R	R	R	—
ON TIMER	R(off)	R	R	R	R	R	—
ON ERROR	R(off)	R	R	R	R	R	—
TRACE	R(off)	R	R	—	—	—	—
TRACE VAR	R(off)	R	R	—	U	—	—
TRACE ALL	R(off)	R	R	—	—	—	—
Binary programs	R(none)	—	R	—	—	—	—
SCALE	R(0,100,0,100)	R	—	—	—	—	—
ROMS	R(initialize)	R	—	—	—	—	—
PEN	R(positive)	R	—	—	—	—	—
PENUP	R(up)	R	—	—	—	—	—
Last plotted point	R(0,0)	R	—	—	—	—	—
LDIR	R(horizontal)	R	—	—	—	—	—
ASSIGN# buffers	R(none)	R	R	R	—	R	—
COMmon variables	R(none)	—	R	R	—	R	—

HP-85 Character and Key Codes

A numeric code is attached to each character and/or key. The first column of characters in the table below may be accessed by holding down the (CTRL) key while typing the character denoted by the superscript "c." Five characters in the last column are not apparent from the keyboard. They may be accessed by holding down the (SHIFT) key while typing the character or key denoted by the superscript "s."

In addition, each of the characters in the table below has a complementary underscored character with a decimal-value of 128 larger than its given decimal value. The CHR$ function and certain keys on INPUT enable you to access the underscored characters. For instance CHR$(74+128) is J̲.

Char.	Binary	Octal	Dec	Char.	Binary	Octal	Dec	Char.	Binary	Octal	Dec	Char.	Binary	Octal	Dec
π @^{c}	00000000	000	0	SPACE	00100000	040	32	@	01000000	100	64	` (KEY LABEL)^s	01100000	140	96
÷ A^c	00000001	001	1	!	00100001	041	33	A	01000001	101	65	a	01100001	141	97
× B^c	00000010	002	2	"	00100010	042	34	B	01000010	102	66	b	01100010	142	98
√ C^c	00000011	003	3	#	00100011	043	35	C	01000011	103	67	c	01100011	143	99
α D^c	00000100	004	4	$	00100100	044	36	D	01000100	104	68	d	01100100	144	100
β E^c	00000101	005	5	%	00100101	045	37	E	01000101	105	69	e	01100101	145	101
Γ F^c	00000110	006	6	&	00100110	046	38	F	01000110	106	70	f	01100110	146	102
↓ G^c	00000111	007	7	'	00100111	047	39	G	01000111	107	71	g	01100111	147	103
Δ H^c	00001000	010	8	(	00101000	050	40	H	01001000	110	72	h	01101000	150	104
σ I^c	00001001	011	9	)	00101001	051	41	I	01001001	111	73	i	01101001	151	105
↑ J^c	00001010	012	10	*	00101010	052	42	J	01001010	112	74	j	01101010	152	106
λ K^c	00001011	013	11	+	00101011	053	43	K	01001011	113	75	k	01101011	153	107
μ L^c	00001100	014	12	,	00101100	054	44	L	01001100	114	76	l	01101100	154	108
CR M^c	00001101	015	13	–	00101101	055	45	M	01001101	115	77	m	01101101	155	109
τ N^c	00001110	016	14	.	00101110	056	46	N	01001110	116	78	n	01101110	156	110
Φ O^c	00001111	017	15	/	00101111	057	47	O	01001111	117	79	o	01101111	157	111
Θ P^c	00010000	020	16	0	00110000	060	48	P	01010000	120	80	p	01110000	160	112
Ω Q^c	00010001	021	17	1	00110001	061	49	Q	01010001	121	81	q	01110001	161	113
δ R^c	00010010	022	18	2	00110010	062	50	R	01010010	122	82	r	01110010	162	114
Ȧ S^c	00010011	023	19	3	00110011	063	51	S	01010011	123	83	s	01110011	163	115
ȧ T^c	00010100	024	20	4	00110100	064	52	T	01010100	124	84	t	01110100	164	116
Ä U^c	00010101	025	21	5	00110101	065	53	U	01010101	125	85	u	01110101	165	117
ä V^c	00010110	026	22	6	00110110	066	54	V	01010110	126	86	v	01110110	166	118
Ö W^c	00010111	027	23	7	00110111	067	55	W	01010111	127	87	w	01110111	167	119
ö X^c	00011000	030	24	8	00111000	070	56	X	01011000	130	88	x	01111000	170	120
Ü Y^c	00011001	031	25	9	00111001	071	57	Y	01011001	131	89	y	01111001	171	121
ü Z^c	00011010	032	26	:	00111010	072	58	Z	01011010	132	90	z	01111010	172	122
Æ [^{c}	00011011	033	27	;	00111011	073	59	[	01011011	133	91	π (/)^s	01111011	173	123
æ \\$^c$	00011100	034	28	<	00111100	074	60	\	01011100	134	92	¦	01111100	174	124
≠]^c	00011101	035	29	=	00111101	075	61	]	01011101	135	93	→ (−)^s	01111101	175	125
£ ∧^{c}	00011110	036	30	>	00111110	076	62	^	01011110	136	94	Σ (✱)^s	01111110	176	126
▒ _^{c}	00011111	037	31	?	00111111	077	63	_	01011111	137	95	⊢ (+)^s	01111111	177	127

Key Response During Program Execution

Decimal codes above 128 are assigned to program, editing, and system control keys. The table below describes the response of the system when the specified key is pressed during the execution of a running program and its response to an INPUT statement.

Key Codes

Key	Response in ALPHA Mode	Response in Graphics Mode	Decimal Value
k1			128
k2			129
k3			130
k4			131
k5			132
k6			133
k7			134
k8			135
REWIND			136
COPY	A/L	A/L	137
PAPER ADVANCE	A/L	A/L	138
RESET	A/L	A/L	139
INIT			140
RUN			141
PAUSE	A/L	A/L	142
CONT			143
STEP			144
TEST			145
CLEAR	A/L	A/L	146
GRAPH	A/L	A/L	147
LIST			148
PLIST			149
KEY LABEL	A/L	A/L	150
not used			151
not used			152
BACK SPACE	A	A	153
ENDLINE	A	A	154
SHIFT BACK SPACE	A		155
Cursor left	A		156
Cursor Right	A		157
Roll up	A/L	A/L	158
Roll down	A/L	A/L	159
-Line	A		160
Cursor up	A		161
Cursor down	A		162
INS/RPL	A		163
-CHR	A		164
Cursor home	A		165
RESULT	A		166
not used			167
DELETE			168
STORE			169
LOAD			170
not used			171
AUTO			172
SCRATCH			173

L indicates that the specified key is live (i.e., performs its expected function) during the execution of a running program. All other keys halt a running program and then perform the indicated function.

A indicates that the specified key is active on INPUT. In other words, when the input prompt (?) appears, the keys designated by A perform their respective functions. All other keys output their respective character codes.

Glossary and BASIC Syntax Guidelines

The HP-85 BASIC language consists of **statements, functions, operators,** and **commands**. Operators and functions are used with variables, numbers, and strings to create numeric and string **expressions**. Expressions and functions can be included in statements and executed from the keyboard. Each statement can be preceded by a statement number and stored as a program statement. Most functions, statements, and commands can also be separately executed from the keyboard; exceptions are noted.

Operators

Arithmetic

+	Add
-	Subtract
*	Multiply
/	Divide
^	Exponentiate
MOD	Modulo: A MOD B=A-B*INT(A/B)
\ or DIV	Integer divide: A DIV B=IP(A/B)

Logical Evaluation

Logical expressions return the values 0 for false and 1 for true. Non-zero values are considered true; zero values are false.

Relational

=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<> or #	Not equal to

Non-numeric values can also be compared with relational operators. Strings are compared, character by character, from left to right until a difference is found. If one string ends before a difference is found, the shorter string is considered the lesser.

Logical

AND
OR
EXOR
NOT

String

& String concatenation

Math Hierarchy

( )	Performed First
Functions	
^	
NOT	
*, /, MOD, \ or DIV	
+, -	
Relational operators (=, >, <, >=, <=, <>, or #)	
AND	
OR, EXOR	Performed Last

Expressions are evaluated from left to right for operators at the same level. Operations within parentheses are performed first. Nested parentheses are evaluated inward out.

Data Precision

Precision	Accuracy	Range	Maximum array size with standard memory and no program.
REAL	12 Digits	±9.99999999999E±499	1800
SHORT	5 Digits	±9.9999E±99	3600
INTEGER	5 Digits	-99999 through 99999	4800

Special Characters

@ Enables multi-statement lines.

```
100 CLEAR @ KEY LABEL
```

! Remarks follows.

```
110 DISP C ! Display cost.
```

? INPUT prompt. Input items are expected.

Variables

Simple Numeric Variables: A1, B, C3.

The name consists of a letter or a letter and one digit. Real precision is assumed unless SHORT or INTEGER type is declared.

Arrays: A1(50,5), B(20,20), C3(10).

The name consists of a letter or a letter and one digit. An array name can be the same as a simple variable name used elsewhere in the program, but a one-dimensional array cannot have the same name as a two-dimensional array. Arrays contain numeric elements only. Subscripts dimension the row or row and column in DIM, COM, or type (REAL, INTEGER, SHORT) declaration statements. The lower bound of an array subscript is 0 unless OPTION BASE 1 is specified before all array references. The default upper bound for row and column subscripts is 10.

Subscripts reference a particular array element in non-declaratory statements with three exceptions. Entire arrays (either one- or two-dimensional) may be referenced in TRACE VAR, PRINT#, or READ# statements by specifying the array name followed by a pair of parentheses and no subscripts (e.g., C3( )). A comma may be enclosed within the parentheses for documentation purposes to specify a two-dimensional array (e.g., A1(,)). This notation enables you to trace, write onto tape, or read from tape all elements of the specified array.

String Variables: A1$, B$, C3$.

The name consists of a letter or a letter and one digit followed by a dollar sign. The default length is 18 characters unless otherwise specified in a COM or DIM statement. The maximum length of a string is limited only by available memory. Dimension strings in a DIM or COM statement by specifying the variable name followed by the length enclosed within brackets: A1$[25], B$[415], C3$[5].

Substrings: A1$[2,25], B$[5], C3$[3,3].

Substrings are specified by one or two numbers (or expressions) enclosed within *brackets*. One number specifies a beginning character; the substring extends to the end of the string. Two numbers separated by a comma specify beginning and ending character positions, respectively.

Strings can be compared with the relational operators and can be concatenated by & operator. (Refer to Operators.)

Syntax Guidelines to Commands and BASIC Statements

These terms and conventions are used in the following list of statements and commands.

dot matrix — All items in dot matrix denote system commands or BASIC statements that must appear exactly as shown. Either small or capital letters may be used to spell keywords.

[ ] — All items enclosed within square brackets are optional unless the brackets are in dot matrix.

... — Three dots indicate that the previous item can be repeated.

statement number — An integer from 1 through 9999.

numeric expression — A logical combination of variables, constants, operators, and functions (including user-defined functions), grouped within parentheses as necessary.

string expression — A logical combination of text within quotes, string variables, substrings, string concatenations, and string functions.

file name or **program name** — A program or file name can be any string expression. Any letter, number, symbol, or character except quotes or the null string may be used. Only the first six letters of the string expression are used for the name.

```
LOAD "PROG1$"
LOAD T$
```

buffer number — The number assigned to a tape data file by an ASSIGN# statement, and referenced by the PRINT# and READ# statements. Its range is 1 through 10.

Commands

Non-Programmable

AUTO [beginning line number [, increment value]]	Page 80
CONT [statement number]	Page 98
DELETE first statement number [, last statement number]	Page 95
INIT	Page 99
LOAD program name	Page 179
REN [first statement number [, increment value]]	Page 96
RUN [statement number]	Page 99
SCRATCH	Page 78
STORE program name	Page 176
UNSECURE file name , security code , secure type	Page 194

Programmable

CAT	Page 175
COPY	Page 35
CTAPE	Page 282
ERASETAPE	Page 175
FLIP	Page 34
LIST [beginning statement number [, ending statement number]]	Page 97
PLIST [beginning statement number [, ending statement number]]	Page 97
PRINT ALL	Page 35
REWIND	Page 280
SECURE file name , security code , secure type	Page 193

BASIC Statements

ASSIGN# buffer number TO file name	Page 183
ASSIGN# buffer number TO *	Page 184
BEEP [tone , duration]	Page 89
CHAIN file name	Page 179
CLEAR	Page 19
COM common variable list	Page 123
CRT IS output code number	Page 169
CREATE file name , number of records [, number of bytes per record]	Page 180
DATA data list	Page 137
DEFAULT OFF	Page 70
DEFAULT ON	Page 70
DEF FN numeric variable name [(parameter)][= numeric expression]	Page 145
DEF FN string variable name [(parameter)][= string expression]	Page 145
DEG	Page 66
DIM dimension list	Page 121
DISP display list	Page 84

DISP USING image format string [; disp using list]	Page 167
DISP USING statement number [; disp using list]	Page 161
END	Page 77
FN END	Page 147
FOR loop counter = initial value TO final value [STEP increment value]	Page 111
GOSUB statement number	Page 151
GOTO statement number	Page 91
GRAD	Page 66
IF numeric expression THEN statement number or executable statement [ELSE statement number or executable statement]	Page 106
IMAGE image format string	Page 161
INPUT variable name$_1$ [, variable name$_2$ ...]	Page 87
INTEGER numeric variable [(subscripts)][, numeric variable] (subscripts)] ...]	Page 122
KEY LABEL	Page 154
[LET] numeric variable$_1$ [, numeric variable$_2$ ...] = numeric expression	Page 90
[LET] string variable$_1$ [, string variable$_2$ ...] = string expression	Page 90
[LET] FN variable name = expression	Page 147
LOAD BIN file name	Page 193
NEXT loop counter	Page 111
NORMAL	Pages 35, 80
OFF ERROR	Page 261
OFF KEY# key number	Page 156
OFF TIMER# timer number	Page 157
ON ERROR GOSUB statement number	Page 261
ON ERROR GOTO statement number	Page 261
ON numeric expression GOSUB statement number list	Page 153
ON numeric expression GOTO statement number list	Page 108
ON KEY# key number [, key label] GOSUB statement number	Page 154
ON KEY# key number [, key label] GOTO statement number	Page 154
ON TIMER# timer number , milliseconds GOSUB statement number	Page 156
ON TIMER# timer number , milliseconds GOTO statement number	Page 156
OPTION BASE 1 or 0	Page 121
PAUSE	Page 99
PRINT [print list]	Page 86
PRINT# buffer number ; print # list	Page 185
PRINT# buffer number , record number [; print # list]	Page 188
PRINT USING image format string [; print using list]	Page 167
PRINT USING statement number [; print using list]	Page 161
PRINTER IS output code number	Page 169
PURGE file name [, purge code number]	Page 192
RAD	Page 66
RANDOMIZE [numeric expression]	Page 64
READ variable name$_1$ [, variable name$_2$ ...]	Page 137

READ# buffer number ; variable list	Page 187
READ# buffer number , record number [; variable list]	Page 190
REAL numeric variable [(subscripts)][, numeric variable [(subscripts)] ...]	Page 122
REM [any combination of characters]	Page 83
RENAME old file name TO new file name	Page 192
RESTORE [statement number]	Page 139
RETURN	Page 151
SETTIME seconds since midnight , Julian day in form yyddd	Page 56
SHORT numeric variable [(subscripts)][, numeric variable [(subscripts)] ...]	Page 122
STOP	Page 77
STORE BIN file name	Page 193
TRACE	Page 255
TRACE ALL	Page 256
TRACE VAR variable_1 [, variable_2 ...]	Page 255
WAIT number of milliseconds	Page 100

Graphics Statements

ALPHA	Page 197
BPLOT character string, number of characters per line	Page 237
DRAW x-coordinate , y-coordinate	Page 211
GCLEAR [y]	Page 199
GRAPH	Page 197
IDRAW x-increment , y-increment	Page 217
IMOVE x-increment , y-increment	Page 217
LABEL character string	Page 221
LDIR numeric expression	Page 224
MOVE x-coordinate , y-coordinate	Page 211
PEN numeric expression	Page 207
PENUP	Page 207
PLOT x-coordinate , y-coordinate	Page 208
SCALE xmin , xmax , ymin , ymax	Page 199
XAXIS y-intercept [, tic length [, xmin , xmax]]	Page 202
YAXIS x-intercept [, tic length [, ymin , ymax]]	Page 202

BASIC Predefined Functions

ABS(X)	Absolute value of X.	Page 60
ACS(X)	Arcosine of X, in 1st or 2nd quadrant.	Page 66
ASN(X)	Arcsine of X, in 1st or 4th quadrant.	Page 66
ATN(X)	Arctangent of X, in 1st or 4th quadrant.	Page 66
ATN2(Y,X)	Arctangent of Y/X, in proper quadrant.	Page 67
CEIL(X)	Smallest integer >=X.	Page 61
CHR$(X)	Character whose decimal character code is X, 0<=X <=255.	Page 131
COS(X)	Cosine of X.	Page 66
COT(X)	Cotangent of X.	Page 66

CSC(X)	Cosecant of X.	Page 66
DATE	Julian date in format yyddd (assumes system timer has been set properly).	Page 57
DTR(X)	Degree to radian conversion.	Page 67
EPS	Smallest positive machine number (1E-499).	Page 64
ERRL	Line number of latest error.	Page 261
ERRN	Number of latest error.	Page 261
EXP(X)	e^x	Page 65
FLOOR(X)	Same as INT(X) (relates to CEIL).	Page 61
FP(X)	Fractional part of X.	Page 60
INF	Largest machine number (9.99999999999E499).	Page 64
INT(X)	Largest integer <=X.	Page 61
IP(X)	Integer part of X.	Page 60
LEN(S$)	Length of string S$.	Page 128
LGT(X)	Log to the base 10 of X, X>0.	Page 65
LOG(X)	Natural logaritm, X>0.	Page 65
MAX(X,Y)	If X>Y then X, else Y.	Page 62
MIN(X,Y)	If X<Y then X, else Y.	Page 62
NUM(S$)	Decimal character code of first character of S$.	Page 132
PI	3.14159265359	Page 63
POS(S1$, S2$)	Searches string S1$ for the first occurrence of string S2$. Returns starting index if found, otherwise returns 0.	Page 129
RMD(X,Y)	Remainder of X/Y: X − Y * IP(X/Y).	Page 62
RND	Next number, X, in a sequence of pseudo-random numbers, 0 < = X < 1.	Page 64
RTD(X)	Radian to degree conversion.	Page 67
SEC(X)	Secant of X.	Page 66
SGN(X)	The sign of X, -1 if X<0, 0 if X=0, and +1 if X>0.	Page 62
SIN(X)	Sine of X.	Page 66
SQR(X)	Positive square root of X.	Page 62
TAB(N)	Skips to specified column.	Page 168
TAN(X)	Tangent of X.	Page 66
TIME	Time in seconds since midnight (assumes system timer has been set properly).	Page 57
UPC$(S$)	Returns string with all lower-case alphabetic characters converted to upper-case.	Page 133
VAL(S$)	Returns the numeric equivalent of the string S$.	Page 130
VAL$(X)	String equivalent of X.	Page 131

Error Messages

Error Number	Error Condition	Default values (errors 1-8 only) with DEFAULT ON
	Math Errors (1 thru 13)	
1	Underflow: expression underflows machine	0
2	Overflow: • Expression overflows machine • Attempt to store value >99999 or <−99999 in INTEGER variable. • Attempt to store value >**9.9999E99** or <**−9.9999E99** in SHORT variable.	±9.99999999999E499 ±99999 ±9.9999E99
3	COT or CSC of n*180°; n=integer.	9.99999999999E499
4	TAN or SEC of n*90°; n=odd integer.	9.99999999999E499
5	Zero raised to negative power.	9.99999999999E499
6	Zero raised to zero power.	1
7	Null data: • Uninitialized string variable, or missing string function assignment. • Uninitialized numeric variable, or missing numeric function assignment.	 "" 0
8	Division by zero.	±9.99999999999E499
9	Negative value raised to non-integer power.	Remaining errors are non-defaultable.
10	Square root of negative number.	
11	Argument (parameter) out of range: • ATN2(0,0). • ASN or ACSN (−1≮n<+1). • ON expression GOTO/GOSUB; expression of range.	
12	Logarithm of zero.	
13	Logarithm of negative number.	
14	Not used.	
	System Errors (15 thru 25)	
15	System error; correct by reloading program, pressing RESET, or turning system off, then on again.	
16	Continue before run; program not allocated.	
17	FOR nesting too deep; more than 255 active FOR-NEXT loops.	
18	GOSUB nesting too deep; more than 255 nested subroutines.	

Error Number	Error Condition
19	Memory overflow: • Attempting to RUN a program that requires more than given memory. • Attempting to edit too large a program; delete a nonexisting line to deallocate program, then edit. • Attempting to load a program larger than available memory. • Attempting to open a file with no available buffer space. • Attempting any operation that requires more memory than available. • Attempting to load or run a large program after a ROM has been installed. ROMs use up a certain amount of memory. Refer to the appropriate ROM manual.
20	Not used.
21	ROM missing; attempting to RUN program that requires ROM. An attempt to edit program with missing ROM will usually SCRATCH memory.
22	Attempt to edit, list, store, or overwrite a SECUREd program.
23	Self-test error; system needs repair.
24	Too many (more than 14) ROMS.
25	Two binary programs; attempting to load a second binary program into memory (only one binary program allowed in memory at any time).
26 thru 29	Not used.
	Program Errors (30 thru 57)
30	OPTION BASE error: • Duplicate OPTION BASE declaration. • OPTION BASE after array declaration. • OPTION BASE parameter not 0 or 1.
31	CHAIN error; CHAIN to a program other than BASIC main program: e.g., CHAINing to a binary program.
32	COMmon variable mismatch.
33	DATA type mismatch: • READ variable and DATA type do not agree. • READ# found a string but required a number.
34	No DATA to read: • READ and DATA expired. • RESTORE executed with no DATA statement.
35	Dimensioned existing variable; attempt to dimension a variable that has been previously declared or used. Move DIM statement to beginning of program and try again.
36	Illegal dimension: • Illegal dimension in default array declaration. • Array dimensions don't agree; e.g., referencing A(2) when A(5,5) is dimensioned or referencing A(0) when OPTION BASE 1 declared.
37	Duplicate user-defined function.
38	Function definition within function definition; needs FN END.
39	Reference to a nonexistent user-defined function: • Finding FN END with no matching DEF FN. • Exiting a function that was not entered with a function call after branching to the middle of a multi-line function.

Error Number	Error Condition
40	Illegal function parameter; function parameter mismatch (e.g., declared as string, called as numeric).
41	FN=; user-defined function assignment. Function assignment does not occur between DEF FN and FN END.
42	Recursive user-defined function.
43	Numeric input wanted.
44	**Too few inputs. Less items were given than requested by an INPUT statement.**
45	**Too many inputs. More items were given than requested by an INPUT statement.**
46	NEXT missing; FOR with no matching NEXT.
47	FOR missing; NEXT with no matching FOR.
48	END statement necessary.
49	Null data; uninitialized data.
50	Binary program missing; attempting to RUN program that requires binary program. An attempt to edit will usually SCRATCH memory.
51	RETURN without GOSUB reference.
52	Illegal IMAGE format string; unrecognized character in IMAGE.
53	Illegal PRINT USING: • Data overflows IMAGE declaration. • Numeric data with string IMAGE. • String data with numeric IMAGE. • PRINT USING image format string is not correct.
54	Illegal TAB argument. With DEFAULT ON, an illegal TAB argument gives a warning message and defaults to TAB(1).
55	Array subscript out of range.
56	String variable overflow; string too big for variable.
57	Missing line; reference to a nonexistent statement number.
58 thru 59	Not used.
	Tape Errors (60 thru 75)
60	Tape cartridge is write-protected; RECORD slide tab is in left-most position.
61	Attempting to create/record more than 42 files on tape.
62	Cartridge out when attempting tape operations.
63	Duplicate file name for RENAME or CREATE.
64	Empty file; attempting to access file that was never recorded (e.g., tape was ejected *before* program was stored but *after* name was written in directory). Refer to PURGE.

Error Number	Error Condition
65	End of tape: • Tape run-off; check cartridge. • Tape is full. • Not enough space to CREATE data file.
66	File closed: • Attempting READ#/PRINT# to file that has not been opened with ASSIGN#. • Attempting to close a closed file (warning only). • Tape has been ejected and reinserted.
67	File name: • Name does not exist when attempt to LOAD, ASSIGN#, LOAD BIN, PURGE, RENAME, or SECURE. • Name not in quotes. • Attempt to PURGE an open file.
68	File type mismatch: • Attempting to treat program as data file, or vice versa. • Attempting to treat binary program as BASIC main program file, or vice versa. • Attempting to treat data as binary program, or vice versa.
69	Random overflow; attempting to READ#/PRINT# beyond existing number of bytes in logically-defined record with random file access.
70	READ error; system cannot read tape.
71	End-of-File; no data beyond EOF mark in data file.
72	Record: • Attempting to READ#/PRINT# to record that doesn't exist; e.g., READ# 1,120 when only 100 records in file. • Attempting to READ#/PRINT# at end of file. • Lost in record: close file to release buffer.
73	Searches and does not find: • Bad tape cartridge; may have been exposed to magnetic field. • Cannot find directory, tape may need to be initialized.
74	Stall; either bad tape cartridge or transport problem, refer to Tape Operations, appendix B.
75	Not an HP-85 file; cannot read.
76 thru 79	Not used.
	Syntax Errors (80 thru 92)
80	Right parentheses,), expected.
81	Bad BASIC statement or bad expression. If it is an expression, try it again with DISP <expression> to get a better error message.
82	String expression error; e.g., right quote missing or null string given for file name.
83	Comma missing or more parameters expected (separated by commas).
84	Excess characters; delete characters at end of good line, then press END LINE.
85	Expression too big for system to interpret.

Error Number	Error Condition
86	Illegal statement after THEN.
87	Bad DIM statement.
88	Bad statement: • COM in calculator mode. • User-defined function in calculator mode. • INPUT in calculator mode.
89	Invalid parameter: • ON KEY# less than 1 or greater than 8. • Attempt to TRACE a calculator mode variable. • PRINTER IS or CRT IS with invalid parameter. • CREATE with invalid parameters. • ASSIGN#, PRINT#, or READ# with buffer number other than 1 through 10. • Random READ# to record 0. • SETTIME with illegal time parameter. • ON TIMER#, OFF TIMER# with number other than 1, 2, or 3. • SCALE with invalid parameters. • AUTO or REN with invalid parameters. • LIST with invalid parameters. • DELETE with invalid parameters. • VAL$ with non-numeric parameter. • Any statement, command, or function for which parameters are given but they are invalid.
90	Line number too large; greater than 9999.
91	Missing parameter; e.g., DELETE with missing or invalid parameters.
92	Syntax error. Cursor returns to character where error was found.

Sample Solutions to Problems

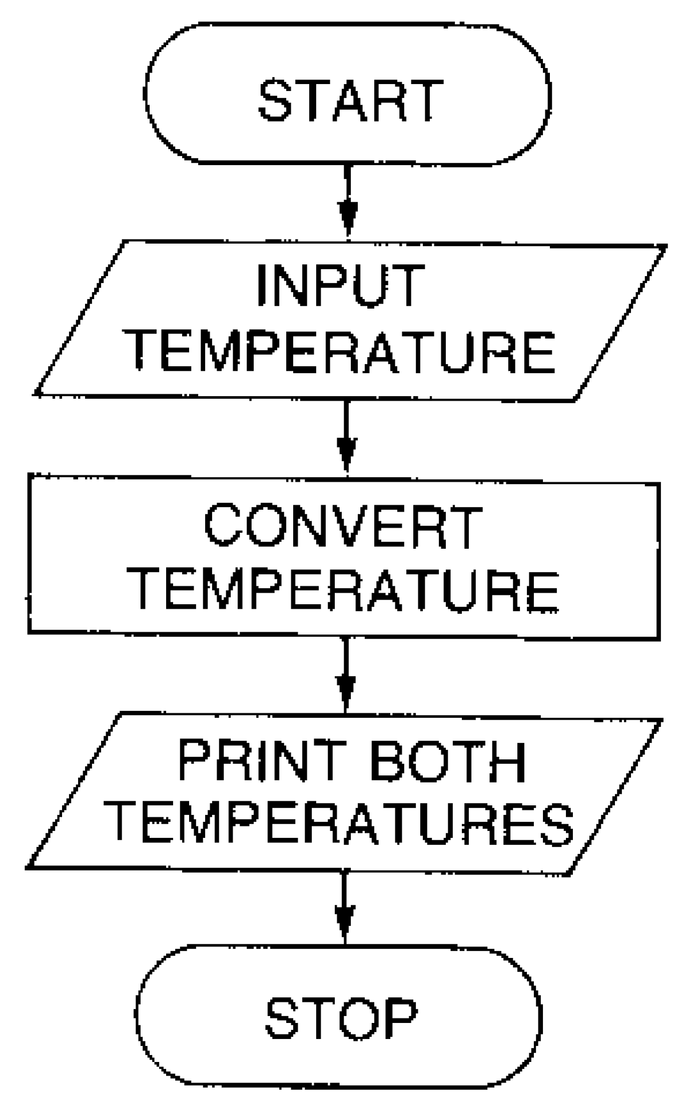
Section 5

Problem 5.1

(a)

```
10 REM *CELSIUS TO FAHRENHEIT
20 DISP "CELSIUS TEMP";
30 INPUT C
40 LET F=1.8*C+32
50 PRINT C;"C  EQUALS  ";F;"F"
60 END
```

Flowchart:



Display:

```
CELSIUS TEMP?
15
CELSIUS TEMP?
-10
```

Printer:

```
 15 C  EQUALS   59 F

-10 C  EQUALS   14 F
```

(b)

```
10 REM *FAHRENHEIT TO CELSIUS
20 DISP "FAHRENHEIT TEMP";
30 INPUT F
40 C=5/9*(F-32)
50 PRINT F;"F  EQUALS  ";C;"C"
60 END
```

Display:

```
FAHRENHEIT TEMP?
59
FAHRENHEIT TEMP?
14
```

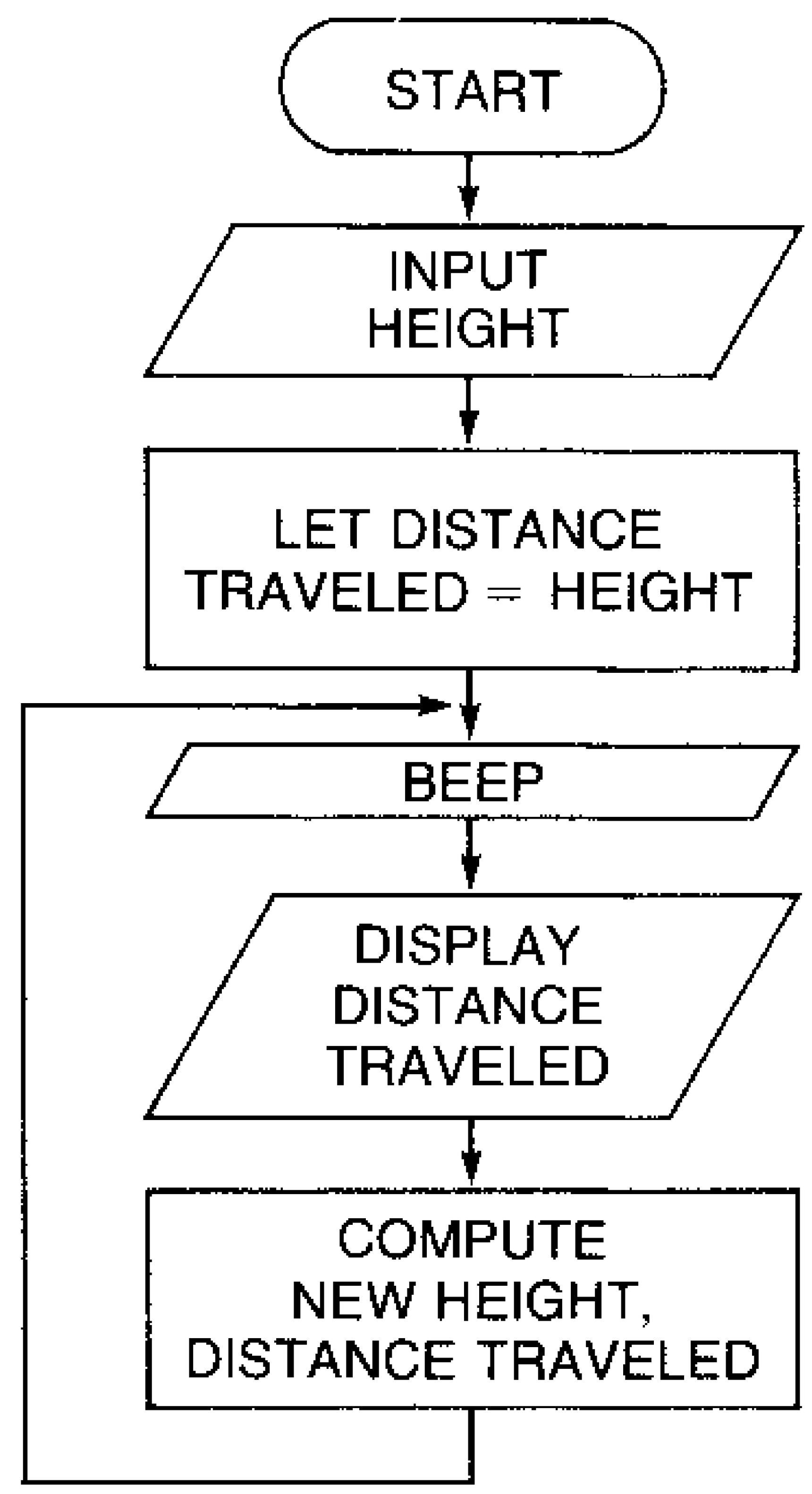
Printer:

```
59 F  EQUALS   15 C

14 F  EQUALS  -10 C
```

Problem 5.2

```
 10 REM *REBOUNDER
 20 DISP "HEIGHT RELEASED"
 30 INPUT H
 40 D=H
 50 BEEP
 60 DISP D
 70 H=.65*H
 80 D=D+2*H
 90 GOTO 50
100 END
```

Flowchart:



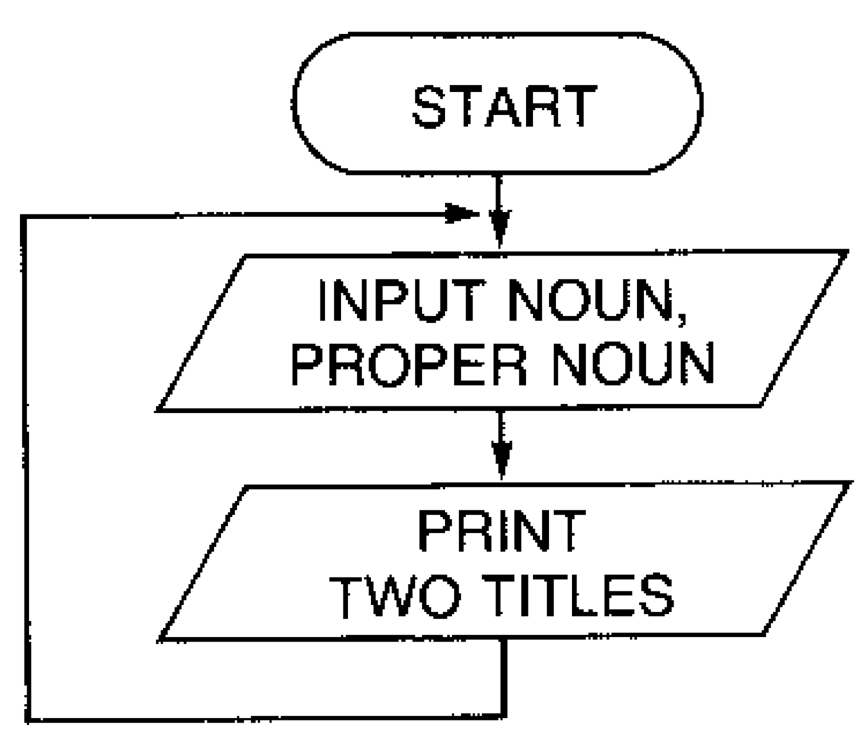
Display:

```
HEIGHT RELEASED
?
100
 100
 230
 314.5
 369.425
 405.12625
 428.3320625
 443.415840625
 453.220296406
 459.593192664
 463.735575232
 466.428123901
 468.178290536
 469.315882349
 470.055323527
```

Problem 5.3

```
10 REM *BOOK TITLES
20 DISP "NOUN, PROPER NOUN";
30 INPUT N$,P$
40 PRINT "THE ";N$;" OF ";P$
50 PRINT "TO ";P$;" WITH THE ";
   N$
60 GOTO 20
70 END
```

Flowchart:



Display:

```
NOUN, PROPER NOUN?
ANIMALS, AUSTRALIA
NOUN, PROPER NOUN?
ICEBERGS,ICELAND
NOUN, PROPER NOUN?
ATHLETES, THE OLYMPICS
NOUN, PROPER NOUN?
```

Printer:

```
THE ANIMALS OF AUSTRALIA
TO AUSTRALIA WITH THE ANIMALS
THE ICEBERGS OF ICELAND
TO ICELAND WITH THE ICEBERGS
THE ATHLETES OF THE OLYMPICS
TO THE OLYMPICS WITH THE
ATHLETES
```

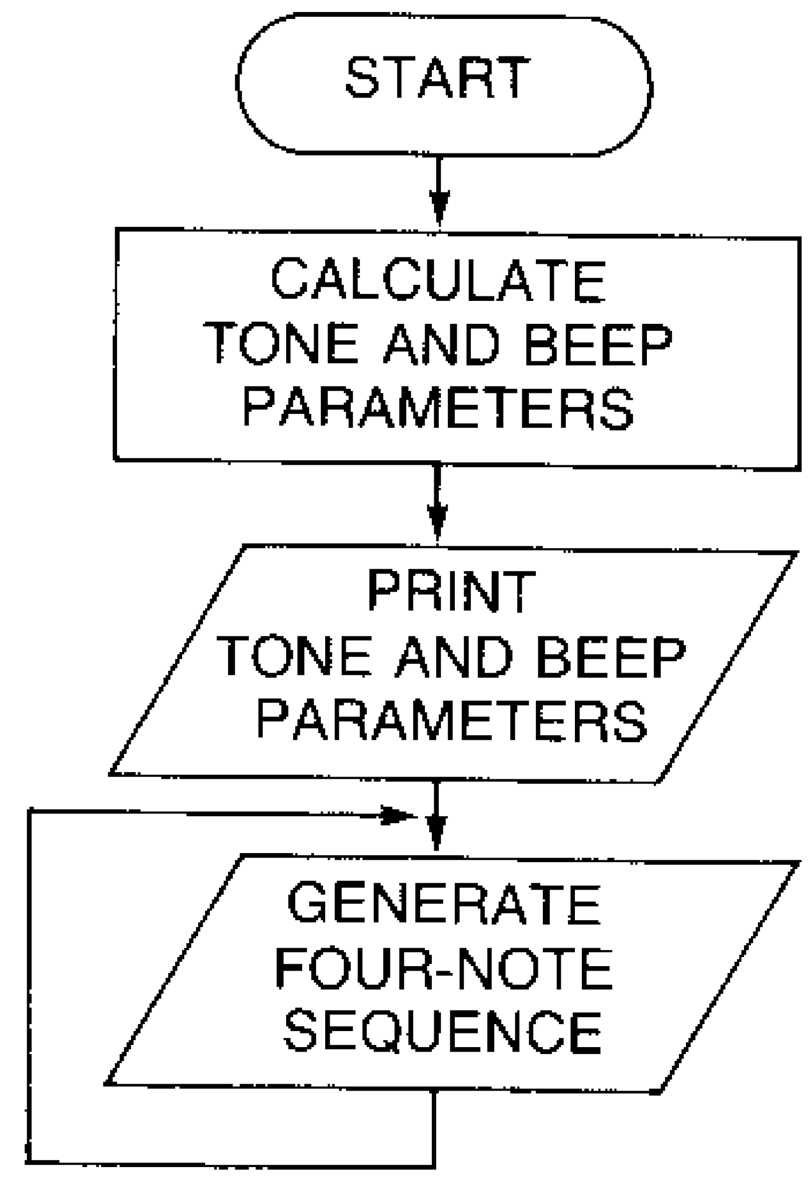
Problem 5.4

```
10 REM *BASS RHYTHM
20 F1=613062.5/(11*98)-134/11
30 D1=.5*98
40 F2=613062.5/(11*130.81)-134/
   11
50 D2=.5*130.81
60 F3=613062.5/(11*196)-134/11
70 D3=.5*196
80 DISP " F"," D",F1,D1,F2,D2,F
   3,D3
90 BEEP F2,D2
100 BEEP F3,D3
110 BEEP F1,D1
120 BEEP F3,D3
130 GOTO 90
140 END
```

Display:

```
F                    D
556.521799629        49
413.878533056        65.405
272.169990723        98
```

Flowchart:



Problem 5.5

```
10 REM *FACTORIAL APPROX
20 DISP "X = ";
30 INPUT X
40 F=EXP(-X)*X^X*SQR(2*PI*X)
50 PRINT F,X;"!"
60 GOTO 20
70 END
```

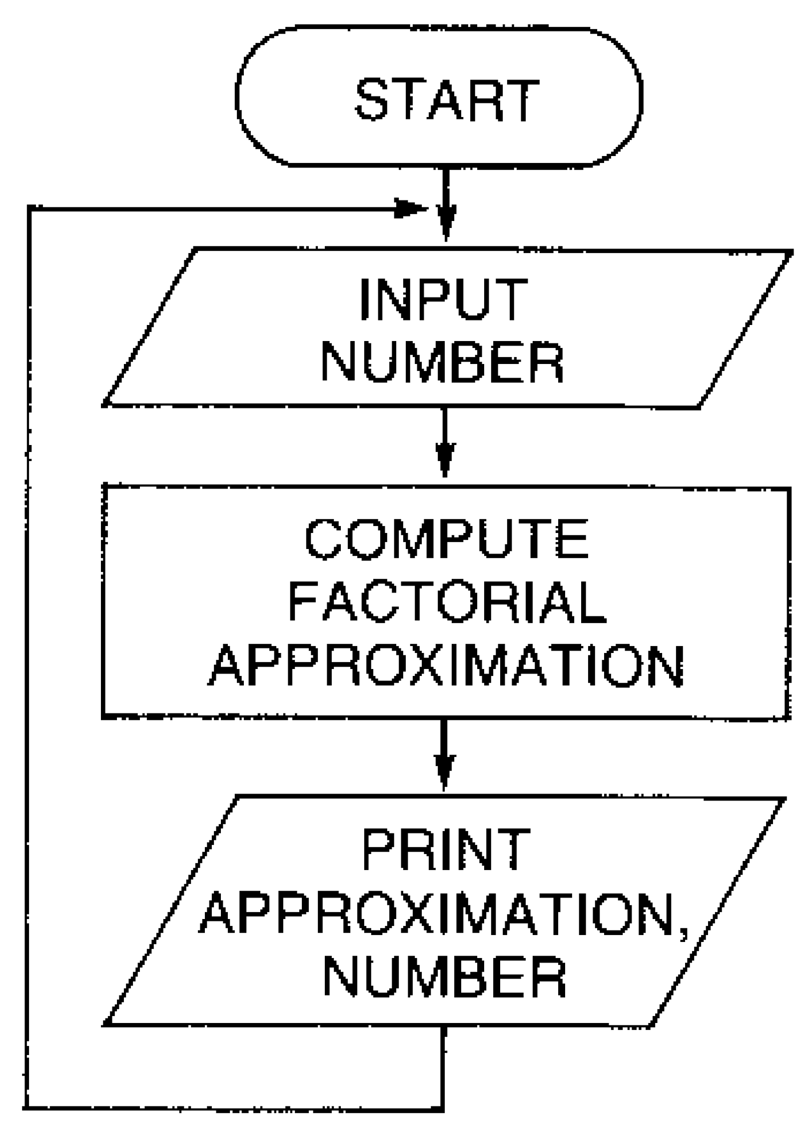
Display:

```
X = ?
3
X = ?
6
X = ?
10
X = ?
50
X = ?
```

Printer:

```
5.83620959134          3 !
710.078184645          6 !
3598695.61874          10 !
3.03634459394E64       50 !
```

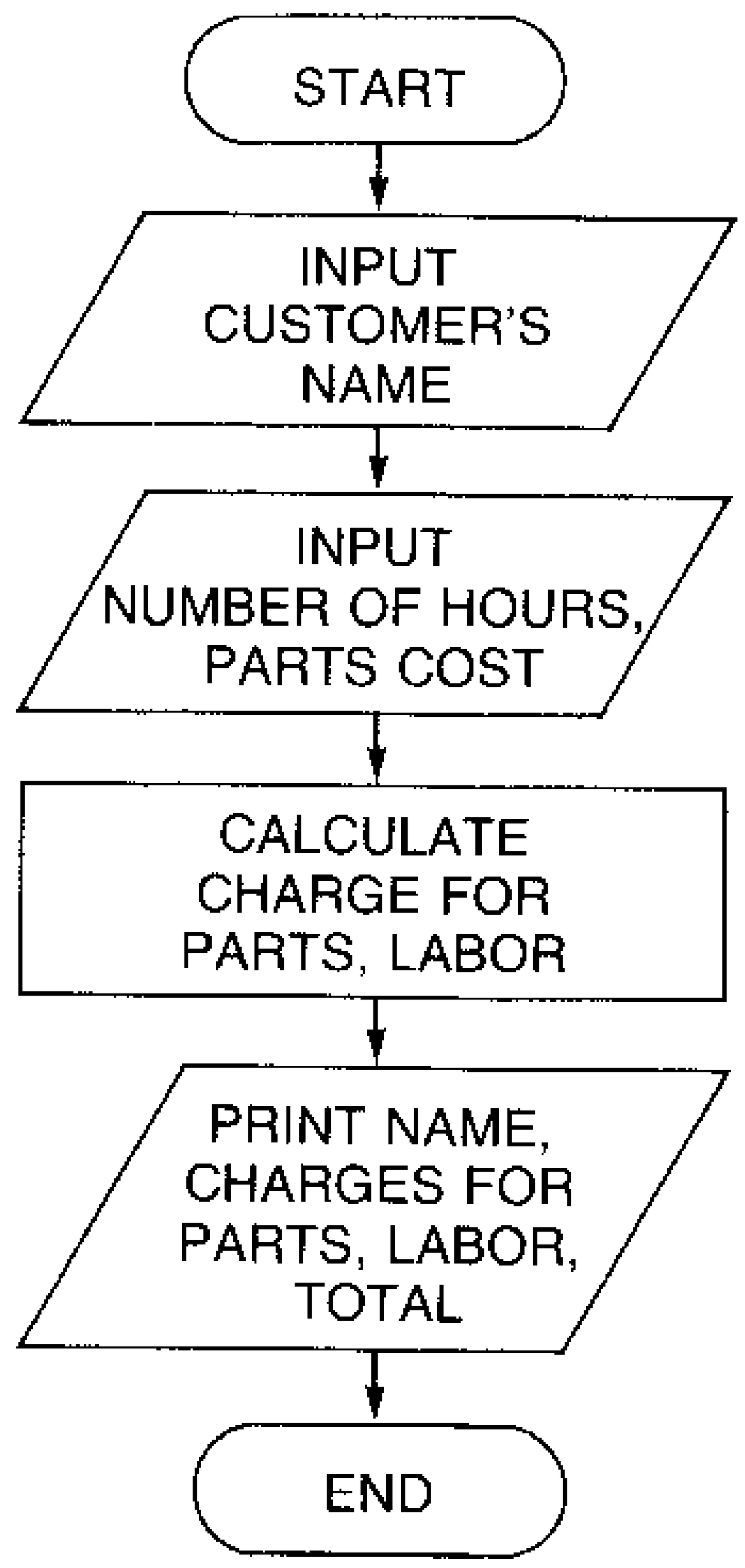
Flowchart:



Problem 5.6

```
 10 REM *WATCH REPAIR
 20 DISP "CUSTOMER";
 30 INPUT N$
 40 DISP "HOURS WORKED, PARTS CO
    ST"
 50 INPUT H,P
 60 L=8.5*H
 70 P1=1.1*P
 80 PRINT
 90 PRINT "ITEMIZED REPAIR BILL:
     ";N$
100 PRINT "  PARTS        $";P1
110 PRINT "  LABOR        $";L
120 PRINT "TOTAL CHARGE   $";P1+L
130 END
```

Flowchart:



Display:

```
CUSTOMER?
WHIMPLE
HOURS WORKED, PARTS COST
?
2.5,12.00
```

Printer:

```
ITEMIZED REPAIR BILL:  WHIMPLE
  PARTS        $ 13.2
  LABOR        $ 21.25
TOTAL CHARGE   $ 34.45
```

Section 6

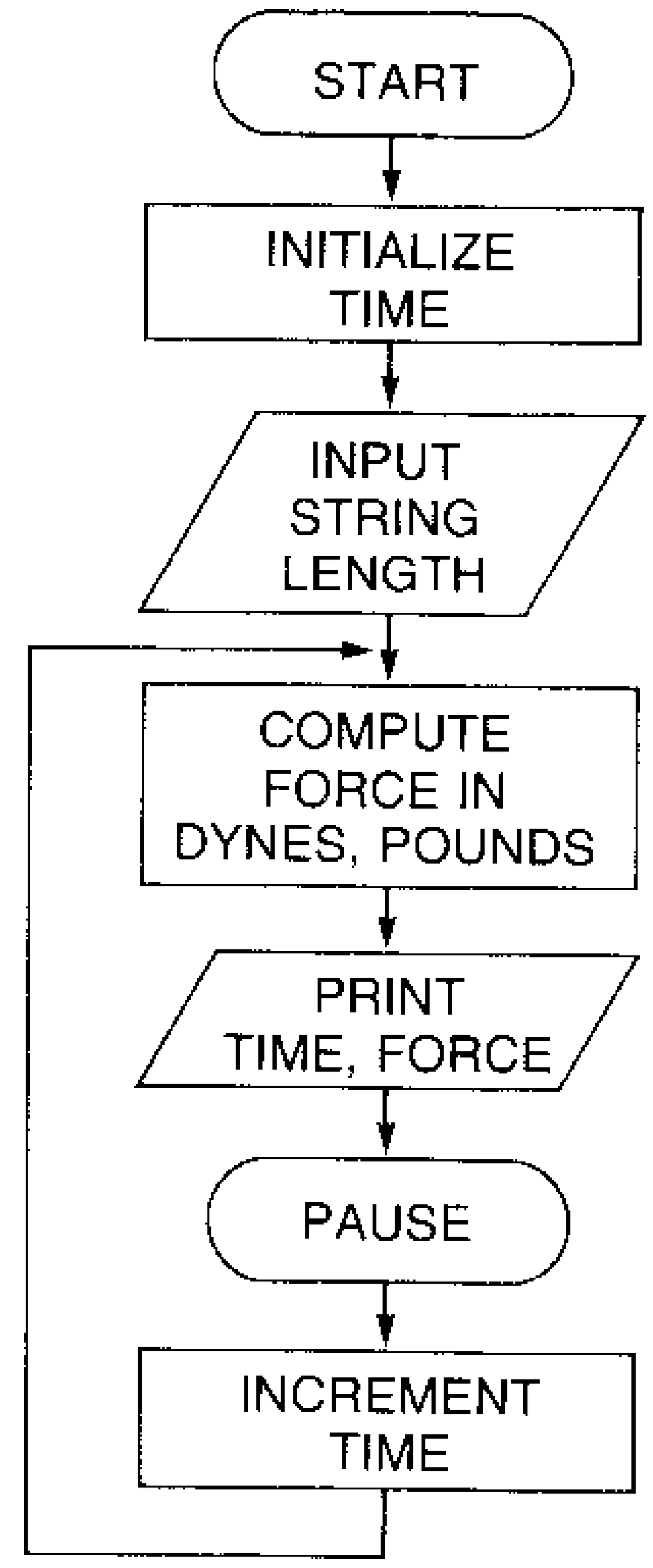
Problem 6.1

```
 10 REM *REBOUNDER
 20 DISP "HEIGHT RELEASED"
 30 INPUT H
 35 T=3000      <---------- Added line.
 40 D=H
 50 BEEP
 60 DISP D
 65 WAIT T      <---------- Added line.
 70 H=.65*H
 80 D=D+2*H
 85 T=.806*T    <---------- Added line.
 90 GOTO 50
100 END
```

Problem 6.2

```
 10 REM *CENTRIFUGAL
 20 T=0
 30 DISP "STRING LENGTH";
 40 INPUT R
 50 F=350*(30*T)^2/R
 60 PRINT "SECONDS =";T
 70 PRINT "DYNES =";F
 80 PRINT "POUNDS =";.00000225*F
 90 PRINT
100 PAUSE
110 T=T+1
120 GOTO 50
130 END
```

Flowchart:



Display:

```
STRING LENGTH?
14
```

Printer:

```
SECONDS = 0
DYNES = 0
POUNDS = 0

SECONDS = 1
DYNES = 22500
POUNDS = .050625

SECONDS = 2
DYNES = 90000
POUNDS = .2025

SECONDS = 3
DYNES = 202500
POUNDS = .455625

SECONDS = 4
DYNES = 360000
POUNDS = .81

SECONDS = 5
DYNES = 562500
POUNDS = 1.265625

SECONDS = 6
DYNES = 810000
POUNDS = 1.8225

SECONDS = 7
DYNES = 1102500
POUNDS = 2.480625

SECONDS = 8
DYNES = 1440000
POUNDS = 3.24

SECONDS = 9
DYNES = 1822500
POUNDS = 4.100625

SECONDS = 10
DYNES = 2250000
POUNDS = 5.0625
```

Section 7

Problem 7.1

```
 10 REM *BASKETBALL
 20 A,W=0
 30 PRINT " A   W"
 40 PRINT
 50 INPUT C$
 60 IF C$="A" THEN A=A+2
 70 IF C$="W" THEN W=W+2
 80 IF C$="a" THEN A=A+1
 90 IF C$="w" THEN W=W+1
100 PRINT A;W
110 GOTO 50
120 END
```

Problem 7.1 (Cont)

Display:

```
?
A
?
A
?
W
?
w
?
A
?
A
?
a
?
W
?
w
?
```

Printer:

```
A  W

2  0
4  0
4  2
4  3
6  3
8  3
9  3
9  5
9  6
```

Problem 7.2

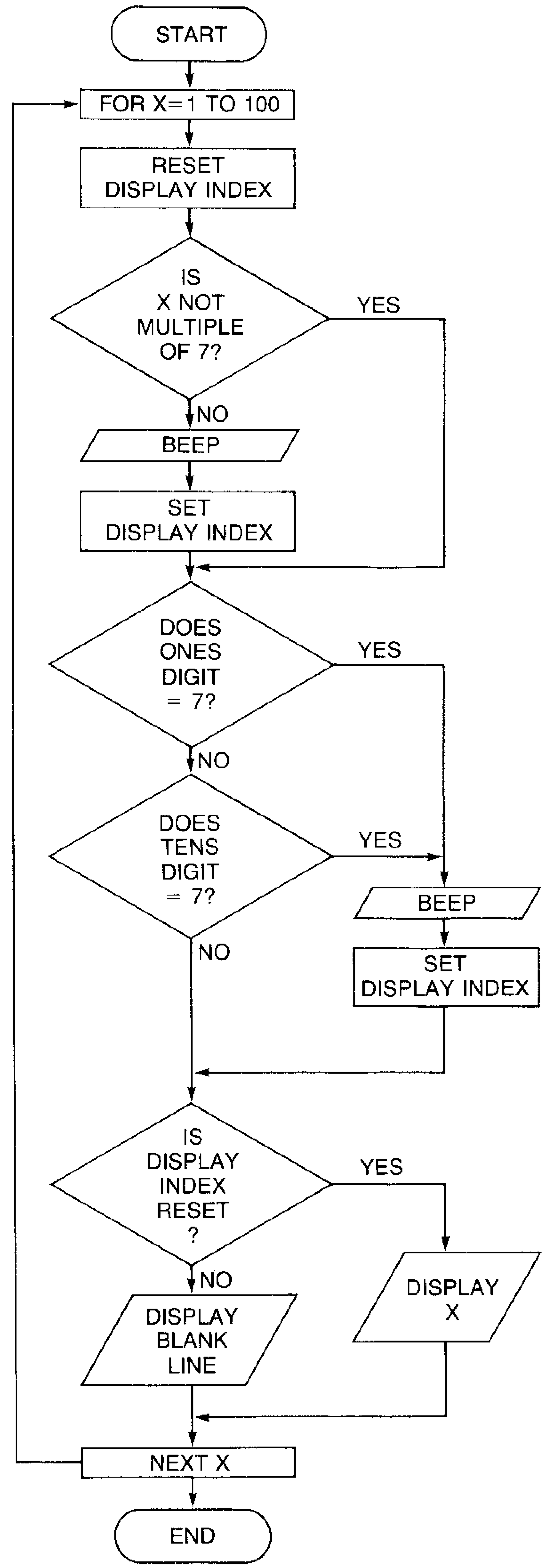
```
 10 REM *BEEP GAME
 20 FOR X=1 TO 100
 30 S=0
 40 IF FP(X/7)#0 THEN 70
 50 BEEP
 60 S=1
 70 IF 10*FP(X/10)=7 THEN 100
 80 IF IP(X/10)=7 THEN 100
 90 GOTO 120
100 BEEP
110 S=1
120 IF S=0 THEN DISP X ELSE DISP
130 NEXT X
140 END
```

Display:

```
1
2
3
4
5
6

8
9
10
11
12
13

15
16

18
19
.
.
.
```

Problem 7.2

Flowchart:



Problem 7.3

```
 10 REM *TELEPATHY
 20 R,W=0
 30 DISP "ENTER 1 TO 5 EACH TIME
    "
 40 FOR I=1 TO 10
 50 P=INT(1+5*RND)
 60 WAIT 5000
 70 INPUT N
 80 IF N=P THEN 170
 90 W=W+1
100 DISP "INCORRECT"
110 NEXT I
120 A=100*R/(R+W)
130 PRINT A;"% ACCURACY"
140 PRINT " FOR";R+W;"PICKS"
150 IF A>20 THEN PRINT "**TELEPA
    THY**" ELSE PRINT "JUST GUES
    SING"
160 GOTO 40
170 R=R+1
180 DISP "CORRECT"
190 GOTO 110
200 END
```

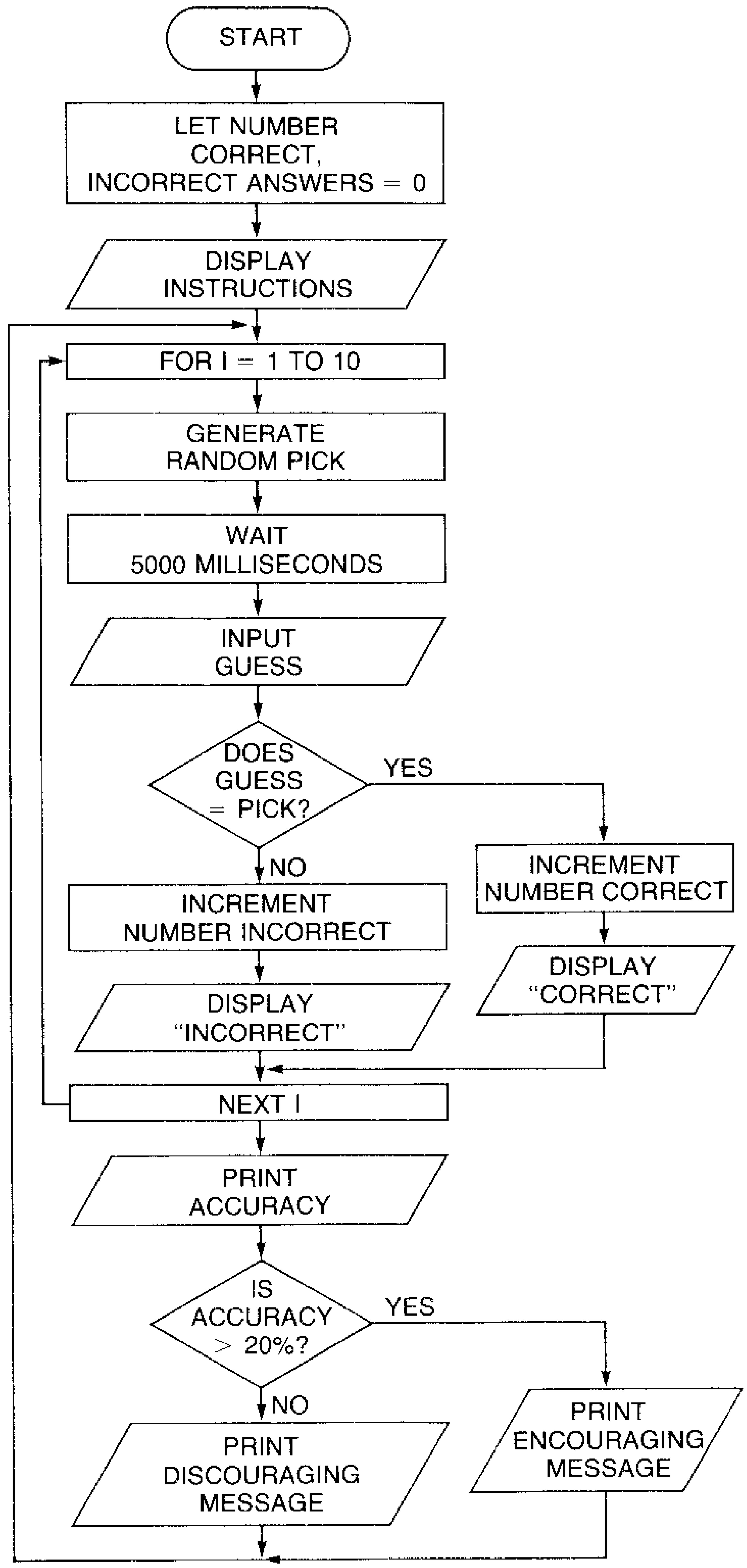
Display:

```
ENTER 1 TO 5 EACH TIME
?
2
INCORRECT
?
1
CORRECT
?
4
INCORRECT
?
5
INCORRECT
?
2
CORRECT
?
3
INCORRECT
?
5
INCORRECT
?
1
INCORRECT
?
2
INCORRECT
?
3
CORRECT
?
```

Printer:

```
 30 % ACCURACY
 FOR 10 PICKS
**TELEPATHY**
```

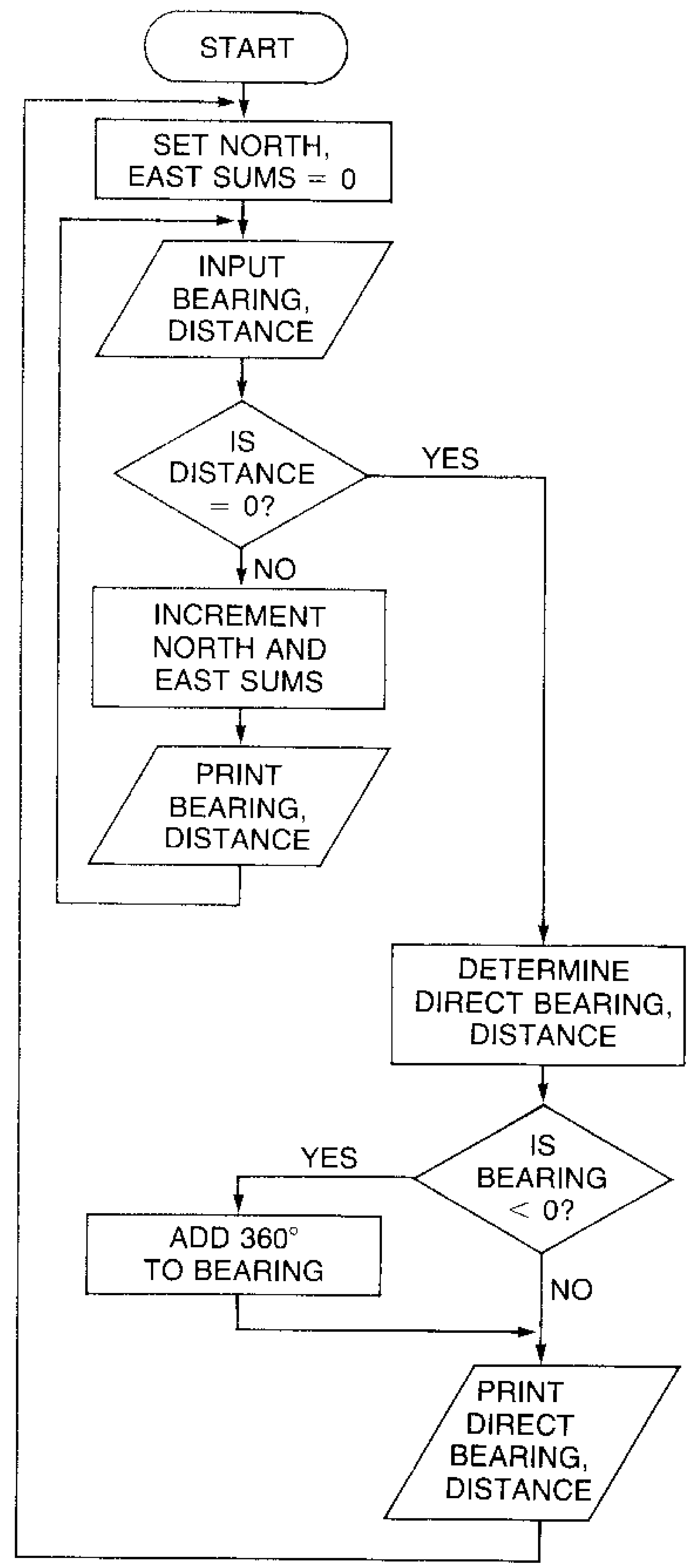
Flowchart:



Problem 7.4

```
 10 REM *COMPASS COURSE
 20 DEG
 30 N,E=0
 40 DISP "BEARING, DISTANCE";
 50 INPUT B,D
 60 IF D=0 THEN 110
 70 N=N+D*COS(B)
 80 E=E+D*SIN(B)
 90 PRINT "BEAR";B;" DIST";D
100 GOTO 40
110 A=ATN2(E,N)
120 X=SQR(N*N+E*E)
130 IF A<0 THEN A=A MOD 360
140 PRINT "DIRECT ROUTE"
150 PRINT " BEARING ";A
160 PRINT " DISTANCE";X
170 PRINT
180 GOTO 30
190 END
```

Flowchart:



Display:

```
BEARING, DISTANCE?
85,350
BEARING, DISTANCE?
190,400
BEARING, DISTANCE?
265,250
BEARING, DISTANCE?
20,50
BEARING, DISTANCE?
0,0
BEARING, DISTANCE?
```

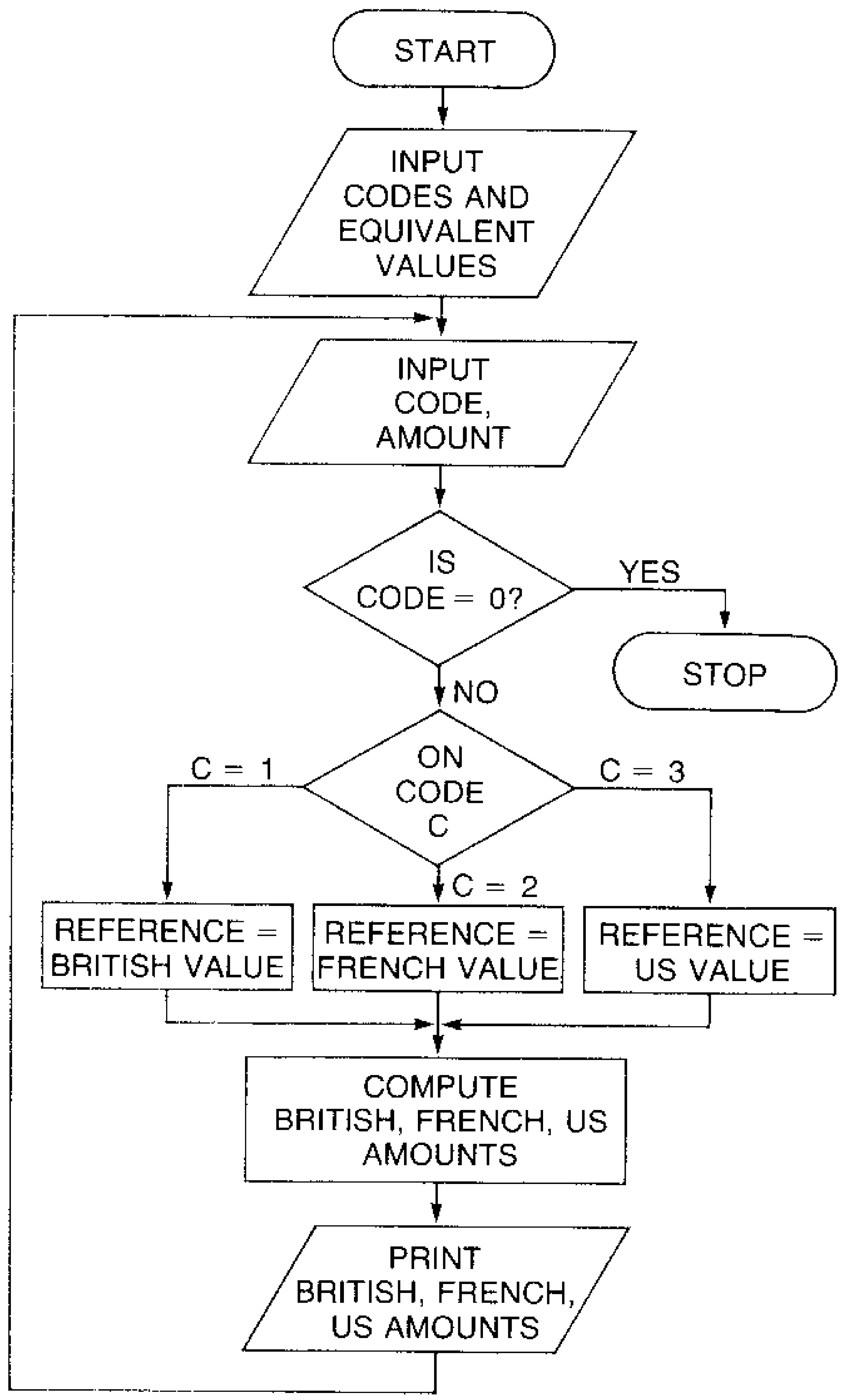
Printer:

```
BEAR 85  DIST 350
BEAR 190  DIST 400
BEAR 265  DIST 250
BEAR 20  DIST 50
DIRECT ROUTE
 BEARING  172.045343208
 DISTANCE 341.508929443
```

Problem 7.5

```
10 REM *CURRENCY EXCHANGE
20 DISP "ENTER CODE, AMT FOR 3
   SYSTS"
30 DISP "1=BR, 2=FR,3=US"
40 INPUT C1,A1,C2,A2,C3,A3
50 DISP
60 DISP "CODE, AMT TO CONVERT (
   0,0=STOP)"
70 INPUT C,A
80 IF C=0 THEN STOP
90 REM *DETERMINE REFERENCE
100 ON C GOTO 110,130,150
110 R=A1
120 GOTO 160
130 R=A2
140 GOTO 160
150 R=A3
160 V1=A*A1/R
170 V2=A*A2/R
180 V3=A*A3/R
190 PRINT "EQUIVALENT AMOUNTS:"
200 PRINT "BR. POUND ";V1
210 PRINT "FR. FRANC ";V2
220 PRINT "US DOLLAR ";V3
230 PRINT
240 GOTO 50
250 END
```

Flowchart:



Display:

```
ENTER CODE, AMT FOR 3 SYSTS
1=BR, 2=FR,3=US
?
1,1,2,8.3981,3,1.8248

CODE, AMT TO CONVERT (0,0=STOP)
?
1,284

CODE, AMT TO CONVERT (0,0=STOP)
?
3,1205

CODE, AMT TO CONVERT (0,0=STOP)
?
0,0
```

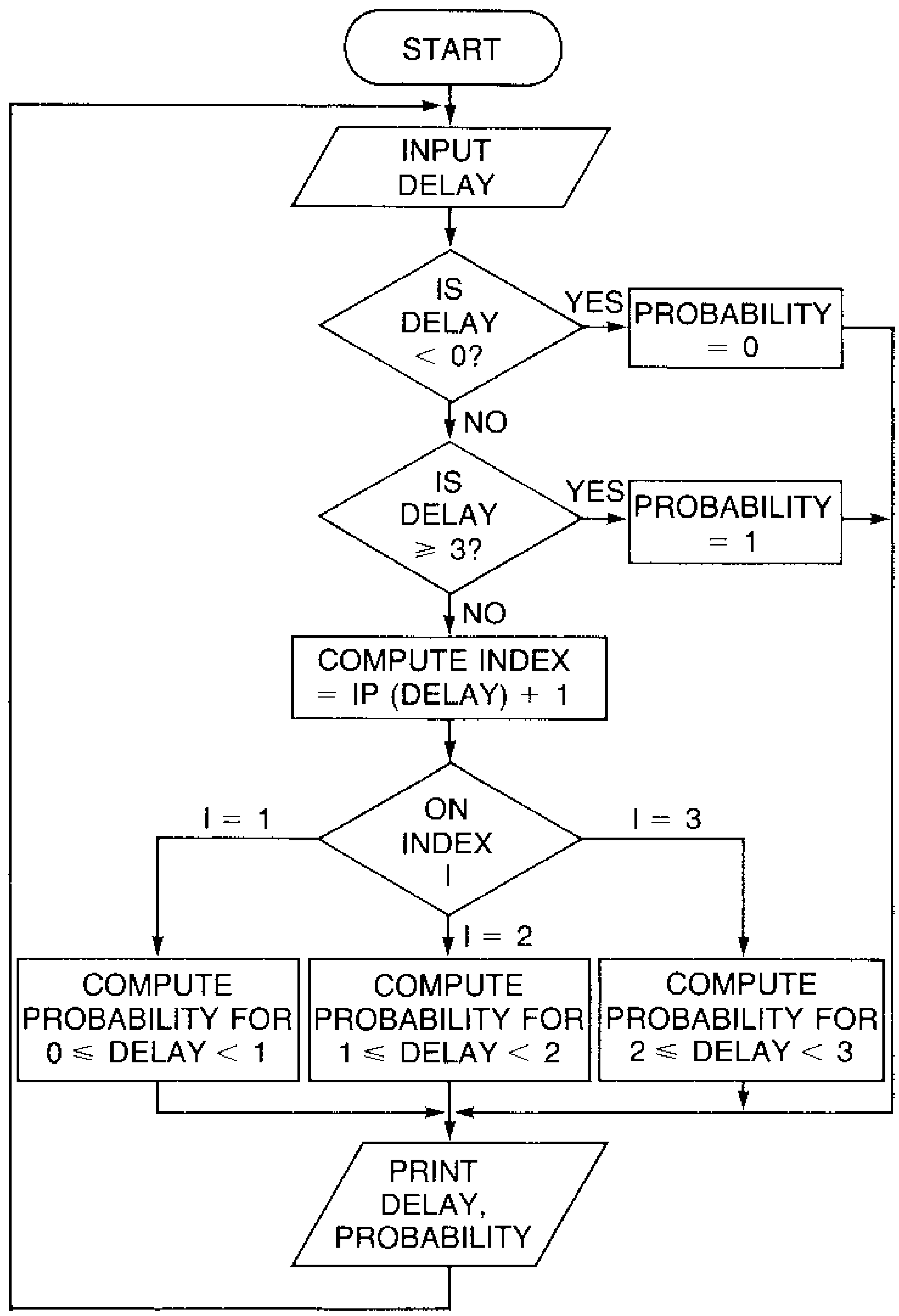
Printer:

```
EQUIVALENT AMOUNTS:
BR. POUND  284
FR. FRANC  2385.0604
US DOLLAR  518.2432

EQUIVALENT AMOUNTS:
BR. POUND  660.346239325
FR. FRANC  5545.65459228
US DOLLAR  1205
```

Problem 7.6

```
 10 REM *DIMSBURG DELAY
 20 DISP "TOTAL DELAY IN MINUTES
    ";
 30 INPUT T
 40 IF T<0 THEN 80
 50 IF T>=3 THEN 160
 60 I=IP(T)+1
 70 ON I GOTO 100,120,140
 80 P=0
 90 GOTO 170
100 P=T^3/6
110 GOTO 170
120 P=.5-T*(1.5-T*(1.5-T/3))
130 GOTO 170
140 P=-3.5+T*(4.5-T*(1.5-T/6))
150 GOTO 170
160 P=1
170 PRINT "FOR DELAY LESS THAN";
    T
180 PRINT "  PROBABILITY IS";P
190 PRINT
200 GOTO 20
210 END
```

Flowchart:



Display:

```
TOTAL DELAY IN MINUTES?
.5
TOTAL DELAY IN MINUTES?
1.5
TOTAL DELAY IN MINUTES?
2
TOTAL DELAY IN MINUTES?
```

Printer:

```
FOR DELAY LESS THAN .5
  PROBABILITY IS
 2.08333333333E-2

FOR DELAY LESS THAN 1.5
  PROBABILITY IS .5

FOR DELAY LESS THAN 2
  PROBABILITY IS .83333333332
```

Section 8

Problem 8.1

```
 10 REM *INVENTAWORD
 20 DIM B$[20],F$[20],W$[21]
 30 DISP "BASE STRING";
 40 INPUT B$
 50 IF LEN(B$)=0 THEN STOP
 60 D$=B$[1,1]
 70 DISP "FIRST-LETTER STRING";
 80 INPUT F$
 90 IF LEN(F$)=0 THEN 30
100 FOR I=1 TO LEN(F$)
110 IF F$[I,I]=D$ THEN 140
120 W$=F$[I,I]&B$
130 PRINT W$
140 NEXT I
150 GOTO 70
160 END
```

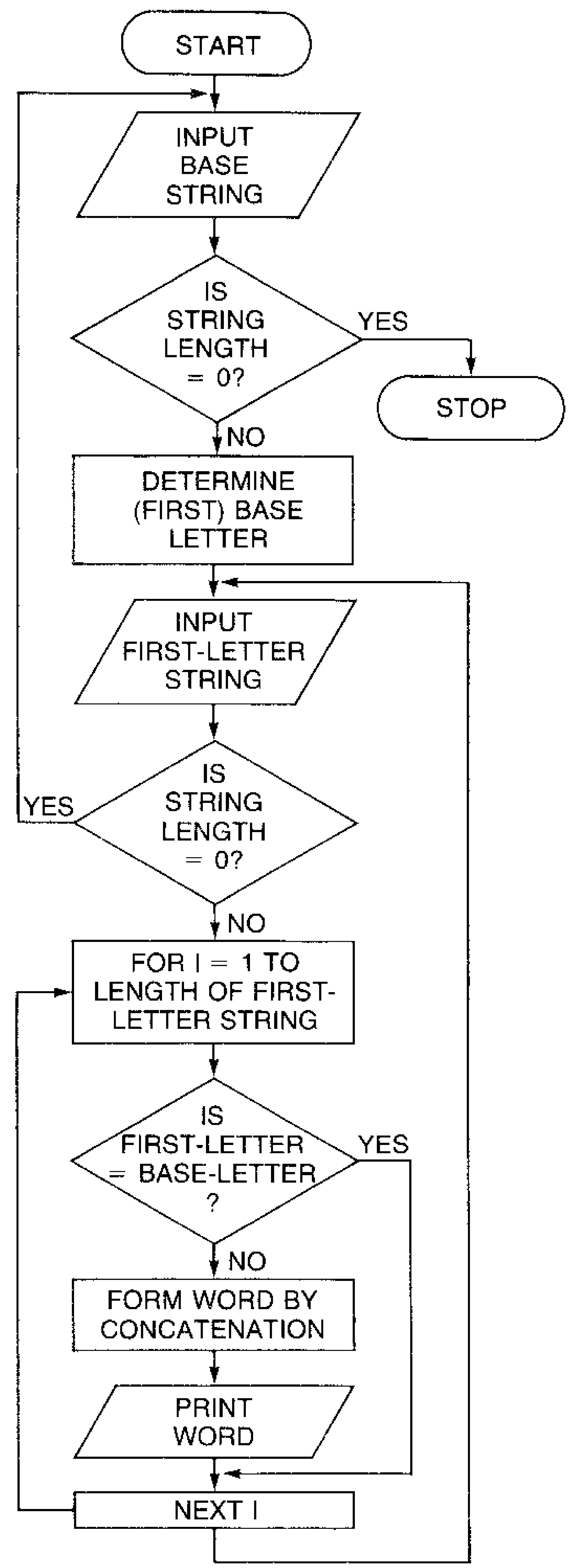
Display:

```
BASE STRING?
LUB
FIRST-LETTER STRING?
GBLF
FIRST-LETTER STRING?
OZ
FIRST-LETTER STRING?
```

Printer:

```
GLUB
BLUB
FLUB
OLUB
ZLUB
```

Flowchart:



Problem 8.2

```
 10 REM *STAR DISTANCES
 20 OPTION BASE 1
 30 INTEGER N(15)
 40 SHORT D
 50 DATA 4.3,5.9,7.6,8.1,8.6,8.9
    ,9.4,10.3,10.7,10.8,10.8,11.
    2,11.2,11.4,11.5,11.6
 60 DATA 11.7,11.9,12.2,12.5,12.
    7,12.8,13.1,13.1,13.9,14.2,1
    4.5
 70 FOR I=1 TO 15
 80 N(I)=0
 90 NEXT I
100 FOR I=1 TO 27
110 READ D
120 J=IP(D)
130 N(J)=N(J)+1
140 NEXT I
150 PRINT "INTERVAL   STARS"
160 FOR I=1 TO 15
170 PRINT I-1;"-";I;"     ";N(I)
180 NEXT I
190 PRINT
200 DISP "INTERVAL";
210 INPUT K
220 IF K=0 THEN STOP
230 PRINT "INTERVAL";K-1;"-";K
240 IF N(K)=0 THEN 370
250 RESTORE
260 IF K=1 THEN GOTO 320
270 FOR I=1 TO K-1
280 FOR J=1 TO N(I)
290 READ D
300 NEXT J
310 NEXT I
320 FOR I=1 TO N(K)
330 READ D
340 PRINT D
350 NEXT I
360 GOTO 190
370 PRINT "  NO STARS"
380 GOTO 190
390 END
```

Display:

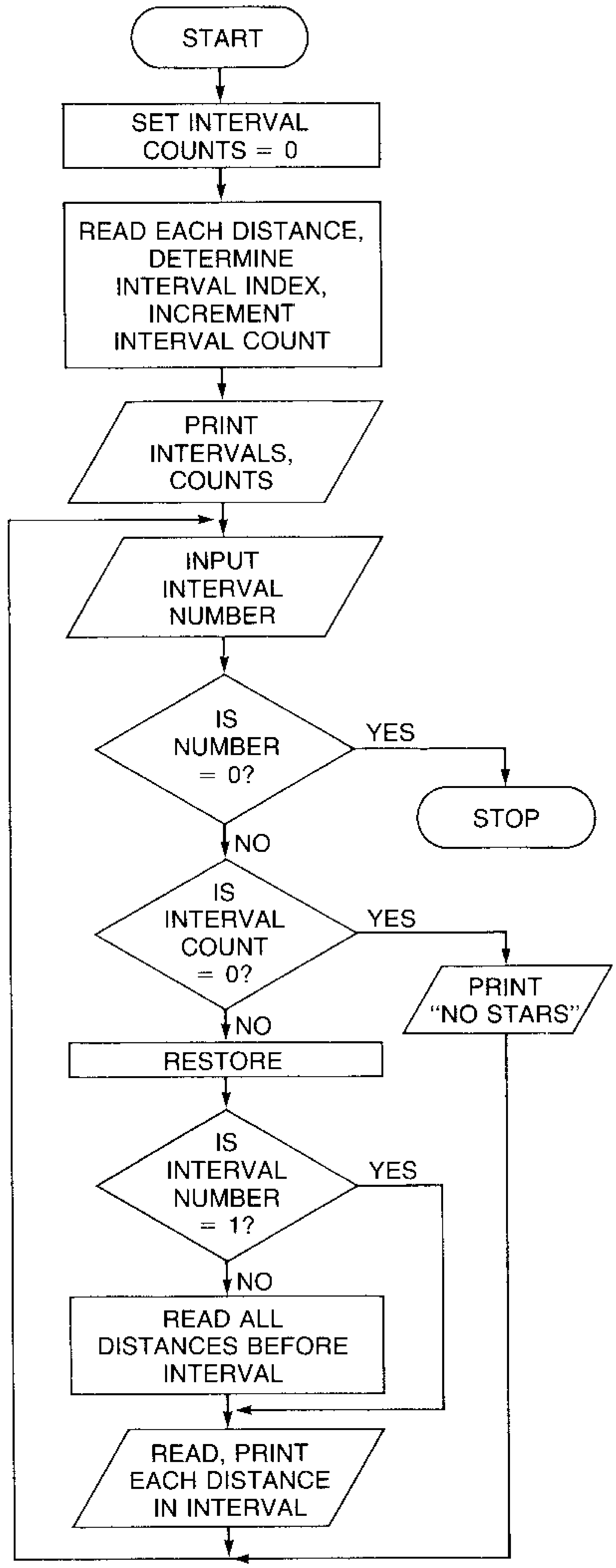
```
INTERVAL?
10
INTERVAL?
0
```

Printer:

```
INTERVAL    STARS
 0 - 1        0
 1 - 2        0
 2 - 3        0
 3 - 4        1
 4 - 5        1
 5 - 6        0
 6 - 7        1
 7 - 8        3
 8 - 9        1
 9 - 10        4
 10 - 11        7
 11 - 12        4
 12 - 13        3
 13 - 14        2
 14 - 15        0
```

```
INTERVAL 9 - 10
 10.3
 10.7
 10.8
 10.8
```

Flowchart:



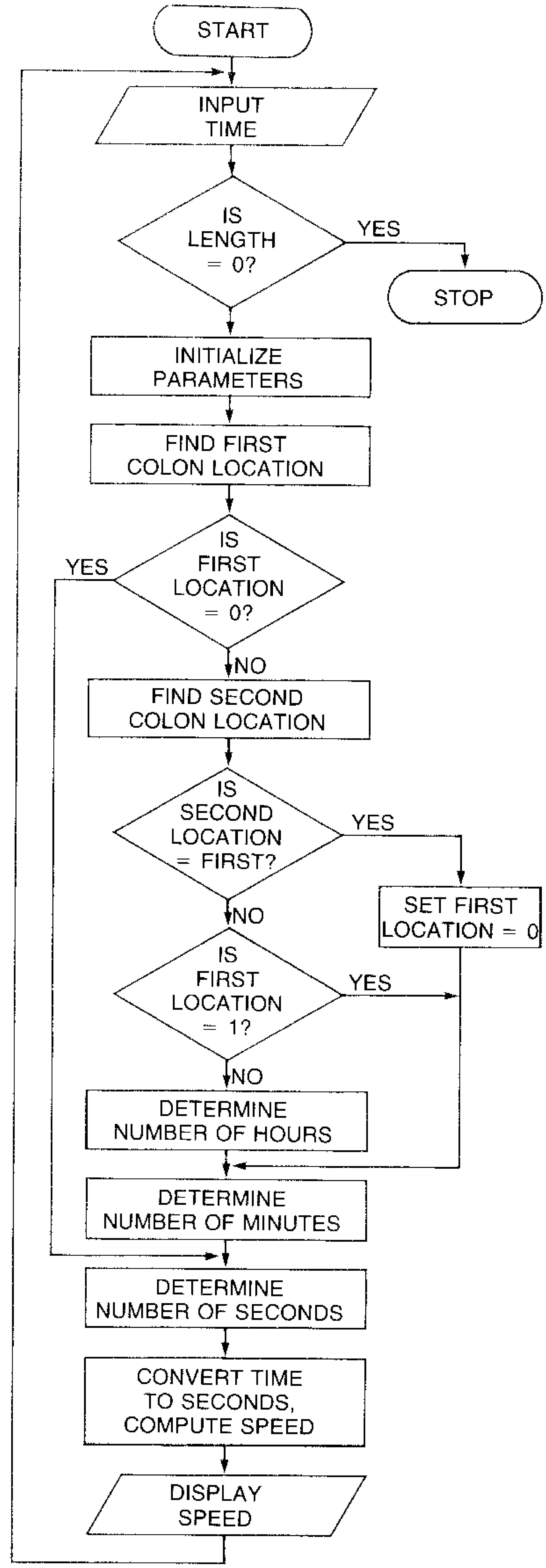
Problem 8.3

```
 10 REM *30-KM SPEEDS
 20 DISP "30--KM TIME";
 30 INPUT T$
 40 IF LEN(T$)=0 THEN STOP
 50 C1,C2=0
 60 H,M,S=0
 70 C1=POS(T$,":")
 80 IF C1=0 THEN 140
 90 C2=POS(T$[C1+1],":")+C1
100 IF C2=C1 THEN 190
110 IF C1=1 THEN 130
120 H=VAL(T$[1,C1-1])
130 M=VAL(T$[C1+1,C2-1])
140 S=VAL(T$[C2+1])
150 V=30000/(S+60*(M+60*H))
160 DISP "SPEED (m/s) =";V
170 DISP
180 GOTO 20
190 C1=0
200 GOTO 130
210 END
```

Display:

```
30-KM TIME?
1:31:30.4
SPEED (m/s) = 5.46408276264

30-KM TIME?
2:12:58.0
SPEED (m/s) = 3.76034093758

30-KM TIME?
1:30:29.38
SPEED (m/s) = 5.52549278186

30-KM TIME?
26:44
SPEED (m/s) = 18.7032418953

30-KM TIME?
```

Flowchart:



Problem 8.4

```
10 REM *COUNTING
20 DIM W$[60]
30 W$=" ZERO  ONE   TWO   THREE
   FOUR  FIVE  SIX   SEVEN EIG
  HT NINE "
40 FOR I=0 TO 9
50 FOR J=0 TO 9
60 K=1+6*I
70 IF I=0 THEN N$="" ELSE N$=W$
  [K,K+5]
80 K=1+6*J
90 N$=N$&W$[K,K+5]
100 DISP N$
110 NEXT J
120 NEXT I
130 END
```

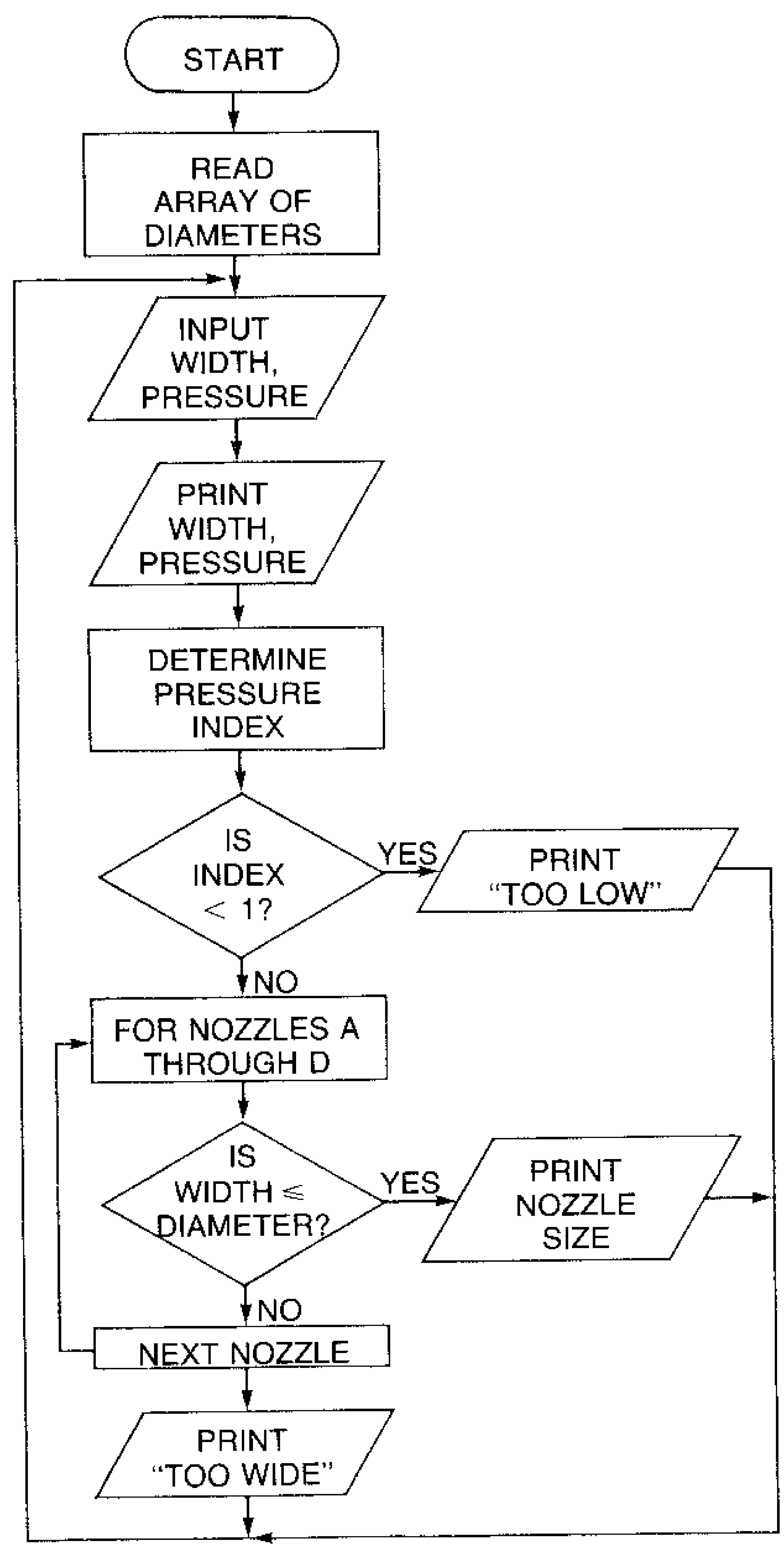
Display:

```
ZERO
ONE
TWO
THREE
FOUR
FIVE
SIX
SEVEN
EIGHT
NINE
ONE   ZERO
ONE   ONE
ONE   TWO
ONE   THREE
ONE   FOUR
ONE   FIVE
```

Problem 8.5

```
10 REM *SPRINKLER
20 OPTION BASE 1
30 DIM D(5,4)
40 FOR I=1 TO 5
50 READ D(I,1),D(I,2),D(I,3),D(
  I,4)
60 DATA 124,133,138,142,126,136
  ,141,146,129,139,144,149
70 DATA 132,142,147,152,134,145
  ,150,155
80 NEXT I
90 N$="ABCD"
100 PRINT " FT   PSI  NOZZLE"
110 DISP "WIDTH, PRESSURE";
120 INPUT W,P
130 PRINT
140 PRINT W;P;
150 J= MIN (5,IP(P/5-11))
160 IF J<1 THEN PRINT " TOO LOW"
   @ GOTO 110
170 FOR I=1 TO 4
180 IF W<=D(J,I) THEN 220
190 NEXT I
200 PRINT " TOO WIDE"
210 GOTO 110
220 PRINT " NOZZLE ";N$[I,I]
230 GOTO 110
240 END
```

Flowchart:



Display:

```
WIDTH, PRESSURE?
150,75
WIDTH, PRESSURE?
140,75
WIDTH, PRESSURE?
140,60
WIDTH, PRESSURE?
```

Printer:

```
FT   PSI  NOZZLE

150  75   NOZZLE D

140  75   NOZZLE B

140  60   NOZZLE D
```

Section 9

Problem 9.1

(a)

```
10 REM *ROUND AT RADIX*
20 DEF FNR(D) = INT(D+.5)
30 FOR I=-5 TO 5 STEP .3
40 DISP I,FNR(I)
50 NEXT I
60 END
```

Display:

-5	-5
-4.7	-5
-4.4	-4
-4.1	-4
-3.8	-4
-3.5	-3
-3.2	-3
-2.9	-3
-2.6	-3
-2.3	-2
-2	-2
-1.7	-2
-1.4	-1
-1.1	-1
-.8	-1
-.5	0
-.2	0
.1	0
.4	0
.7	1
1	1
1.3	1
1.6	2
1.9	2
2.2	2
2.5	3
2.8	3
3.1	3
3.4	3
3.7	4
4	4
4.3	4
4.6	5
4.9	5

(b)

```
10 REM *ROUND TO 3 DECIMAL PLAC
   ES
20 DEF FNR3(D) = INT(D*1000+.5)
   /1000
30 FOR I=1 TO 10 STEP .5
40 DISP I,FNR3(SQR(I))
50 NEXT I
60 END
```

Display:

1	1
1.5	1.225
2	1.414
2.5	1.581
3	1.732
3.5	1.871
4	2
4.5	2.121
5	2.236
5.5	2.345
6	2.449
6.5	2.55
7	2.646
7.5	2.739
8	2.828
8.5	2.915
9	3
9.5	3.082
10	3.162

Problem 9.2

(a)

```
10 REM *AREA
20 DEF FNC(R) = PI*R*R
30 FOR I=350 TO 360
40 DISP I,FNC(I)
50 NEXT I
60 END
```

Display:

```
350
384845.100066
351
387047.356515
352
389255.896149
353
391470.718972
354
393691.824977
355
395919.214167
356
398152.886546
357
400392.842107
358
402639.080856
359
404891.602729
360
407150.407904
```

(b)

```
10 REM *ROUND AREA
20 DEF FNC(R) = PI*R*R
30 FOR I=350 TO 360
40 DISP I,FNR(FNC(I))
50 NEXT I
60 DEF FNR(N) = INT(N*100+.5)/1
   00
70 END
```

Display:

350	384845.1
351	387047.36
352	389255.9
353	391470.72
354	393691.82
355	395919.21
356	398152.89
357	400392.84
358	402639.08
359	404891.6
360	407150.41

Problem 9.3

```
10 REM *FIND HYPOTENUSE
20 INPUT X
30 DEF FNA(X) = SQR(X*X+Y*Y)
40 FOR I=1 TO 5
50 INPUT Y
60 DISP FNA(X)
70 NEXT I
80 END
```

Problem 9.3 (Cont)

Display:

```
?
5
?
4
 6.40312423743
?
3
 5.83095189485
?
6
 7.81024967591
?
7
 8.60232526704
?
9
 10.295630141
```

Problem 9.4

(a)

```
 10 REM *OCTAL TO DECIMAL FN
 20 DEF FND(O)
 30 D=100000000000
 40 S=0
 50 X=IP(O/D)
 60 S=S*8+X
 70 O=O-X*D
 80 D=D/10
 90 IF D>=1 THEN 50
100 FND=S
110 FN END
```

(b)

```
 10 REM *OCTAL TO DECIMAL FN
 20 DEF FND(O)
 30 D=100000000000
 40 S=0
 50 X=IP(O/D)
 60 IF X>=8 OR X<0 THEN DISP "IN
    PUT POSITIVE OCTAL" @ FND=0
    @ GOTO 120
 70 S=S*8+X
 80 O=O-X*D
 90 D=D/10
100 IF D>=1 THEN 50
110 FND=S
120 FN END
130 DISP "INPUT OCTAL NUMBER";
140 INPUT I8
150 DISP FND(I8)
160 GOTO 140
170 END
```

Display:

```
INPUT OCTAL NUMBER?
200
 128
?
201
 129
?
208
INPUT POSITIVE OCTAL
 0
?
-10
INPUT POSITIVE OCTAL
 0
?
255
 173
?
217
 143
?
```

Problem 9.5

```
 10 REM *FACTORIAL FN
 20 INPUT X
 30 DISP X,FNF(X)
 40 GOTO 10
 50 DEF FNF(A)
 60 F=1
 70 FOR P=A TO 1 STEP -1
 80 F=F*P
 90 NEXT P
100 FNF=F
110 FN END
120 END
```

Display:

```
?
6
 6                       720
?
18
 18
 6.40237370574E15
?
12
 12                      479001600
?
10
 10                      3628800
?
```

Problem 9.6

```
 10 REM *ROUND TO 2 DECIMAL PLAC
    ES AND ADD '$'
 20 DEF FNR2$(D)
 30 N=INT(D*100+.5)/100
 40 IF FP(N)=0 THEN FNR2$="$"&VA
    L$(N)&".00" ELSE FNR2$="$"&V
    AL$(N)
 50 FN END
 60 DISP "INPUT NUMERIC VALUE";
 70 INPUT N
 80 DISP FNR2$(N)
 90 GOTO 70
100 END
```

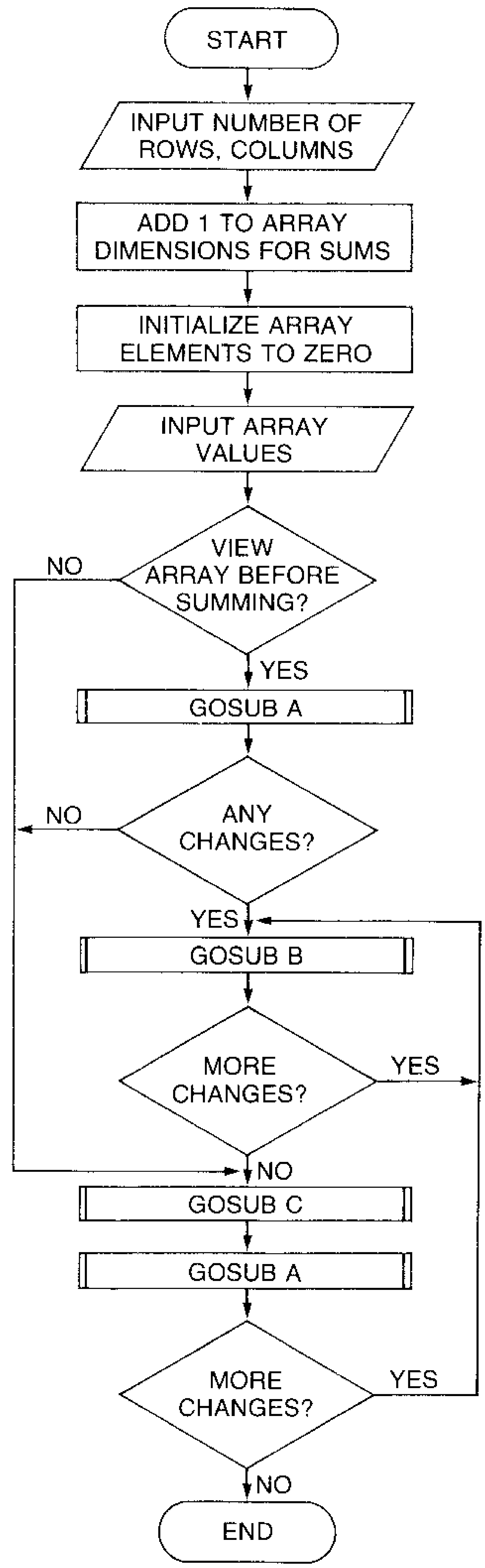
Display:

```
INPUT NUMERIC VALUE?
134.9876
$134.99
?
150
$150.00
?
```

Problem 9.7

```
10 REM *ARRAY ROUTINES*
20 OPTION BASE 1
30 DIM A(10,5)
40 DISP "MAX ARRAY SIZE IS 10 R
   OWS BY 5 COLUMNS"
50 DISP "NUMBER ROWS,COLUMNS"
60 INPUT R1,C1
70 REM *INITIALIZE ARRAY
80 R2=R1+1 @ C2=C1+1
90 FOR R=1 TO R2
100 FOR C=1 TO C2
110 A(R,C)=0
120 NEXT C
130 NEXT R
140 REM *ENTER DATA, ONE VALUE A
    T A TIME, BY ROW.
150 FOR R=1 TO R1
160 DISP "INPUT DATA IN ROW";R
170 FOR C=1 TO C1
180 INPUT A(R,C)
190 NEXT C
200 NEXT R
210 DISP "DO YOU WANT TO SEE THE
     DATA TABLE BEFORE SUMMING (
    Y OR N)";
220 INPUT A$
230 IF A$="N" THEN 310
240 GOSUB 370
250 DISP "ANY CHANGES (Y OR N)";
260 INPUT B$
270 IF B$="N" THEN 310
280 GOSUB 510
290 DISP "MORE CHANGES";
300 GOTO 260
310 GOSUB 560
320 GOSUB 370
330 DISP "CHANGES";
340 INPUT C$
350 IF C$="Y" THEN 280
360 STOP
370 REM *COPY ARRAY
380 DISP "COPY TABLE ON PRINTER
    OR DISPLAY (P OR D)";
390 INPUT Z$
400 IF Z$="P" THEN CRT IS 2
410 DISP
420 DISP
430 FOR R=1 TO R2
440 FOR C=1 TO C2
450 DISP A(R,C);
460 NEXT C
470 DISP
480 NEXT R
490 CRT IS 1
500 RETURN
510 REM *CHANGE ELEMENT
520 DISP "ENTER ROW, COLUMN, VAL
    UE"
530 INPUT R,C,V
540 A(R,C)=V
550 RETURN
560 REM *SUM EACH ROW AND PLACE
    SUM IN LAST COLUMN.
570 FOR R=1 TO R1
580 FOR C=1 TO C1
590 A(R,C2)=A(R,C2)+A(R,C)
600 NEXT C
610 NEXT R
620 REM *SUM EACH COLUMN AND PLA
    CE SUM IN LAST ROW.
630 FOR C=1 TO C1
640 FOR R=1 TO R1
650 A(R2,C)=A(R2,C)+A(R,C)
660 NEXT R
670 NEXT C
680 REM *FIND SUM OF VALUES IN L
    AST ROW (OR COLUMN)
690 FOR C=1 TO C1
700 A(R2,C2)=A(R2,C2)+A(R2,C)
710 NEXT C
720 DISP "SUMMING COMPLETED"
730 RETURN
```

Flowchart:



Problem 9.7 (Cont)

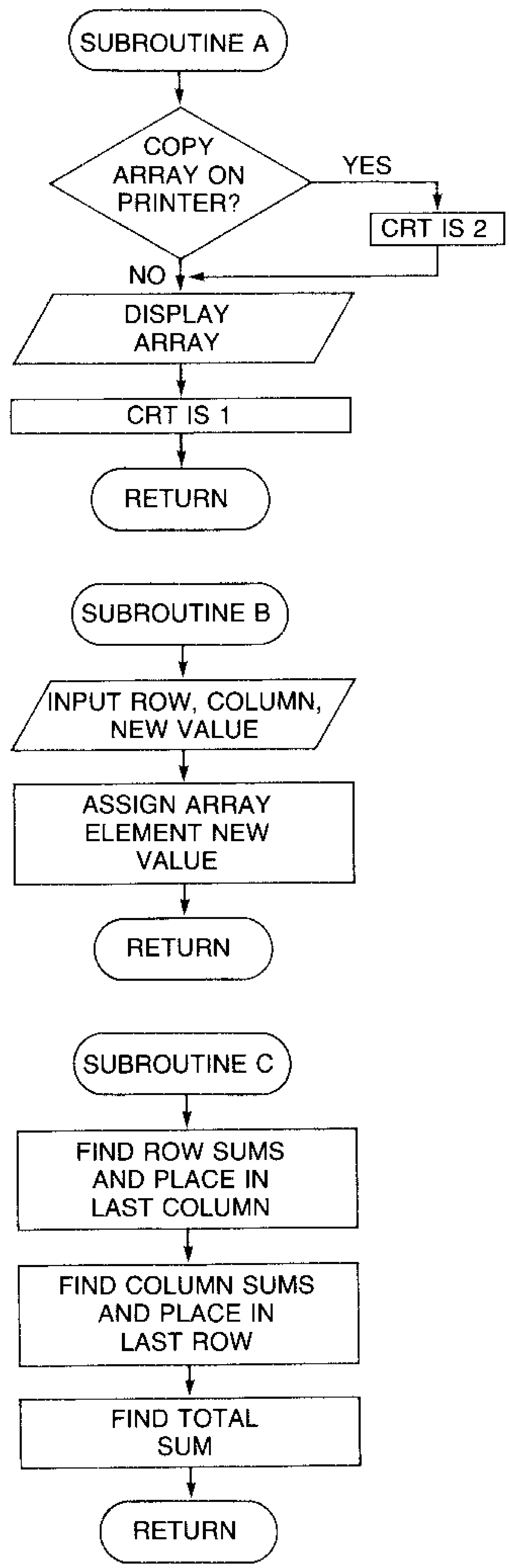
Display:

```
MAX ARRAY SIZE IS 10 ROWS BY 5 C
OLUMNS
NUMBER ROWS,COLUMNS
?
3,3
INPUT DATA IN ROW 1
?
12.59
?
13.69
?
14.78
INPUT DATA IN ROW 2
?
11.43
?
22.56
?
43.78
INPUT DATA IN ROW 3
?
13.52
?
12.78
?
14.98
DO YOU WANT TO SEE THE DATA TABL
E BEFORE SUMMING (Y OR N)?
Y
COPY TABLE ON PRINTER OR DISPLAY
 (P OR D)?
D

 12.59  13.69  14.78  0
 11.43  22.56  43.78  0
 13.52  12.78  14.98  0
 0  0  0  0
ANY CHANGES (Y OR N)?
Y
ENTER ROW, COLUMN, VALUE
?
1,3,14.67
MORE CHANGES?
N
SUMMING COMPLETED
COPY TABLE ON PRINTER OR DISPLAY
 (P OR D)?
P
CHANGES?
N
```

Printer:

```
12.59  13.69  14.67  40.95
11.43  22.56  43.78  77.77
13.52  12.78  14.98  41.28
37.54  49.03  73.43  160
```

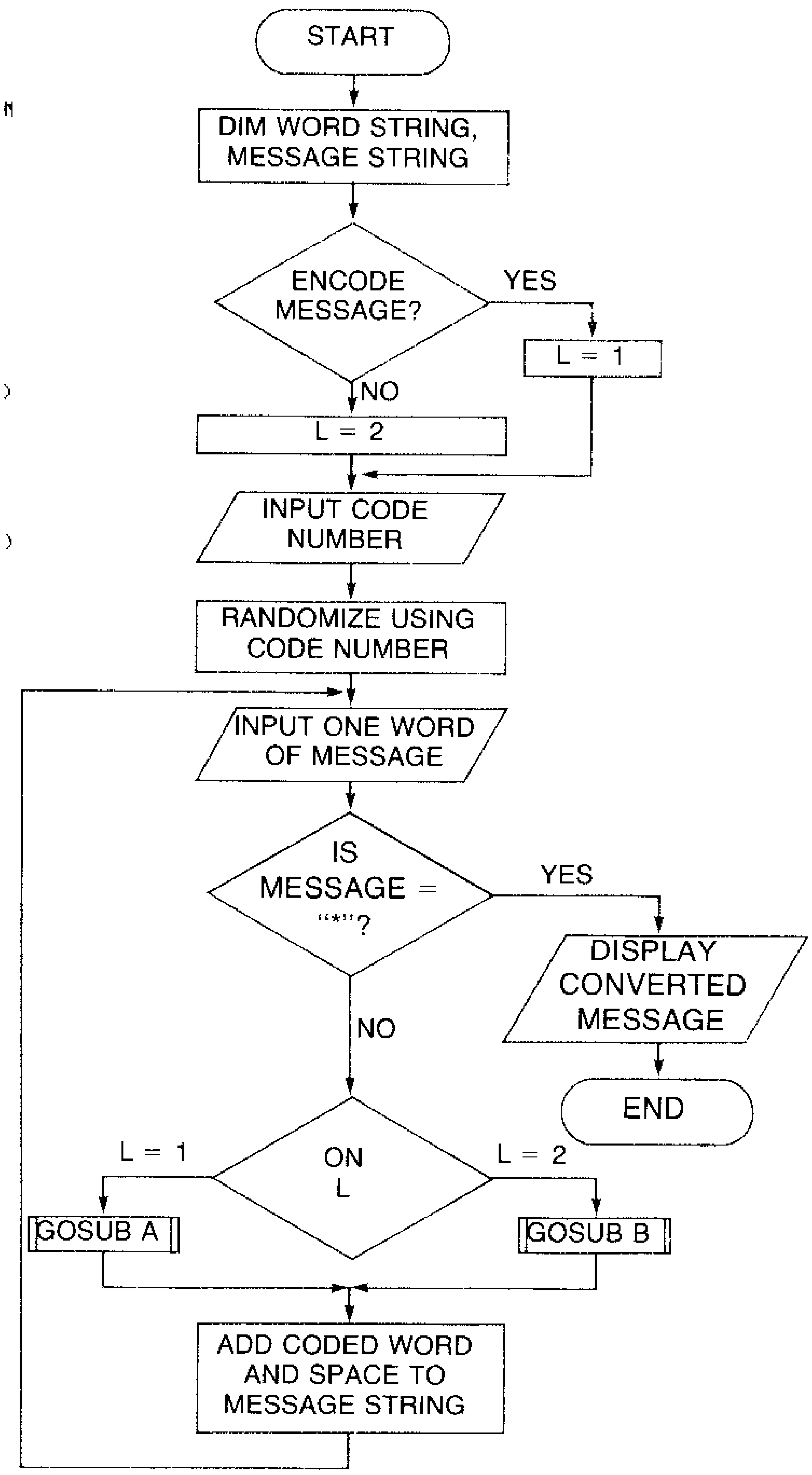
Flowchart (subroutines):



Problem 9.8

```
10 DIM I$[32],F$[1],M$[2000]
20 DISP "CODE OR DECODE: C OR D
   "
30 INPUT F$
40 IF F$="C" THEN L=1 ELSE L=2
50 DISP "CODE NUMBER PLEASE"
60 INPUT S
70 RANDOMIZE S
80 M$=""
90 DISP "TYPE MESSAGE ONE WORD
   AT A TIME. TYPE '*' TO END M
   ESSAGE"
100 DISP "GIVE ME YOUR MESSAGE"
110 INPUT I$
120 IF I$="*" THEN 160
130 ON L GOSUB 1000,2000
140 M$=M$&C$&" "
150 GOTO 110
160 DISP M$
170 END
1000 REM *ENCODING ROUTINE
1010 C$=""
1020 FOR I=1 TO LEN(I$)
1030 C$=C$&CHR$(65+(NUM(I$[I,I])
     +INT(26*RND)) MOD 26)
1040 NEXT I
1050 RETURN
2000 REM *DECODING ROUTINE
2010 C$=""
2020 FOR I=1 TO LEN(I$)
2030 C$=C$&CHR$(65+(NUM(I$[I,I])
     -INT(26*RND)) MOD 26)
2040 NEXT I
2050 RETURN
```

Flowchart:



Problem 9.8 (Cont)

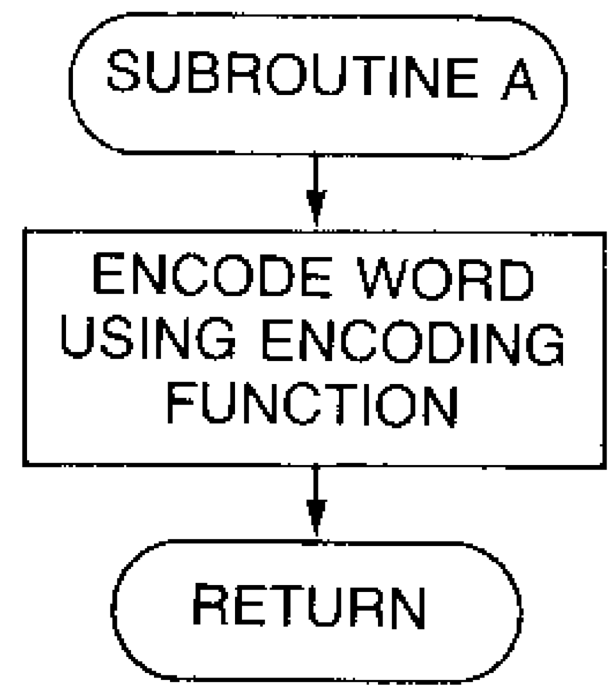
Display:

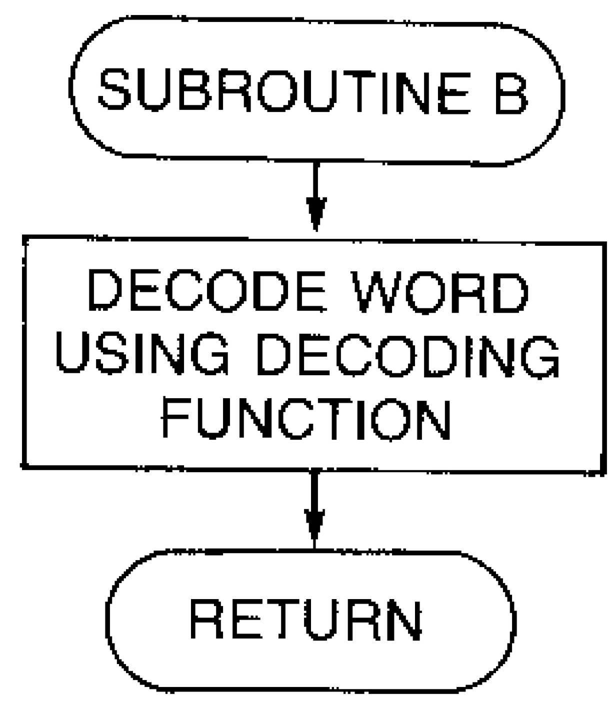
```
CODE OR DECODE: C OR D
?
C
CODE NUMBER PLEASE
?
.123
TYPE MESSAGE ONE WORD AT A TIME.
 TYPE '*' TO END MESSAGE
GIVE ME YOUR MESSAGE
?
GET
?
ME
?
TO
?
THE
?
BANK
?
ON
?
TIME
?
*
EWV SC PR YGB ZSMD NQ VWMS
```

```
CODE OR DECODE: C OR D
?
D
CODE NUMBER PLEASE
?
.123
TYPE MESSAGE ONE WORD AT A TIME.
 TYPE '*' TO END MESSAGE
GIVE ME YOUR MESSAGE
?
EWV
?
SC
?
PR
?
YGB
?
ZSMD
?
NQ
?
VWMS
?
*
GET ME TO THE BANK ON TIME
```

```
CODE OR DECODE: C OR D
?
D
CODE NUMBER PLEASE
?
.3579
TYPE MESSAGE ONE WORD AT A TIME.
 TYPE '*' TO END MESSAGE
GIVE ME YOUR MESSAGE
?
NNLSNUNUS
?
IGPXR
?
RQP
?
BVE
?
*
SYLVESTER WHERE ARE YOU
```

Flowchart (subroutines):





Problem 9.9

```
10 REM *FN KEY ROUTINES*
20 OPTION BASE 1
30 DIM A(20,10)
40 CLEAR
50 ON KEY# 1,"INIT" GOSUB 120
60 ON KEY# 2,"INPUT" GOSUB 230
70 ON KEY# 3,"COPY-A" GOSUB 320
80 ON KEY# 4,"CHANGE" GOSUB 460
90 ON KEY# 5,"SUM" GOSUB 510
100 KEY LABEL
110 GOTO 110
120 DISP "NUMBER ROWS,COLUMNS"
130 INPUT R1,C1
140 REM *INITIALIZE ARRAY
150 R2=R1+1 @ C2=C1+1
160 FOR R=1 TO R2
170 FOR C=1 TO C2
180 A(R,C)=0
190 NEXT C
200 NEXT R
210 DISP "ARRAY INITIALIZED. NOW
  INPUT      YOUR DATA."
220 RETURN
```

Array initialization routine on k1.

```
230 REM *ENTER DATA, ONE VALUE A
  T A TIME, BY ROW.
240 FOR R=1 TO R1
250 DISP "INPUT DATA IN ROW";R
260 FOR C=1 TO C1
270 INPUT A(R,C)
280 NEXT C
290 NEXT R
300 DISP "ARRAY IS FILLED. DATA
  INPUT COMPLETE."
310 RETURN
```

Input data routine on k2.

```
320 REM *COPY ARRAY
330 DISP "COPY TABLE ON PRINTER
  OR DISPLAY (P OR D)";
340 INPUT Z$
350 IF Z$="P" THEN CRT IS 2
360 DISP
370 DISP
380 FOR R=1 TO R2
390 FOR C=1 TO C2
400 DISP A(R,C);
410 NEXT C
420 DISP
430 NEXT R
440 CRT IS 1
450 RETURN
```

Display or print array on k3.

```
460 REM *CHANGE ELEMENT
470 DISP "ENTER ROW, COLUMN, VAL
  UE"
480 INPUT R,C,V
490 A(R,C)=V
500 RETURN
```

Change array element on k4.

```
510 REM *SUM EACH ROW AND PLACE
  SUM IN LAST COLUMN.
520 FOR R=1 TO R1
530 FOR C=1 TO C1
540 A(R,C2)=A(R,C2)+A(R,C)
550 NEXT C
560 NEXT R
570 REM *SUM EACH COLUMN AND PLA
  CE SUM IN LAST ROW.
580 FOR C=1 TO C1
590 FOR R=1 TO R1
600 A(R2,C)=A(R2,C)+A(R,C)
610 NEXT R
620 NEXT C
630 REM *FIND SUM OF VALUES IN L
  AST ROW (OR COLUMN)
640 FOR C=1 TO C1
650 A(R2,C2)=A(R2,C2)+A(R2,C)
660 NEXT C
670 DISP "SUMMING COMPLETED"
680 RETURN
```

Find sum of rows, columns, and total on k5.

```
--------------------------------
SUM
INIT     INPUT     COPY-A   CHANGE
```

The key labels appear on the display and the program waits for you to press a special function key. In this program, you must initialize the array before you do anything else. So, first press k1 and answer the question that appears on the display for summing the rows and columns of some tables of your own.

Section 10

Problem 10.1

```
 10 REM *NATIONAL SUMMARIES
 20 PRINT USING 30
 30 IMAGE "  POPULATION",5X,"ARE
    A   POP DENS"/
 40 PRINT USING 50
 50 IMAGE 7X,"ANNUAL GNP",5X,"GN
    P/PERS"/
 60 PRINT USING 70
 70 IMAGE 32("_")
 80 DISP "NATION"
 90 INPUT N$
100 IF LEN(N$)=0 THEN STOP
110 DISP "POP, AREA, GNP";
120 INPUT P,A,G
130 PRINT USING  "K,//" ; N$
140 PRINT USING 150 ; P;A;P/A
150 IMAGE X,2(3DC3DC3D),2X,DCDDZ
    .D/
160 PRINT USING 170 ; G;G/P
170 IMAGE "    $",DC3DC3DC3DC3D,"
      $",2DC3D//
180 GOTO 80
190 END
```

Display:

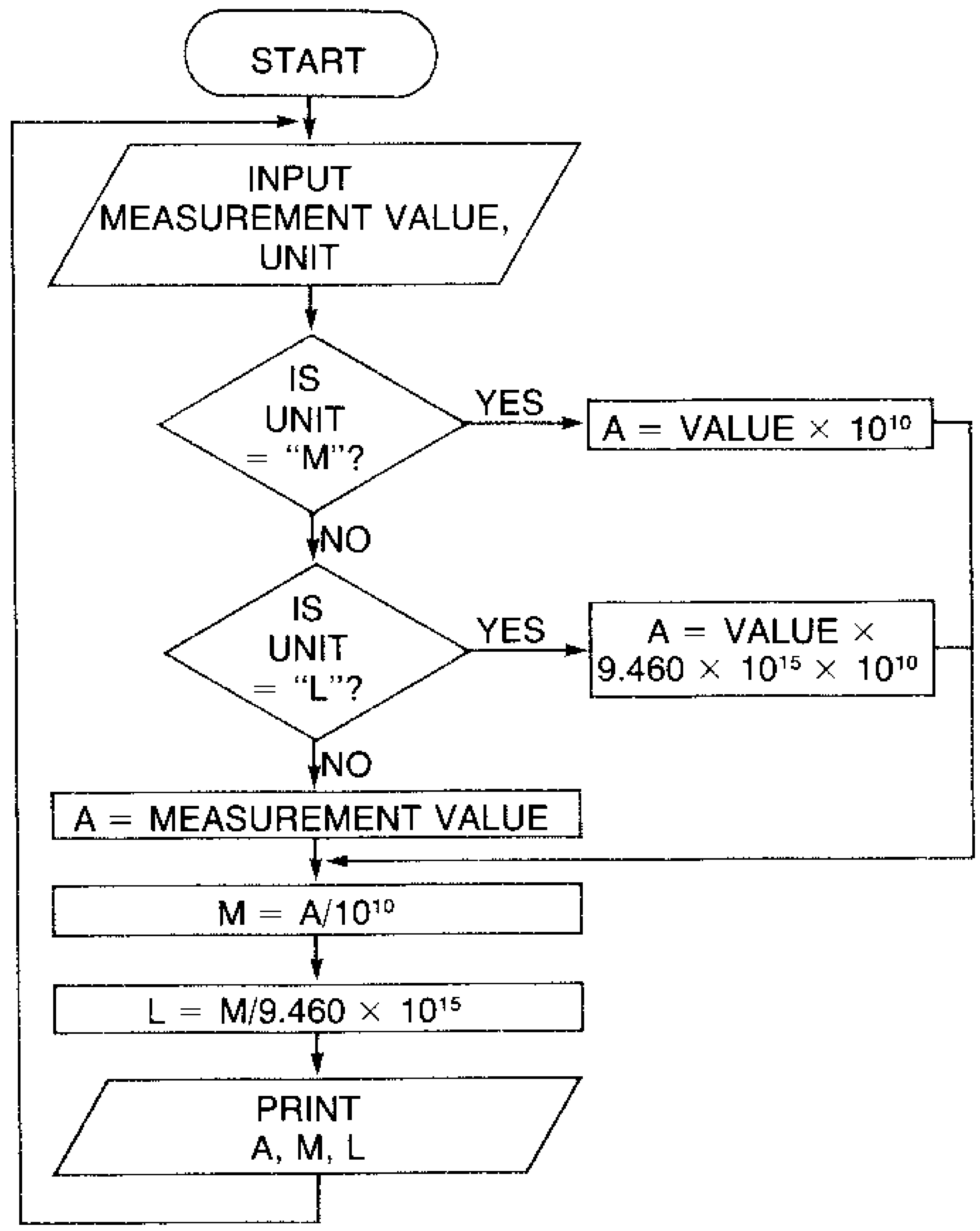
```
NATION
?
CHINA
POP, AREA, GNP?
865193550,9560980,223000000000
NATION
?
UNITED STATES
POP, AREA, GNP?
216817000,9363123,1781400000000
NATION
?
CANADA
POP, AREA, GNP?
23469142,9976139,195785000000
NATION
?
SINGAPORE
POP, AREA, GNP?
2322576,581,5885600000
NATION
?
MONGOLIA
POP, AREA, GNP?
1531940,1565000,547000000
NATION
?
QATAR
POP, AREA, GNP?
97792,11000,4044000000
NATION
?
```

Printer:

```
  POPULATION      AREA   POP DENS

       ANNUAL GNP      GNP/PERS
________________________________
CHINA

 865,193,550  9,560,980      90.5

   $  223,000,000,000  $    258

UNITED STATES

 216,817,000  9,363,123      23.2

   $1,781,400,000,000  $ 8,216

CANADA

  23,469,142  9,976,139       2.4

   $  195,785,000,000  $ 8,342

SINGAPORE

   2,322,576        581   3,997.5

   $      5,885,600,000  $ 2,534

MONGOLIA

   1,531,940  1,565,000       1.0

   $        547,000,000  $    357

QATAR

      97,792     11,000       8.9

   $      4,044,000,000  $41,353
```

Problem 10.2

Flowchart:




```
10 REM *EXTREMES
20 PRINT USING 30
30 IMAGE "ANGSTROMS     METERS
   LIGHT-YEARS"
40 PRINT USING 50
50 IMAGE 32("_")
60 DISP "UNITS: A,M,L"
70 DISP "VALUE comma UNITS";
80 INPUT X,U$
90 IF U$="M" THEN 170
100 IF U$="L" THEN 190
110 A=X
120 M=A/10000000000
130 L=M/9.46E15
140 PRINT USING 150 ; A;M;L
150 IMAGE 3(D.3DE,X)
160 GOTO 70
170 A=X*10000000000
180 GOTO 120
190 A=X*9.46E15*10000000000
200 GOTO 120
210 END
```

Display:

```
UNITS: A,M,L
VALUE comma UNITS?
5560,A
VALUE comma UNITS?
1410,M
VALUE comma UNITS?
5.6E-3,A
VALUE comma UNITS?
1E-14,M
VALUE comma UNITS?
170000,L
VALUE comma UNITS?
```

Printer:

```
ANGSTROMS     METERS  LIGHT-YEARS
--------------------------------
5.560E+003 5.560E-007 5.877E-023

1.410E+013 1.410E+003 1.490E-013

5.600E-003 5.600E-013 5.920E-029

1.000E-004 1.000E-014 1.057E-030

1.608E+031 1.608E+021 1.700E+005
```

Problem 10.3

```
 10 REM *CHECKING ACCT
 20 DISP "REFERENCE DATE";
 30 INPUT T$
 40 DISP "BEGINNING BALANCE";
 50 INPUT B
 60 PRINT "SUMMARY FOR ";T$
 70 PRINT USING 80
 80 IMAGE /" CHECKS   CHG DEPOSI
    TS   BALANCE"
 90 PRINT USING 100
100 IMAGE 32("_")
110 PRINT USING  "/24X,DCDDZ.DD"
     ; B
120 DISP "TRANSACTION (C,D), AMT
    ";
130 INPUT C$,A
140 IF C$="C" THEN 170
150 IF C$="D" THEN 250
160 STOP
170 F=0
180 IF B<275 THEN F=.22
190 B=B-A-F
200 PRINT USING 210 ; A;F,B
210 IMAGE DCDDZ.DD,2X,.DD,11X,DC
    DDZ.DD
220 IF B<0 THEN PRINT USING 230
    ; -B
230 IMAGE /"**WARNING   $",***Z.
    DD," OVERDRAFT**"/
240 GOTO 120
250 B=B+A
260 PRINT USING 270 ; A;B
270 IMAGE 12X,2(2X,DCDDZ.DD)
280 GOTO 120
290 END
```

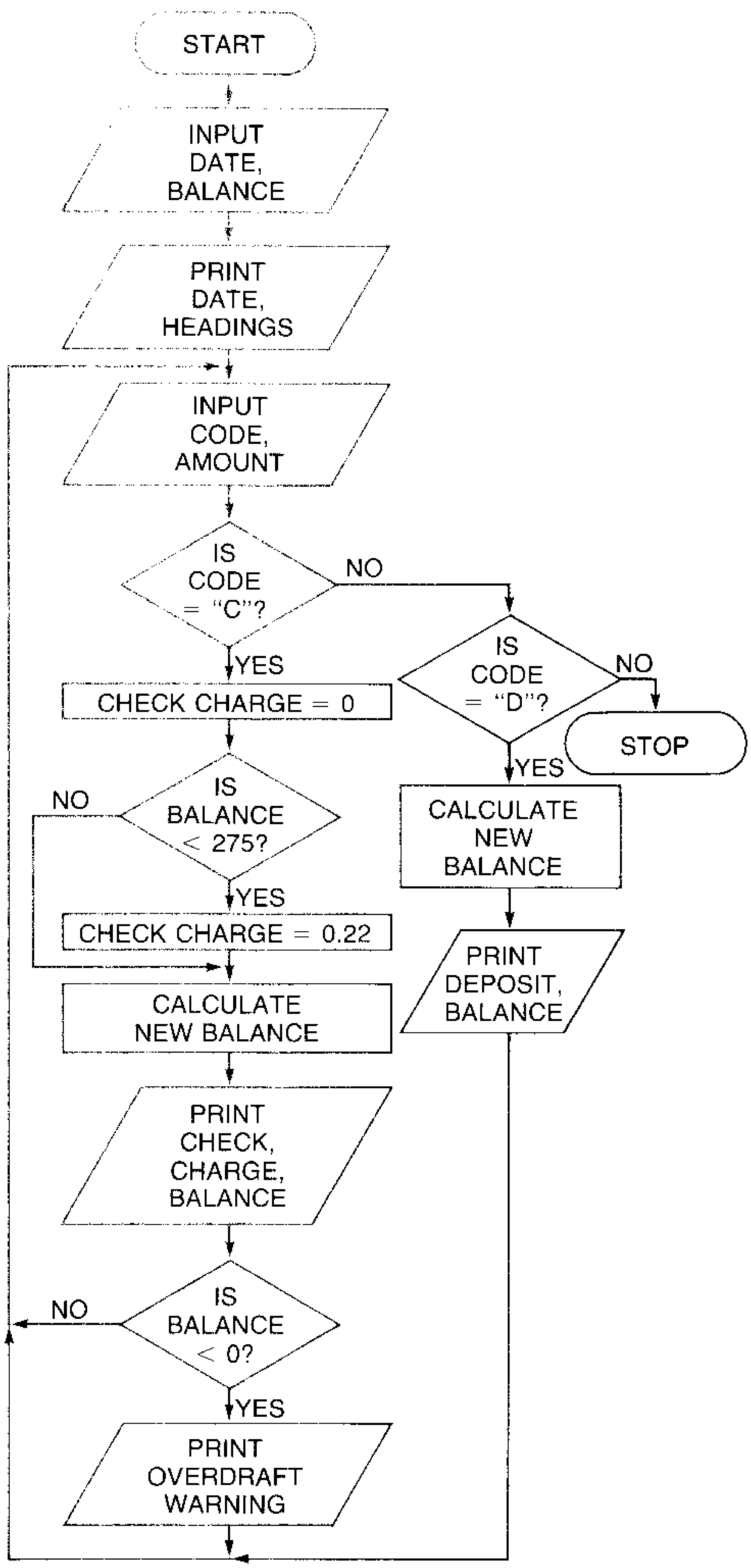
Display:

```
REFERENCE DATE?
JUNE 1979
BEGINNING BALANCE?
1027.41
TRANSACTION (C,D), AMT?
C,850.00
TRANSACTION (C,D), AMT?
C,54.79
TRANSACTION (C,D), AMT?
D,55.45
TRANSACTION (C,D), AMT?
C,185.49
TRANSACTION (C,D), AMT?
D,255.00
TRANSACTION (C,D), AMT?
```

Printer:

```
SUMMARY FOR JUNE 1979

 CHECKS   CHG DEPOSITS   BALANCE
--------------------------------

                        1,027.41
  850.00  .00             177.41
   54.79  .22             122.40
              55.45       177.85
  185 49  .22          -    7.86

**WARNING   $***7.86 OVERDRAFT**

              255 00      247.14
```

Flowchart:



Problem 10.4

```
10 REM *POLYGONS
20 RAD
30 PRINT " n";TAB(9),"p";TAB(24
   );"p/d"
40 PRINT
50 N=3
60 D=35
70 P=D*N*SIN(PI/N)
80 PRINT N;TAB(2);P;TAB(18);P/D
90 N=N+1
100 GOTO 70
110 END
```

Printer:

```
n        p                p/d

3
 90.9326673975    2.59807621136
4
 98.9949493662    2.82842712475
5
 102.862419151    2.93892626146
6
 105              3
7
 106.301516084    3.03718617383
8
 107.151361062    3.06146745891
9
 107.736345148    3.07818128994
10
 108.155948031    3.09016994374
11
 108.467034384    3.09905812526
12
 108.703998943    3.10582854123
13
 108.888627251    3.11110363574
14
 109.035257638    3.11529307537
15
 109.153637679    3.11867536226
```

Problem 10.5

```
10 REM *WEIGHT PLOT
20 DATA 3.2,9.5,11.9,13.9,15.7,
   17.6,19.1,21.9,24.8,28.1,32.
   4,37.1,41.5,46.2,50.5
30 DATA 53.8,55.7,56.7,56.7
40 PRINT USING 50 ; 0;10;20;30;
   40;50
50 IMAGE 2X,D,X,5(3X,2D)
60 PRINT USING 70
70 IMAGE "  +",5("----+")
80 FOR I=0 TO 18
90 READ W
100 PRINT VAL$(I);TAB(3);"+";TAB
   (3+W/2);"*"
110 NEXT I
120 END
```

Printer:

```
  0    10   20   30   40   50
  +----+----+----+----+----+
0 + *
1 +    *
2 +     *
3 +      *
4 +       *
5 +        *
6 +         *
7 +          *
8 +           *
9 +             *
10+               *
11+                  *
12+                    *
13+                      *
14+                        *
15+                          *
16+                           *
17+                           *
18+                           *
```

Section 12

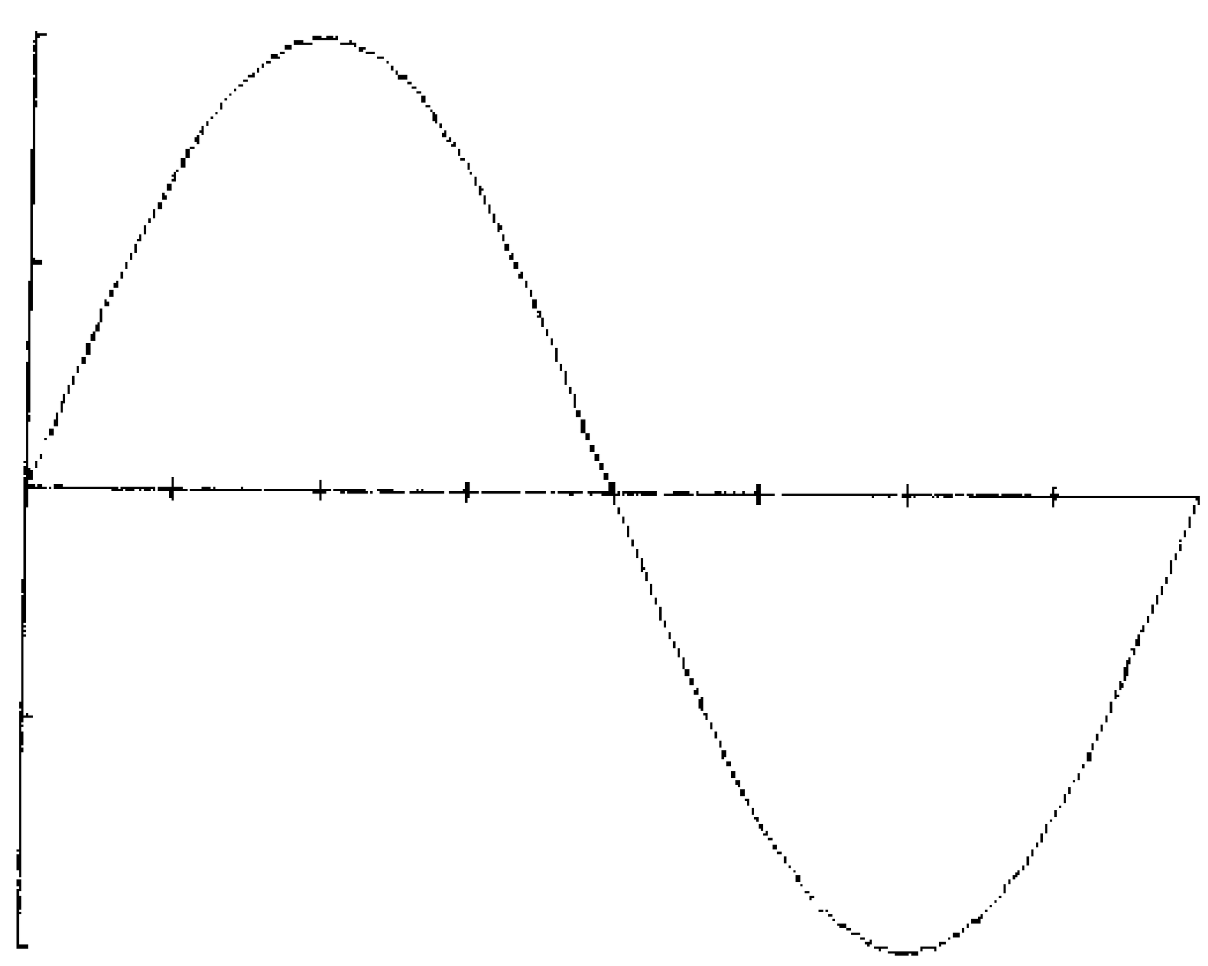
Problem 12.1

```
10 REM *CARDIOID
20 PEN 1 @ GCLEAR
30 SCALE -3,1,-2,2
40 RAD
50 FOR T=0 TO 2*PI STEP PI/25
60 MOVE 0,0
70 R=1-COS(T)
80 DRAW R*COS(T),R*SIN(T)
90 NEXT T
100 END
```

Problem 12.2

```
 10 REM *PAD SINE CURVE
 20 GCLEAR
 30 SCALE 0,2*PI,-1,1
 40 XAXIS 0,PI/4
 50 YAXIS 0,.5
 60 RAD
 70 MOVE 0,0
 80 FOR X=0 TO 2*PI+.3 STEP PI/2
    0
 90 DRAW X,SIN(X)
100 NEXT X
110 END
```

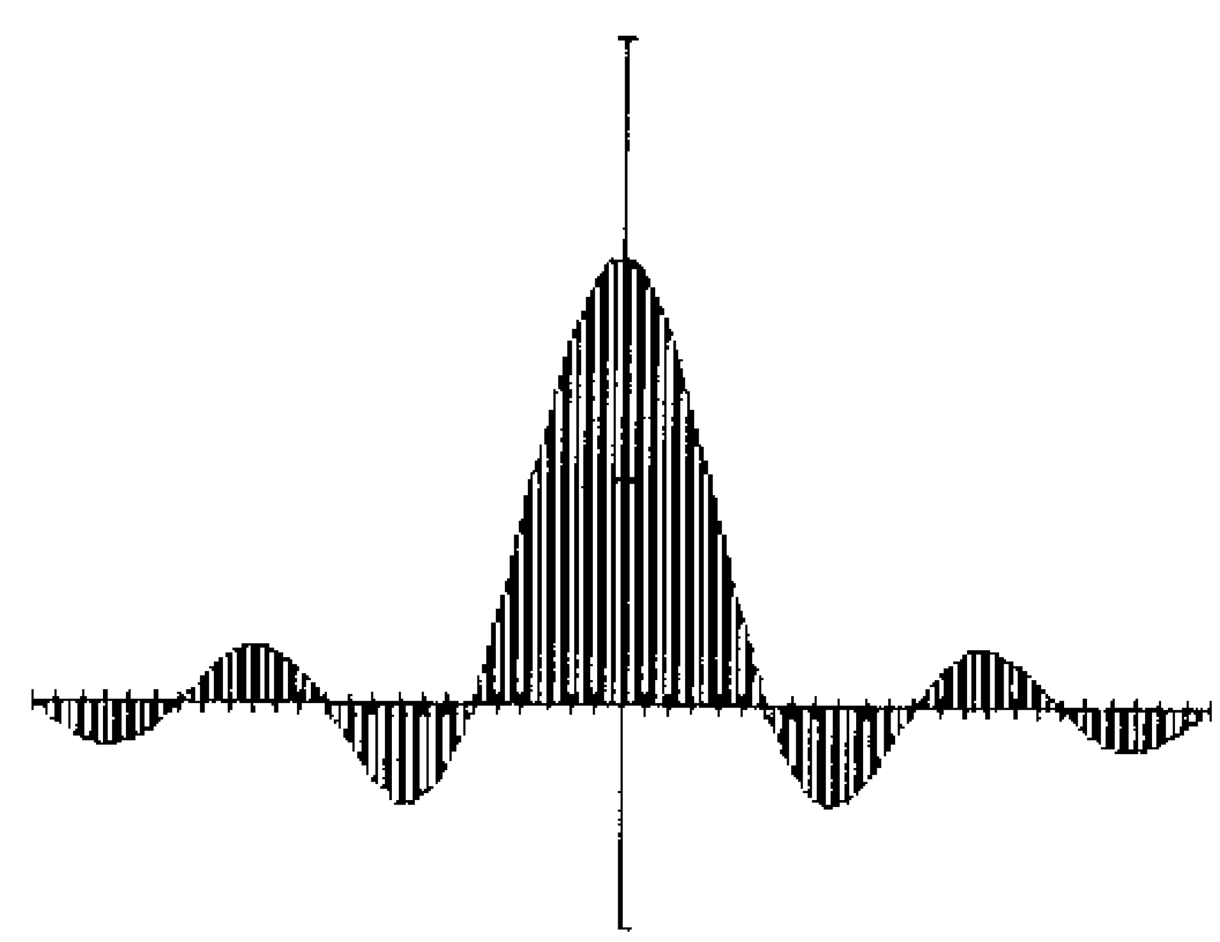
Display:



Problem 12.3

```
 10 REM *FILL SIN(X)/X
 20 GCLEAR
 30 RAD
 40 SCALE -4*PI,4*PI,-.5,1.5
 50 YAXIS 0,.5
 60 XAXIS 0,PI/6
 70 FOR X=-4*PI TO 4*PI STEP PI/
    20
 80 IF X=-4*PI THEN MOVE X,SIN(X
    )/X
 90 IF X=0 THEN 130
100 DRAW X,SIN(X)/X
110 MOVE X,0
120 DRAW X,SIN(X)/X
130 NEXT X
140 END
```

Display:



Problem 12.4

Run two programs back to back. The SCALE statement in the first program would be:

```
60 SCALE -36000,0,-36000,36000
```

The SCALE statement in the second program would be:

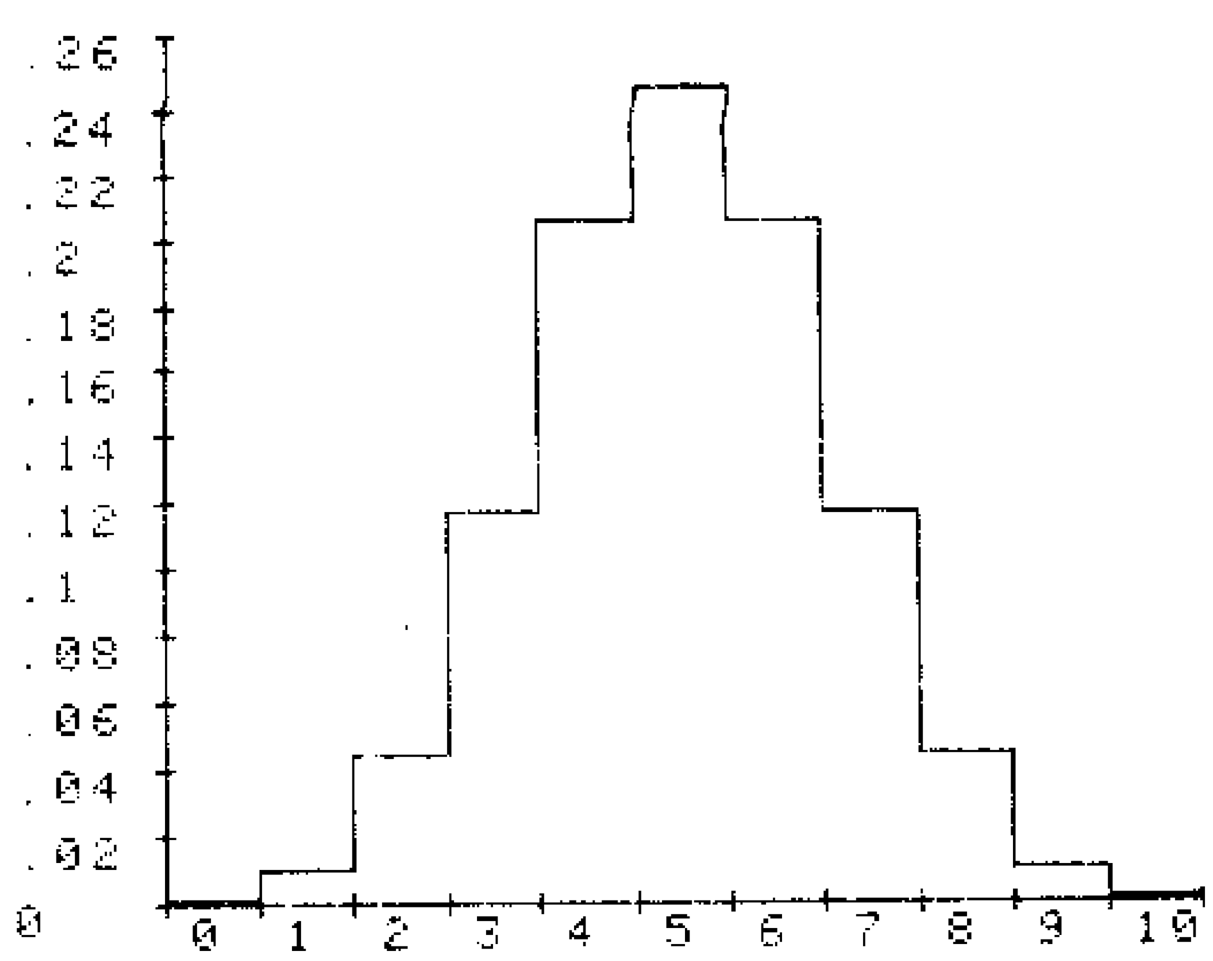
```
60 SCALE 0,36000,-36000,36000
```

Before you run the second program, copy the graphics screen onto the printer. When you copy the graphics from the second program, without advancing the paper, you will have a spiral twice as wide as the original design.

Problem 12.5

```
10 REM *DISTRIBUTION OF HEADS*
20 GCLEAR @ PEN 1
30 SCALE -1.5,11,-.015,.26
40 XAXIS 0,1,0,11
50 YAXIS 0,.02,0,.26
60 REM *LABEL X-AXIS*
70 LDIR 0
80 FOR X=0 TO 10
90 MOVE X+.3,-.015
100 LABEL VAL$(X)
110 NEXT X
120 REM *LABEL Y-AXIS*
130 FOR Y=0 TO .26 STEP .02
140 MOVE -1.5,Y-.01
150 LABEL VAL$(Y)
160 NEXT Y
170 REM *PLOT HISTOGRAM*
180 N=10
190 P,Q=.5
200 PRINT "# HEADS","PROBABILITY
    "
210 MOVE 0,0
220 FOR R=0 TO 10
230 D=P^(N-R)*Q^R*FNF(N)/(FNF(R)
    *FNF(N-R))
240 PRINT R,INT(D*1000+.5)/1000
250 DRAW R,D
260 IDRAW 1,0
270 NEXT R
280 DRAW 11,0
290 DEF FNF(X)
300 F=1
310 FOR I=X TO 1 STEP -1
320 LET F=F*I
330 NEXT I
340 FNF=F
350 FN END
360 END
```

Display:



Printer:

# HEADS	PROBABILITY
0	.001
1	.01
2	.044
3	.117
4	.205
5	.246
6	.205
7	.117
8	.044
9	.01
10	.001

Problem 12.6

```
10 REM *HANGMAN
20 DIM X(10),D$[25],R$[25],A$[2
   8],Q$[52],C$[1],G$[1],B$[26]
30 INTEGER Y(140)
40 Q$="abcdefghijklmnopqrstuvwx
   yzABCDEFGHIJKLMNOPQRSTUVWXYZ
   "
50 I1=1
60 GCLEAR @ CLEAR
61 SCALE -5,35,-10,20
62 DEG
70 DISP "HAVE A FRIEND ENTER A
   WORD--AT MOST 23 CHARACTERS,
   SPACES NOT ALLOWED--NO PEEKI
   NG!!"
80 DISP "WORD";
90 INPUT A$
100 IF POS(A$," ")#0 THEN 80
101 CLEAR ! CLEAR ALPHA DISPLAY
110 GOSUB 1000 ! DRAW SCAFFOLD
120 L=LEN(A$)
130 L1=12-INT(L/2)*1.25
140 R$[1,25]=""
150 D$[1,25]=""
160 B$[1,26]=""
170 R$=A$
180 C=0
190 J=0
200 D$[1,L]="-------------------
    ---------------------------
    --"
210 MOVE L1,-2
220 LABEL D$
230 MOVE -5,-4 @ GCLEAR -2
240 LABEL "WHAT IS YOUR GUESS?"
250 MOVE -5,-6 @ GCLEAR -4
260 LABEL "YOU HAVE "&VAL$(6-C)&
    " GUESSES LEFT" @ MOVE -5,-8
270 C$="-"
280 INPUT C$
310 X=POS(Q$,C$)
320 IF X=0 THEN 250
330 IF X>26 THEN 360
340 X=X+26
350 C$=Q$[X,X]
360 X=POS(B$,C$)
370 IF X=0 THEN 530
380 DISP "YOU TURKEY! YOU ALREAD
    Y GUESSED THAT! NOW TRY AGAI
    N "
390 BEEP
400 WAIT 4000
410 GOTO 250
530 B$[J+C+1]=C$
540 X=POS(R$,C$)
550 IF X=0 THEN 810
560 N=1
570 J=J+1
575 ! CHECK TO SEE IF SAME LETTE
    R OCCURS MORE THAN ONCE
580 D$[X,X]=C$
590 R$[X,X]=" "
595 ! BLANK OUT OCCURENCES OF LE
    TTERS TO CHECK FOR MORE
600 X(N)=X
610 X=POS(R$,C$)
620 IF X=0 THEN 670
640 B$[J+C+1]=C$
650 N=N+1
660 GOTO 570
670 FOR I=1 TO N
680 MOVE L1,-1
685 FOR I4=1 TO X(I)-1
690 IMOVE 1.27,0
700 NEXT I4
705 LABEL C$
710 ! RESTORE WORD TO ORIGINAL F
    ORM
720 R$[X(I),X(I)]=C$
730 NEXT I
740 IF J<L THEN 250
750 BEEP
```

(Continued on next page.)

Problem 12.6 (Cont)

```
760 DISP D$;"  AW SHUCKS! YOU WI
    N!"
765 WAIT 1000
770 GRAPH
780 WAIT 3000
790 GOTO 880
810 DISP C$;" --WRONG!!!!!!"
820 WAIT 400
830 C=C+1
840 GOSUB 2000
850 IF C<6 THEN 250
860 BEEP
865 PRINT "HA HA!! I WIN!!!"
870 PRINT "BY THE WAY, THE WORD
    WAS ";R$
880 DISP "DO YOU WANT TO PLAY AG
    AIN (Y/N)?"
890 INPUT C$
900 IF UPC$(C$)="Y" THEN 50
910 IF UPC$(C$)="N" THEN 930
920 GOTO 900
930 GRAPH
940 END
1000 ! DRAW SCAFFOLD
1010 GCLEAR
1020 PENUP
1030 MOVE 8,1
1040 IDRAW -1,0 @ IDRAW 0,18
1050 IDRAW 7,0 @ IDRAW 0,-3 @ ID
     RAW 0,2
1060 IDRAW -6,0 @ IDRAW 0,-17
1070 IMOVE 0,14 @ IDRAW 3,3
1080 IMOVE 1,0 @ IDRAW -4,-4
1090 RETURN
2000 ON C GOTO 2010,2400,2600,28
     00,3000,3200
2010 ! DRAW THE HEAD
2020 Y=COS(45)
2030 X=1-Y
2040 MOVE 14,15
2050 FOR I=1 TO 2
2060 IDRAW -Y,-X
2070 IDRAW -X,-Y
2080 IDRAW X,-Y
2090 IDRAW Y,-X
2100 X=-X @ Y=-Y
2110 NEXT I
2120 IMOVE 0,-2
2130 IDRAW 0,-.2
2140 GOTO 4000
2400 ! DRAW THE LEFT ARM
2410 MOVE 14,12.5
2420 IDRAW -3,-2.5
2430 GOTO 4000
2600 ! DRAW THE RIGHT ARM
2610 MOVE 14,12.5
2620 IDRAW 3,-2.5
2630 GOTO 4000
2800 ! DRAW THE TRUNK
2810 MOVE 14,12.5
2820 IDRAW 0,-4.5
2830 GOTO 4000
3000 ! DRAW THE LEFT LEG
3010 MOVE 14,8
3020 IDRAW -3,-5
3030 IDRAW -1,0
3040 GOTO 4000
3200 ! DRAW THE RIGHT LEG
3210 MOVE 14,8
3220 IDRAW 3,-5
3230 IDRAW 1,0
4000 RETURN
```

Problem 12.7

```
10 REM *CREATE LANDSCAPE
1000 PEN 1 @ GCLEAR
1010 DEG
1020 SCALE 0,255,0,191
1030 REM *DRAW MOUNTAINS
1040 PENUP
1050 FOR A=-15 TO 255 STEP 15
1060 PLOT A+15,50*(COS(180/255*A
)+1)
1070 NEXT A
1080 PENUP
1090 FOR B=106 TO 255 STEP 15
1100 PLOT B,100*SIN(90/255*B)+10
```

Problem 12.7 (Cont)

```
1110 NEXT B
1120 REM *DRAW LAKE
1130 XAXIS 10,0,0,190
1140 H=100/TAN(10)+10
1150 R=100/SIN(10)
1160 PENUP
1170 FOR C=170 TO 190 STEP 1
1180 PLOT R*SIN(C)+90,R*COS(C)+H
1190 NEXT C
1200 REM *PLOT TREES
1210 T$=CHR$(8)&CHR$(28)&CHR$(62
     )&CHR$(127)
1220 FOR X=4 TO 240 STEP 60
1230 MOVE X,25
1240 GOSUB 2000
1250 NEXT X
1260 FOR X=52 TO 200 STEP 40
1270 MOVE X,29
1280 GOSUB 3000
1290 NEXT X
1300 MOVE 15,35
1310 GOSUB 3000
1320 MOVE 100,84
1330 GOSUB 2000
1340 MOVE 4,116
1350 GOSUB 3000
1360 MOVE 190,15
1370 GOSUB 3000
1380 MOVE 248,124
1390 GOSUB 2000
1400 MOVE 240,124
1410 GOSUB 3000
1420 MOVE 220,50
1430 GOSUB 3000
1440 MOVE 200,60
1450 GOSUB 3000
1460 REM *DRAW CLOUDS
1470 XAXIS 190,0,150,250
1480 XAXIS 185,0,140,253
1490 XAXIS 180,0,130,255
1500 XAXIS 175,0,135,250
1510 XAXIS 170,0,145,245
1520 XAXIS 165,0,150,225
1530 GOTO 5000
2000 FOR I=1 TO 3
2010 BPLOT T$,1
2020 NEXT I
2030 RETURN
3000 FOR I=1 TO 4
3010 BPLOT T$,1
3020 NEXT I
3030 RETURN
5000 REM *BPLOT MAN IN MOON*
5010 DIM M$[46],M2$[69]
5020 MOVE 0,191
5030 GOSUB 5140
5040 BPLOT M$,2
5050 REM *MOVE MOON
5060 FOR Y=191 TO 0 STEP -1
5070 FOR X=0 TO 255 STEP 8
5080 WAIT 1000
5090 MOVE X,Y
5100 BPLOT M2$,3
5110 MOVE X,Y
5120 NEXT X
5130 NEXT Y
5140 M$="E◀⊢◀?@▒E‡xñ⊥ñΣΓατα▒▒⊢⊢⊢
     ⊢Sα‡_▒_◀_◀αċ⊥?x?E⊢`⊢◀E◀"
5150 M2$="EE◀⊢⊢◀?⊢@▒αE‡wxññ⊥ñ×ΣΓ
     iατaα▒◀▒⊢◀⊢⊢◀⊢S<α‡P_▒@_◀__◀
     nnċ±⊥?Gx?0E⊢▒`⊢⊢◀EE◀"
5160 RETURN
5170 END
```

Index

Bold page numbers denote primary references, regular page numbers denote secondary references.

T

U

V

W-Z

HEWLETT
PACKARD

00085-90002 Rev. D 1/81

Printed in U.S.A.