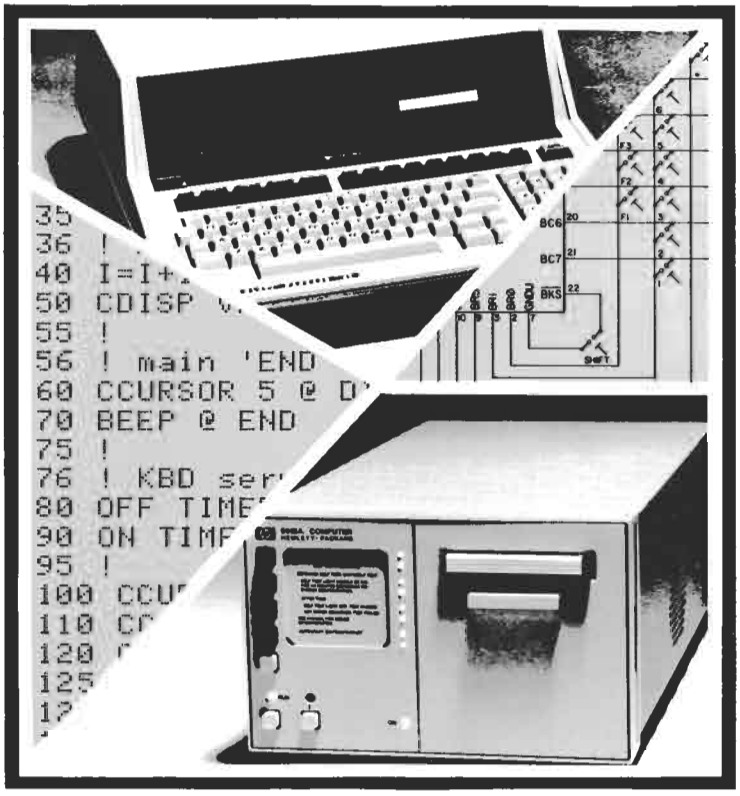
HP 9915
System Development Manual



HEWLETT PACKARD

HP 9915A Modular Computer System Development Manual

Manual Part No. 09915-90010
Microfiche No. 09915-99010

Hewlett-Packard Desktop Computer Division
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

January 1981...First Edition

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Table of Contents

Chapter 1
Table of Contents

System Development Overview

Chapter 1
System Development Overview

Introduction

This manual serves as the **main document** of the HP 9915 manual set; it describes each manual in the set and ties the information together. It also describes the language enhancements provided by the Program Development ROM. The other manuals "fill in the details" of the mainframe language, the I/O language, hardware design of keyboards, requirements of an external CRT display, external system control and status line specifications, and additional program-file operations. This manual will guide you through the manual set as you need the information as you are designing your system.

HP-85 Description

The HP-85 computer incorporates an interpretive-BASIC operating system, and a built-in thermal printer, CRT display, tape drive and autostart routine. Optional plug-in I/O modules (or cards) and the supporting I/O language statements in the I/O Option ROM are available for the computer. An optional memory module and ROM drawer are also available. The HP-85 can be equipped with the optional I/O and Program Development (PD) ROMs; this is the minimum ROM configuration used while developing programs for the HP 9915.

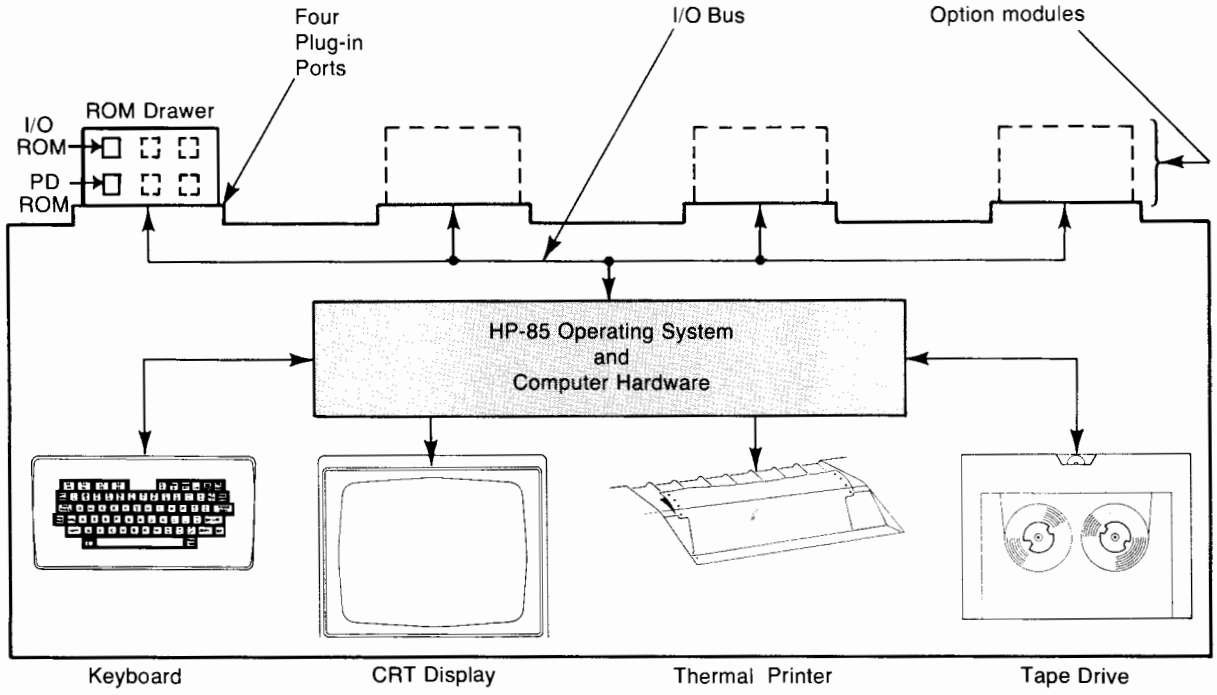


Figure 1-1. HP-85 Computer Block Diagram

HP 9915 Description

The HP 9915, despite its outward differences, has all the capabilities of the HP-85. The 9915 also has features which extend its capabilities beyond those of the HP-85: it is enclosed in a rack-mountable instrument case and has a wider ambient temperature range; it has an extensive, built-in, hardware-test procedure; programs can be stored internally in PROM or externally in other computers; and the 9915 can be controlled by an external computer.

The HP 9915 Modular Computer does not have standard tape drive, CRT, printer, or keyboard. You are free to choose these additional components for your system as needed. The 9915 is shown below to compare it to the HP-85 (see Figure 1-1). Notice that the I/O and PD ROMs are standard features of the 9915.

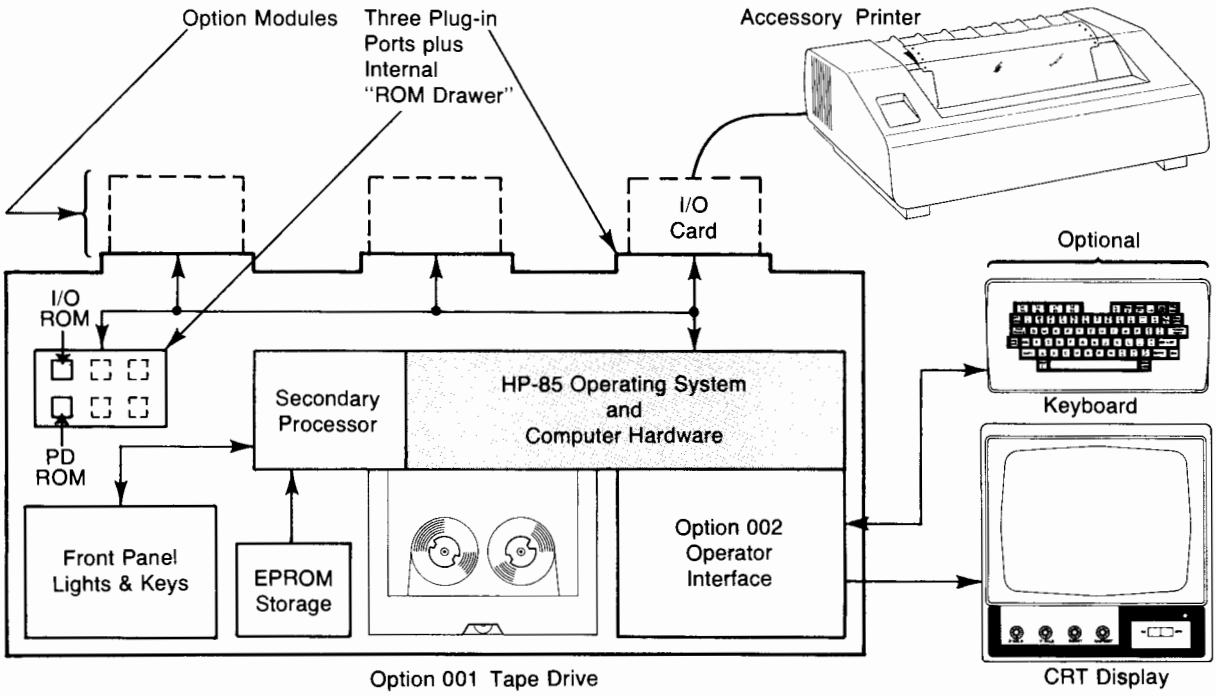


Figure 1-2. HP 9915 Hardware Description

Overview of the Program Development ROM Enhancements

The PD ROM provides additional BASIC-language statements which:

- allow greater control of the keyboard and CRT display so that you may redefine their response characteristics for your needs (Option 002 is required with the 9915)
- allow loading programs from another computer (which includes another 9915 or HP-85), using an interface card as the transfer medium
- allow the use of internal PROM (or EPROM) for program storage in the 9915
- allow control of the eight user lights (simulated on the HP-85 CRT)
- give you lockout control of the SELF TEST and AUTOSTART keys on the 9915
- allow the computer to more completely test its hardware operation

Documentation

This manual describes the following topics: additional language statements provided by the Program Development (or PD) ROM; software development for the 9915 using the HP-85 equipped with the PD and I/O ROMs; hardware and software differences between the two computers; and where to look in the other manuals in the set to find relevant information.

Since the HP-85 and the 9915 have the same operating system, the HP-85 documentation will be used with the 9915. The manual set for the 9915 includes the following 9915 and HP-85 manuals:

- *HP 9915 Installation Manual* [1] (09915-90000)
- *HP 9915 System Development Manual* [1] (09915-90010)
- *Operator Interface Technical Supplement* [1,2] (09915-90021)
- *Networking Technical Supplement* [1,2] (09915-90022)
- *Tape Duplication and EPROM Programming Software Applications Pack* [1] (09915-10010)
- *HP 9915 Service Manual* (09915-90030)
- *HP-85 Owner's Manual and Programming Guide* (00085-90002)
- *HP-85 I/O Programming Guide* (00085-90142)
- Option ROM manuals:
  - *Mass Storage ROM Manual* (00085-90138)
  - *Plotter/Printer ROM Manual* (00085-90140)
  - *Matrix ROM Manual* (00085-90144)
- Interface Card manuals:
  - *HP-IB Interface Card Manual* (82937-90007)
  - *Serial Interface Card Manual* (82939-90007)
  - *GPIO Interface Card Manual* (82940-90007)
  - *BCD Interface Card Manual* (82941-90007)

1 These manuals are available as part no. 09915-87902.

2 These supplements are included in the *System Development Manual* and are available separately.

Manual Contents

HP 9915 Installation Manual — This manual takes the 9915 user from rack-mounting the computer through operational verification procedures. External features, operational specifications, warranty information, rack-mount kits, SELF TEST, and AUTOSTART are some of the subjects covered.

HP 9915 System Development Manual — Again, this manual (which you are reading) is the **main document** of the 9915 manual set. It serves as the guide for system development by describing the Program Development ROM language enhancements and by referencing the other manuals as necessary. Development of 9915 software on the HP-85 is covered, and all differences between the two computers are described.

Operator Interface Technical Supplement — This supplement provides hardware specifications and design examples of components that the designer can add to the 9915 computer system. The components include custom keyboards, CRT displays, external front panel lights and keys, and external speaker. Software examples accompany the designs shown.

Networking Technical Supplement — This supplement provides many examples of using the 9915 as a computer-network element, both as host and node elements. Program transfers (between computers) and use of system status and control lines (accessed with the Operator Interface) are the main topics of this supplement.

Tape Duplication and EPROM Programming Software Applications Pack — This pack, consisting of tape cartridge and documentation, is supplied with both the 98150A and 98150B Program Development kits. The use of the binary programs on the tape cartridge is explained in detail. The binary programs allow duplication of tape files, allow program images to be generated for downloading to other computers and PROM-programming devices, and allow specialized tape-file reading and writing.

9915 Service Manual — This manual describes assembly-level repair of the 9915 computer hardware. The main hardware-diagnosis tool is SELF TEST, which is explained in detail to help service personnel locate hardware failures.

HP-85 Owner's Manual and Programming Guide — Since this manual provides the standard language reference for the HP-85, it serves the same purpose for the the 9915. If you are not familiar with standard HP-85 BASIC, then this is the manual to consult. Fundamentals of computer operation and BASIC programming are described in this manual. Storing programs during development is described here, and is further explained in the *System Development Manual.* Storing programs in EPROMs and other devices is discussed in the *Tape Duplication and EPROM Programming Software Applications Pack.*

HP-85 I/O Programming Guide — This manual describes I/O concepts and gives the complete I/O language reference. Programming techniques for the interface cards are also provided in this manual.

Option ROM Manuals — These manuals describe the additional language provided by the option ROM, and contain many programming examples.

Interface Card Manuals — These manuals provide the hardware description of each interface card. Each manual details connecting the I/O card lines and configuring the switches. Programming techniques are described in the *I/O Programming Guide.*

Where to Begin

Since individual system designers have varying degrees of experience with HP-85 BASIC, it is necessary to establish a common point at which to begin. This documentation assumes that you are familiar with the following information; if not, consult the references given.

1. Fundamentals of HP-85 operation, including program entry and edit, simple program storage, and computer operating modes. The references are Sections 1 through 6 of the *HP-85 Owner's Manual and Programming Guide.*
2. Program storage and retrieval using a tape cartridge or disc. The reference to tape usage is Section 11 of the *HP-85 Owner's Manual and Programming Guide*, and the references to disc usage are Sections 1 through 4 of the *Mass Storage ROM Manual.*
3. Additional references to relevant sections of other manuals are noted in this manual as needed.

Suggested Development Procedure

The following steps are outlined here in order to help you visualize the procedure used when developing the solution to your problem. The steps are only suggestions to aid you in beginning your task; the procedure you use may be somewhat different. An example of using this procedure is shown in the next section.

1. List the required functions of the system.
2. Define the operating environment; list the required operator-system interactions.
3. List the system components necessary to implement the proposed functions and operating environment.
4. Consult the manuals relevant to your design. Keep notes of the design ideas that you have while reading the manuals.
5. Outline the software modules to get an estimate of the scope of the project.
6. Outline the hardware considerations in the project. Include custom hardware designs (keyboard, speaker, etc.), and I/O card connections and configurations.
7. Make an overall design plan for the system, and begin development.
8. When the system development is complete, begin the verification process. If the HP-85 was used for development, the 9915 should be substituted. Verifying that you have reliable software is a complex process, and steps 4 through 8 may have to be repeated several times.
9. After testing is complete, the final software may be stored permanently in the appropriate medium. For EPROM or tape storage, consult the *Tape Duplication and EPROM Programming Software Applications Pack.*

Typical Procedure

Let's assume that your application uses the 9915 to control an environmental-test chamber. The procedure following the system description is typical of the steps that you may take to design and develop this hypothetical system.

The computer system will be used to monitor and control the temperature and humidity inside the chamber. The heating, cooling, and humidifying elements will be controlled through parallel ports from the computer. The voltage and temperature of several key components will be monitored, and the measurements and corresponding times they are made will be stored on magnetic tape.

A keyboard will be used to enter the initial conditions before the test begins, and may be used later to modify experimental data or procedures. A remote keypad, speaker, and LED display will be mounted near the chamber to enable operators to start and stop the tests and to provide a warning of critical conditions.

The temperature of four points will be plotted on a four-color graph, and the individual plots will be selected by previewing several temperature-vs.-time plots on the CRT before the final color plots are made.

The steps necessary for the development of this hypothetical computer system might be as follows:

1. List the required functions of the system.

    a. Instrument and machine control
    b. Data logging
    c. Graphics generation
    d. Communication with test supervisor
    e. Matrix computations

2. Define the operating environment; list the required operator-system interaction.

    a. Operator inputs commands and data via the keyboard and remote keypad.
    b. CRT displays messages and graphics for operator feedback.
    c. Remote lights and speaker provide operator warnings.
    d. Plotter generates color graphics under operator control.
    e. Data is stored on tape at operator discretion.

3. List the system components necessary to implement the proposed functions and operating environment.

   a. ASCII keyboard
   b. Data-display monitor
   c. Remote keypad, speaker, and LED display
   d. 9872 4-Color Graphics Plotter
   e. 3497 Scanner
   f. 82940A GPIO Interface card
   g. 82937A HP-IB Interface card
   h. HP-85 Matrix ROM (00085-15004)

4. Consult the manuals relevant to your design. Keep notes of the design ideas that you have during this investigation.

   a. Become familiar with the *HP-85 Owner's Manual and Programming Guide*. You may be especially interested in special function keys, timers, array variables, and graphics.
   b. Read the *Matrix ROM Manual* to learn the computational capabilities of the computer equipped with this ROM.
   c. Familiarize yourself with the *I/O Programming Guide*, especially sections on binary functions, formatted-I/O operations, and HP-IB and GPIO interface programming.
   d. Read the sections of the *Operator Interface Technical Supplement* concerning keyboards, remote lights, remote speakers, and CRT-display requirements.
   e. Read the *HP-IB* and *GPIO Interface Installation* manuals. This is especialy important for the GPIO interface, as you will need to know the specifications of the lines that you will connect to the environmental-control devices.

5. Outline the software considerations to get an estimate of the scope of the project. If the software is written in modules, each new module can be merged into the main program with the use of the *Tape Duplication and EPROM Programming Software Applications Pack*.

   a. Environmental-control software (includes GPIO programming)
   b. Instrument-control software (includes HP-IB programming)
   c. Display and keyboard software for the human interface
   d. Remote speaker, keypad, and light software
   e. Graphics software
   f. Data-logging software
   g. Test-result computational software

6. Outline the hardware considerations in the project. Include custom hardware designs and I/O card connections and configurations.

    a. Design of ASCII keyboard and remote keypad
    b. Choosing (or modifying) a compatible CRT display
    c. Design of a remote speaker
    d. Design of remote lights
    e. Connection and configuration of the GPIO card to the environmental-control devices
    f. Connection and use of the HP-IB interface card

7. Make an overall design plan for the system, and begin development.

    a. You may want to include scheduling, manpower requirements, and budget in your plan.

8. When the system development is complete, begin the verification process. If the HP-85 was used for development, the 9915 should be substituted. Verifying that you have reliable software is a complex process, and steps 4 through 8 may have to be repeated several times.

    a. You may want to perform extensive software reliability tests of the human-interface software.
    b. The programs should make adequate checks to ensure that the operator is warned of important test failures and hazardous conditions.

9. After verification, the final software may be stored using the desired storage medium. For EPROM and tape storage, consult the *Tape Duplication and EPROM Programming Software Applications Pack*. For storage in other computers, consult the *Networking Technical Supplement*.

Chapter 2
Display Control

Front Panel Lights

The only display capability on the standard modular computer is front panel lights. The lights provide visual feedback to inform the 9915 user of important conditions. The front panel lights include the system status lights (RUN, SELF TEST and ON) and the eight, programmable user lights.

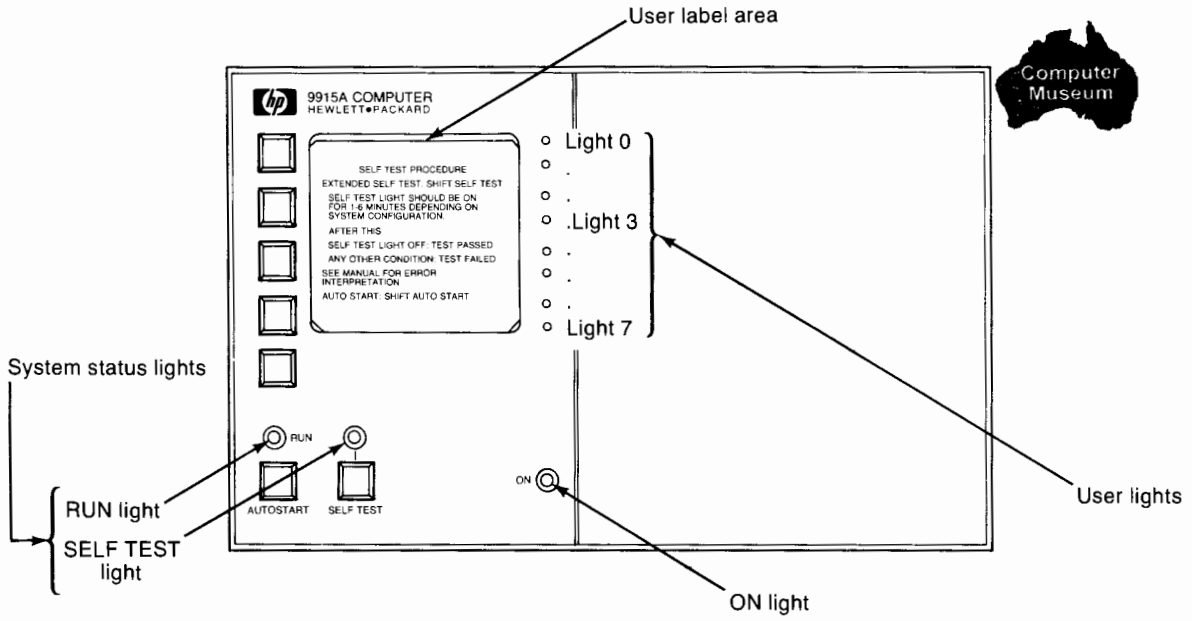


Figure 2-1. Front Panel of the 9915

The user lights can be used to annunciate any condition that you desire, but the system lights annunciate only the defined system conditions. The RUN light signifies that a program is running. The ON light signifies that computer power is on. The SELF TEST light signifies that either SELF TEST or AUTOSTART is currently testing the computer hardware.

The current definition of the user lights can be listed on a label area insert that fits into the recessed label area. The special-function-key definitions can also be listed on the removable insert. For further details, see the *HP 9915 Installation Manual.*

The user lights are controlled with the following function:

```
SLITE ( light , action )
```

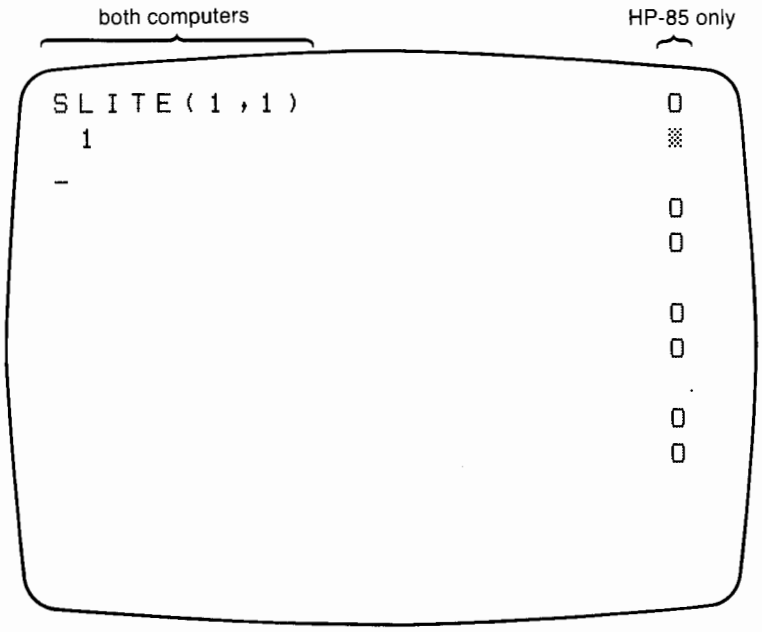
The SLITE function serves two purposes. It turns the specified light on or off and returns the resultant light status (1, -1, or 0; see following table). The lights are numbered 0 through 7 from top to bottom. Different values of action cause the following results.

Table 2-1. SLITE Parameter Definitions

Specified action	Resultant action	Status returned
any expression that rounds to 1	turns specified light on	1
any expression that rounds to 0	no change (used to read current status of light)	±1
any expression that rounds to −1	turns specified light off	−1
any other value of action	no change in any light due to improper action value (0 also returned if improper light value specified)	0

As with the values of action, the values of light can be any real number; however, the values must round to an integer in the range 0 through 7. Using values of light outside this range are reported by the function returning zero, as stated in the preceding table.

Using the function in live-keyboard mode performs the specified action and returns the status of the specified light on the following line. Executing the statement with light = 1 and action = 1 turns on light 1 and returns its status as shown below.



Chapter 2
Table of Contents

Display Control

When the function is used in a program it also returns a numerical value; therefore, it must be used with a statement (or expression) which requires a numerical value and cannot be the only expression on a line. Typical examples might be:

```
320 S=SLITE(5,0) !  read status only
480 L2=SLITE(2,1) ! turn on light #2
```

As an example, suppose that you want to define two of the eight lights to display two important conditions. Let light 0 being on (green) signify that a certain desirable condition exists. Let light 1 being on (yellow) signify that some hazardous condition exists.

The following program checks two I/O devices at 1-second intervals and displays the corresponding conditions. Light number 1 is made to blink if the condition reported by the device at address 400 is critical.

```
105 ! Example of defining user lights
106 !
110 ON TIMER#1,1000 GOSUB 200 !          Check every second.
120 .
    .
    .
195 !                                    Interrupt subroutine.
200 ENTER 722;V !                            Read voltage.
210 Z=SLITE(0,1-2*(V<108 OR V>126)) !        #0 on / off.
215 !
216 !
220 ENTER 400;B !                            Read B.
230 IF B<3.25 THEN Z=SLITE(1,-SLITE(1,0)) !  #1 toggled.
240 IF B>=3.25 AND B<5 THEN Z=SLITE(1,1) !   #1 on.
250 IF B>=5 THEN Z=SLITE(1,-1) !             #1 off.
255 !
260 RETURN
```

In this example, the conditions that the lights report are:

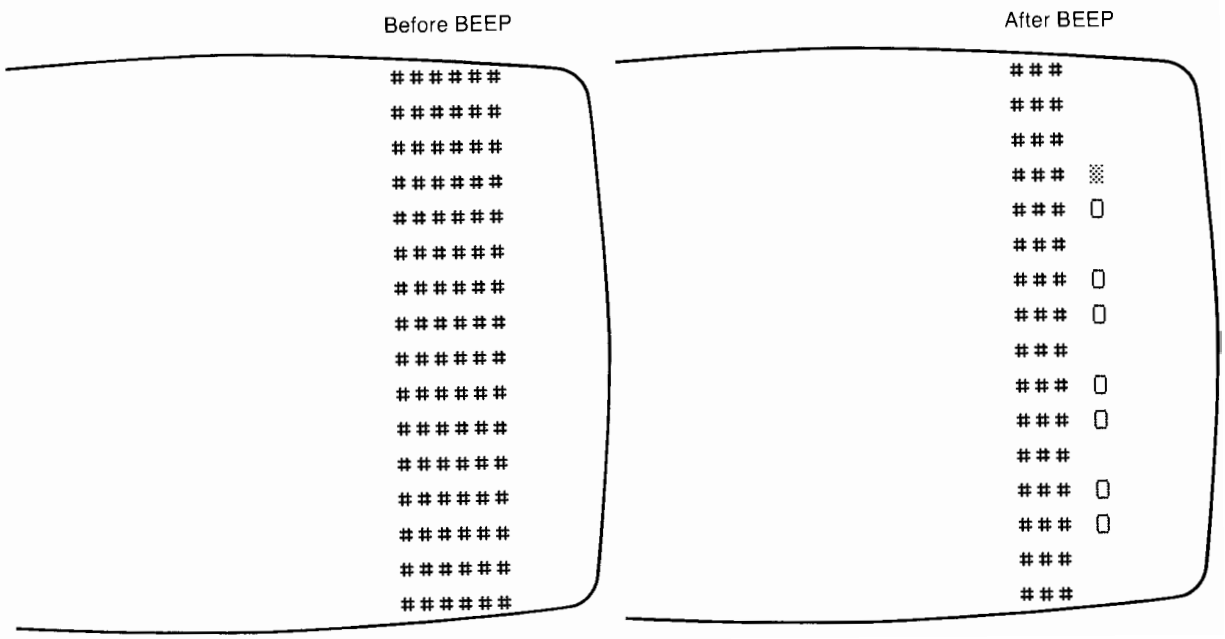
Light	State	Condition reported
0	On	108 <= V <= 126
	Off	
1	Blinking	B < 3.25
	On	3.25 <= B < 5
	Off	B >= 5

HP-85 User Lights

The HP-85 has no actual user lights; however, it does display characters on the CRT that correspond to the 9915's user lights. The result of executing SLITE on the HP-85 is that the last three columns of the display are blanked and "light" characters are displayed in the pattern of the 9915 lights (▒ signifies on, and 0 signifies off; CHR$(31) and CHR$(79), respectively).

If any characters have been previously written in these columns, they will be overwritten by the "light" characters and the surrounding blanks. The absolute address of these characters is dependent on the current display window, which is discussed in the next section. To see the effects of the overwriting tendencies of the statement, execute the following program.

```
105 ! Example of HP-85 user lights
106 !
110 DIM A$[32]
120 CLEAR
130 FOR C=1 TO 32 !              Create 32-character string.
140 IF C<=26 THEN A$[C]=" "
150 IF C>26 THEN A$[C]="#" !     Last six characters are #'s.
160 NEXT C
170 FOR L=0 TO 15 !              Fill screen.
180 DISP A$
190 NEXT L
200 WAIT 1000                    Wait one second, then
210 L0=SLITE(0,1) @ BEEP !       display lights and beep.
220 END
```



Remote Front Panel

The front panel lights and keys of the 9915 are also remotely accessible with the Operator Interace (Option 001). The statements used to control the user lights are identical, and the hardware specifications are covered in the *Operator Interface Technical Supplement.* The supplement also describes remote keys.

Enhanced CRT Display Control

The Operator Interface (Option 002) provides the video signal used to produce a display with the format of the HP-85. The hardware requirements of the CRT and the electrical specifications of the video signal are described in the *Operator Interface Technical Supplement.* If you are designing or choosing a CRT to be compatible with the 9915, you may want to read the supplement at this point.

The use of the Program Development ROM allows the program full control of the display. The program can give the display any response characteristics desired, as it can read and write characters into any location and roll the screen up and down.

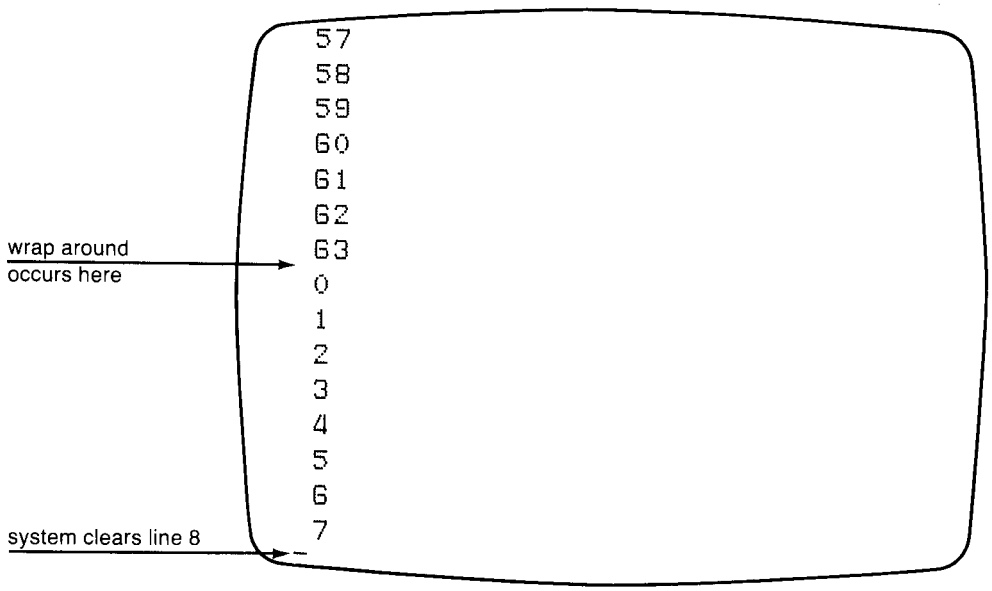
Without the PD ROM, the only CRT control on the HP-85 is with the DISP, PRINT, and OUTPUT statements[1]. The characters are written into the next available line or after the last character printed on the current line. The difference between the starting positions depends on the use of carriage control (";" or IMAGE used with DISP statement).

How the Display Works

The display memory consists of 64 lines of 32-character, alphanumeric data. Only 16 lines, known as the display window, can be viewed at one time. The contents of all 64 lines of display memory can be viewed, 16 lines at a time, by executing the following program and using the ROLL (and shift ROLL) key.

```
105 ! Example to show display organization
106 !
110 FOR I=0 TO 63+8 !  Fill all 64 lines, overwriting first
120 DISP I MOD 64      eight to show wrap-around point.
130 NEXT I
140 END
```

1 See Sections 5 and 10 of the *HP-85 Owner's Manual and Programming Guide* for further details.



As you can see, the memory is organized so that it "wraps around" from line 63 to line 0 when the window is within 15 lines of either the top or bottom of display memory. If you roll the display up a few lines after running the program, you will notice that line 8 has been cleared and the cursor is positioned in the first column. This is the result of the operating system regaining control of the display after the program has been run.

The operating system anticipated that you needed a blank line to type in subsequent commands, so it cleared the next line and positioned the cursor at the beginning of the clear line. Thus, the cursor shows where the next characters will appear as you type them in from the keyboard.

As you experiment with the display statements in this section, keep in mind that the operating system takes over control of the display after you type in commands and execute them from the keyboard.

The following drawing may help you visualize the display memory organization and character addressing used.

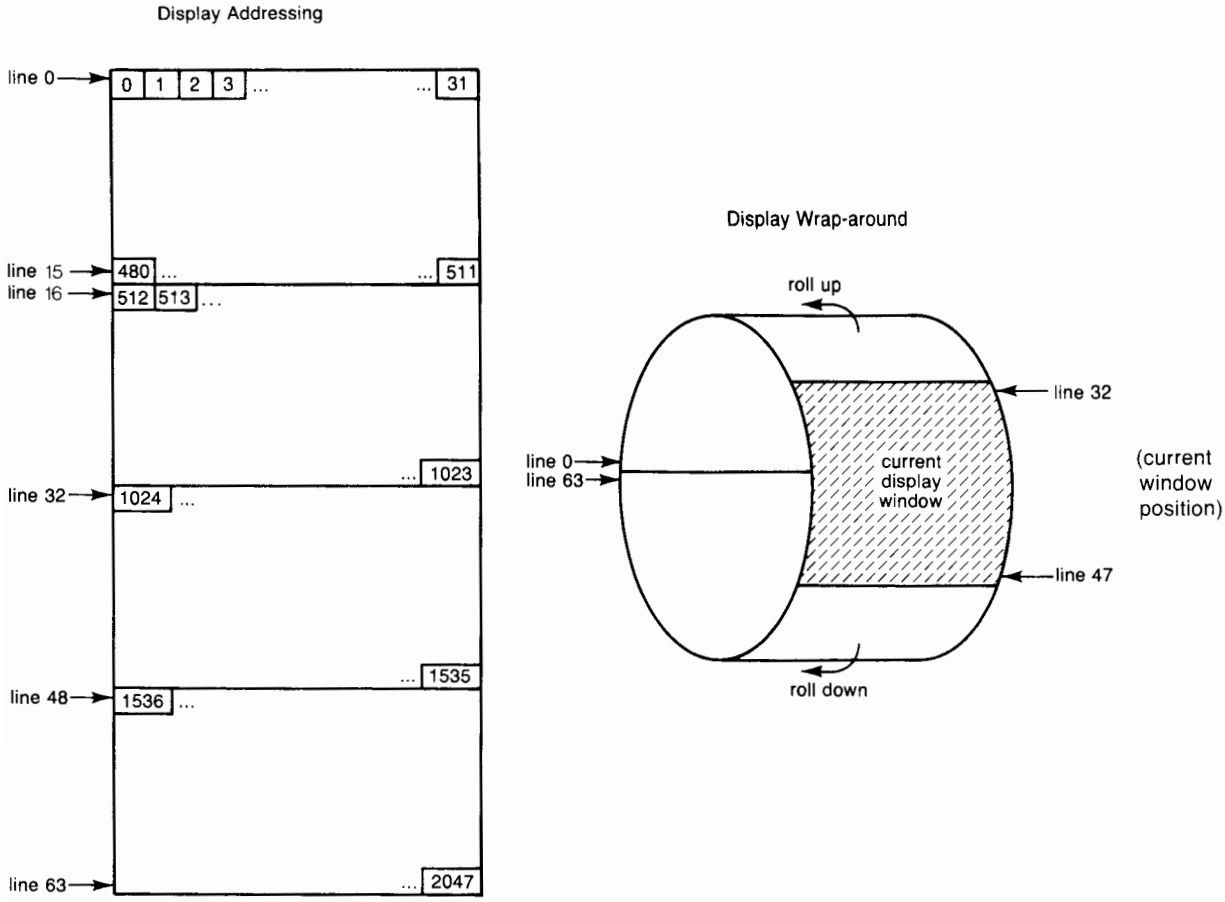


Figure 2-2. Display Memory Organization

For the preceding display, the Program Development ROM provides the abilities:

- to write to any location in display memory without affecting the surrounding characters (see "Writing to Display Memory").
- to read all characters currently in display memory (see "Reading the Display Contents").
- to display any 16 contiguous lines of display memory (see "Positioning the Display Window").
- to determine which 16 lines of display memory are currently seen on the CRT (see "Sensing the Window Position").
- to control the location of successive character entries into display memory, as determined by the location of the cursor (see "Moving the Cursor").
- to determine the location of the cursor in display memory (see "Sensing the Cursor Location").
- to execute user-defined display routines when attempting to display (or print) certain control characters (see "Displaying Control Codes" and "Printing Control Codes").

Clearing the Screen

At RESET[1], the computer blanks display memory contents. The display window is moved to the "top" of display memory (i.e., the first line displayed is line 0), and the cursor is positioned at address 0. The same display state is obtained by executing:

```
CCLEAR
```

Notice that this statement is different from the CLEAR statement. When the screen is CLEARed, the operating system rolls the display up until the line containing the cursor is at the top of the screen, and then clears the next 16 lines of memory. The display window and cursor location are not set to zero, and the entire display contents are not blanked. This action can be seen by moving the cursor to the middle of a screen full of characters and then pressing CLEAR followed by ROLL.

1 A complete table of RESET conditions is given in Appendix B.

Positioning the Display Window

The starting line of the display window is kept in computer memory. Using the ROLL key increments or decrements the value of this pointer. This function is now programmable with the statement:

CLINE line

where line is a numerical expression that selects the beginning line of the display window. Beginning line numbers range from 0 through 63.

The value of line is not restricted to the integers 0 through 63; it can be any real number within the computer's range. However, values of line less than -32767 are truncated to -32767. Similarly, values greater than 32767 are truncated to 32767. Line is then automatically mapped into the 64-line range with the expression:

new window position = INT ((line MOD 64) + 0.5)

This extended range of values allows you to use a variable without having to first ensure that it is an integer between 0 and 63. Some of the translated values are shown in the following table:

Rounded Line Value	Resultant Window Position	
.	.	
.	.	
.	.	
-32769	1	
-32768	1	
-32767	1	
-32766	2	
.	.	
.	.	
.	.	
-66	62	
-65	63	
-64	0	
-63	1	
-62	2	
.	.	
.	.	
.	.	
-2	62	
-1	63	
0	0	Rounded value = resultant window position
1	1	
2	2	
.	.	
.	.	
.	.	
63	63	
64	0	
65	1	
66	2	
.	.	
.	.	
.	.	
32766	62	
32767	63	
32768	63	
32769	63	
.	.	
.	.	
.	.	

Sensing the Window Position

The current window position (beginning line of the current display window) is obtained by calling the numerical function:

```
CLPOS
```

This function returns the window position as an integer ranging from 0 through 63.

Relative window positioning is implemented by using CLPOS with the CLINE statement. The following example rolls the display up 16 lines (or one "page") from the current window position.

```
CLINE CLPOS+16
```

The following statement rolls the display down one line.

```
CLINE CLPOS-1
```

Of course, absolute positioning is often used. The following statement positions the window at line 48.

```
L=48 @ CLINE L
```

Moving the Cursor

The cursor location determines where the next display statement begins placing characters. The cursor can be moved by changing the value of its address, which is stored in computer memory. Any subsequent display statement will begin writing characters at the new address.

The value of address is changed with the statement:

```
CCURSOR address
```

Address is a numerical expression that specifies the new cursor location. Unique display addresses range from 0 through 2047.

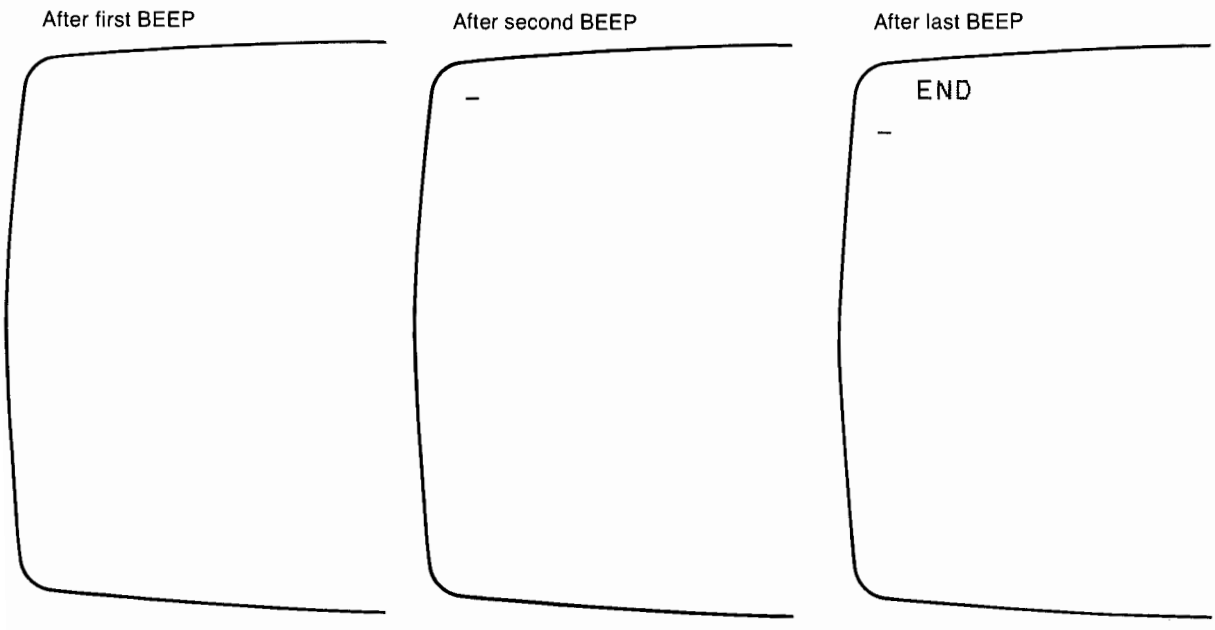
The value of address is not restricted to the integers 0 through 2047; it can be any expression within the range of the computer. However, values of address less than -32767 or greater than 32767 are truncated to these values. Address is then automatically mapped into the proper range with the expression:

$$\text{new cursor location} = \text{INT}((\text{address MOD } 2048) + 0.5)$$

As with the CLINE statement, this automatic mapping allows you to use any expression without first having to ensure that it is an integer between 0 and 2047. Thus, all values of address yield meaningful locations in display memory.

The cursor character need not always be displayed after it is moved. For instance, if the operator does not have a chance to enter data from the keyboard, he need not be shown where the next character will be displayed by the program. For this reason, the cursor is not automatically restored after it is moved with CCURSOR. It is necessary to execute a display statement after moving the cursor to restore the cursor character to the screen. The next program shows this feature of the CCURSOR statement.

```
105 ! Example of moving the cursor
106 !
110 CCLEAR
120 CCURSOR 32 @ BEEP 50,200 ! No cursor character after move.
130 WAIT 1000
135 !
140 DISP "" @ BEEP !           Underline restored with DISP
150 WAIT 1000 !                (to beginning of next line).
155 !
160 DISP "END" @ BEEP 200,50 ! Display begins at address 64.
170 END
```



Sensing the Cursor Location

The cursor location is determined by calling the numerical function:

```
CCPOS
```

The current cursor location (or address) is returned as an integer in the range 0 through 2047.

Relative cursor positioning can be implemented by using CCPOS with the CCURSOR statement. A typical use of the function is to move the cursor to the same column in the next line, which is accomplished with the following program statement.

```
100 CCURSOR CCPOS+32
```

Moving the cursor to the upper-left position of the screen (or "home") is a commonly used function which is implemented with the following program statement.

```
100 CCURSOR CLPOS*32
```

Suppose that the program needs to inform the user of some critical condition by writing into the display. If the program writes characters into the current window, it may destroy some useful information. The following program shows how the window can be moved to a portion of display memory that probably does not contain critical information. After displaying the message, the program restores the old window and then continues.

```
225 ! Example of "exchanging" display windows
226 !
230 L=CLPOS @ C=CCPOS !          Save window and cursor loc.
240 CLINE L+16 @ CLEAR !         Roll and clear window.
250 CCURSOR (CLPOS+4)*32 !       Display message on 5th line.
255 !
260 DISP "WARNING : FLUID LEVEL LOW"
270 BEEP 100,5000 @ DISP "" ! Cursor to next line.
280 DISP "Press CONT when fluid refilled"
290 PAUSE
300 CLINE L @ CCURSOR C !        Restore former display.
310 RETURN !                     Resume.
     .
     .
     .
```

The current line that the cursor is in can be determined by executing:

```
L=CCPOS DIV 32
```

Since the addresses of the beginning of all lines are integral multiples of 32, the cursor can be positioned to the beginning of the current line by executing:

```
CCURSOR (CCPOS DIV 32)*32
```

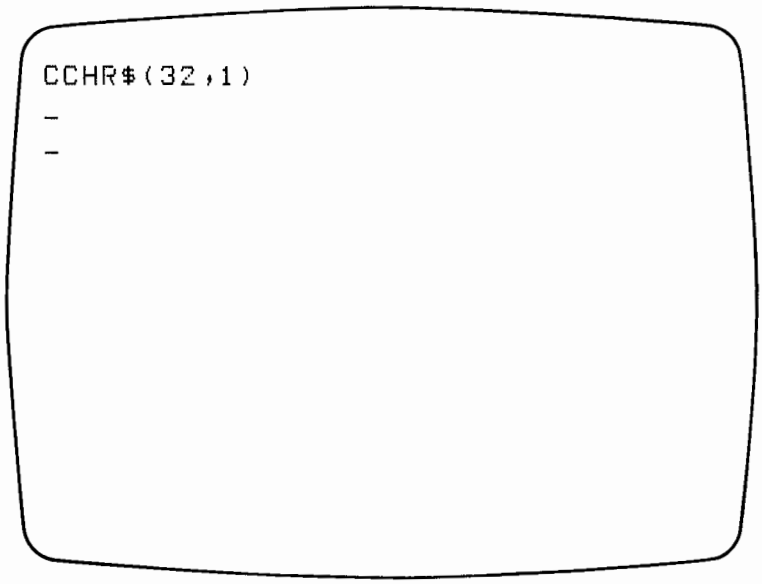
Reading the Display Contents

Display characters are read by calling the string function:

```
CCHR$ ( address , length )
```

The parameters are the starting display address of and number of characters (string length) to be returned. If the length parameter is greater than 2048, characters are re-read, beginning at address, until the length parameter is satisfied.

The display contents are returned as a string of characters. If the cursor character is in the section of the display read, it will be read as a character with a code of 128 or above[1]. When in live-keyboard mode, the operating system again may intervene, causing some unexpected results, as shown in the following example (be sure to execute a CCLEAR before typing in the CCHR$ statement):



1 All characters with codes greater than or equal to 128 (bit 7 = 1) are underlined. To underline a non-underlined character, add 128 to its code. For further details, see "Character and Key Codes" in Appendix B.

Notice that the cursor appeared in line 1 after END LINE was pressed. This was the operating system acknowledging the END LINE keystroke, **after** which it executed the requested action. The second cursor (in line 2) was the operating system acknowledging that it completed the requested action, and that further statements or commands may be entered. The first cursor remained because it was displayed as the character returned by the CCHR$ function.

Notice that the second cursor is the system cursor which erases the first cursor as the up-arrow key is pressed the second time. The overall message to be learned is that the cursor underline character is generated by adding 128 to the character code that is to be underlined. When the operating system moves the cursor, it effectively subtracts 128 from the character away from which the cursor is moved.

Displaying Characters

Two new statements provided by the Program Development ROM allow the program to write to any display-memory location, beginning at the cursor location. The similarities between the two statements are that only the specified number of characters are written into display memory; the rest of the line is not blanked as with the DISP statement. The differences lie in how they affect the display window, how they leave the cursor after writing to the display, and how control characters are interpreted.

Writing to Display Memory

The CWRITE statement allows all characters (including control characters) to be written into display memory beginning at the current cursor location. The characters written to the display must be in string form. The statement syntax is:

```
CWRITE string
```

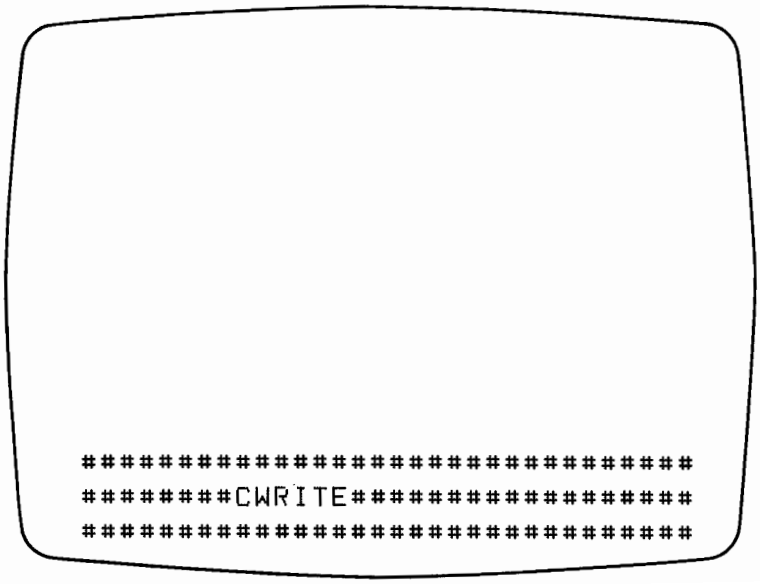
in which string is any string expression or string buffer. If string is longer than 2048 characters, the 2049th character overwrites the first, and so forth.

The most important features of this statement are that it can write all characters in the character set into display memory and that it does not necessarily display them. You may not understand all the implications of these features until they are compared to those of the CDISP statement.

"Writing characters without displaying them" implies that characters can be written into memory locations "off screen". To see this action, enter and run the following program.

```
105 ! CWRITE features
106 !
110 DIM D$[32]
120 CCLEAR
125 !
130 D$="################################" ! 32 #'s
140 DISP D$,D$,D$ !    Display "on screen".
150 BEEP 400,25 !      First beep.
160 WAIT 1000
165 !
170 CLINE 3 !          Roll "off screen".
180 CCURSOR 32+8 !     Move cursor into 2nd line.
190 CWRITE "CWRITE" ! Write 6 characters into memory.
200 BEEP 200,50 !      Second beep.
210 WAIT 1000
215 !
220 CLINE 0 !          Roll "on screen" (no cursor underline).
230 BEEP 100,100 !     Third beep.
240 WAIT 1000
245 !
250 BEEP 50,200 !      Cursor underline restored by
260 END                system at program end.
```

Final display



As the program is run, note the display at different beeps. At the first beep, the first three lines have been filled with characters. At the second beep, the #s have been rolled off screen and the characters "CWRITE" written into the off-screen locations. At the third beep, the characters written off-screen can be seen, but no cursor underline character has been displayed after the CWRITE. Soon after the fourth beep, the operating system has displayed the underline at the cursor location. Typing characters in from the keyboard will cause more of the #s to be replaced.

Two of the most important features of the CWRITE statement have been shown; namely, that the statement can write characters off screen and that the the cursor underline is not displayed at its new location. The rest of the features of CWRITE are best presented in a comparison to the CDISP statement, discussed next.

Displaying Control Codes

The second new statement used to display characters is:

```
CDISP string
```

The string parameter can be any string expression or buffer, just as with the CWRITE statement.

To see the effects of this statement, change the CWRITEs to CDISPs in line 190 of the preceding example and run the program again. After the second beep, the characters "CDISP" appear in the bottom line of the display window. In the previous example, the CWRITE occurred off screen. However, one of the features of the CDISP statement is that it cannot display characters to memory locations off screen. It automatically rolls the display so that the CDISPed characters will be in the display window. It also clears the display line into which it writes the characters to avoid interleaving the new characters with previously displayed characters. Notice also that at the second beep, the cursor underline character is displayed to show the cursor position. This is another feature of CDISP.

There is another feature of CDISP that is even more important than these first two. The CDISP statement is designed to allow you to implement your own special control-code display routines. Enter and run the following program.

```
105 ! CDISP interprets control codes
106 !
110 DIM D$[32]
115 !
120 FOR C=0 TO 31 !              Fill string with
130 D$[C+1]=CHR$(C) !            control characters.
140 NEXT C
145 !
150 DISP "first CWRITE"
160 CWRITE D$ !                  Display control characters.
170 WAIT 1000
175 !
180 DISP @ DISP "then CDISP"
190 CDISP D$ !                   Now take defined action
200 WAIT 1000 !                  (NUL, Beep, BS, LF, FF, & CR).
205 !
210 CLINE 0 @ END !              Reposition window.
```

All of the control characters are displayed with CWRITE, but none is displayed with CDISP. Instead, the control characters either cause specific display action or are ignored (not displayed). The following control characters always cause the defined action when used with the CDISP statement.

Final Display

Table 2-2. Control Codes Defined in CDISP

Control Code	Name	Display Action
0	Null	Ignored by the display.
7	Bell	Short BEEP.
8	Backspace	Move the cursor back one position, but never off the display window.
10	Linefeed	Move the cursor down to the same position in the next line; if it is off the window, clear the line and roll the display until the cursor is in the bottom line.
12	Formfeed	Move cursor to the beginning of the next line and roll until that line is at the top of the window; clear all 16 lines in the window.
13	Carriage Return	Return the cursor to the beginning of the line it is in.

The remaining 26 control characters do not cause any action unless an ON CCODE statement is in effect. ON CCODE is described next.

Control-Code Branching

The display response when control characters (codes 0 through 31) are displayed is usually left up to the hardware designer of a system. However, now you can define the remaining 26 control characters not defined in the CDISP statement to cause any action that you wish. This section shows how to implement your own special control-code display sequences.

The CDISP statement evaluates each character before it writes it into display memory in the following manner:

1. If the character is not a control code, it is displayed. If the character location is off screen, the line onto which the character will be displayed becomes the bottom line and is cleared before the character is displayed on it. The next string character is checked.
2. If the character is one of the defined control codes, the appropriate display response is executed and the next string character is checked.
3. If the character is not a defined control code, the computer checks to see if an ON CCODE branch is active. If one is active, the CDISP is aborted and the EOL branch set up by the ON CCODE statement is taken. If not, the character is ignored and the next string character is checked.

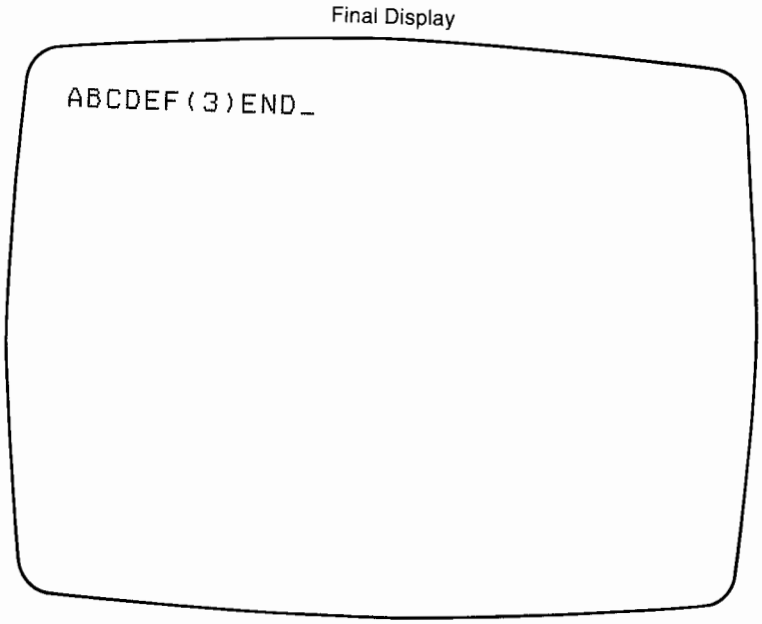
This control-code branch is set up by executing one of the following statements.

```
ON CCODE GOSUB program line
ON CCODE GOTO  program line
```

The statement does two things; it declares the specified line number as the branch destination, and it enables the interrupt. Upon attempting to CDISP any character with a code less than or equal to 31 (and not in the above set), the CDISP statement is aborted and the branch is taken after any other statements on the line are executed.

The line to which the program branches should be the first line of the routine that defines the special display action. The routine can be as long as you wish, but since the speed of its execution can affect the way the display looks to the user, it may be best to minimize the routine. The following is a simple example.

```
105 ! Use of CDISP with ON CCODE
106 !
110 CCLEAR
120 ON CCODE GOSUB 170
130 D$="ABC"&CHR$(7)&"DEF" ! Define string with control codes.
140 D$=D$&"CHR$(3)&"GHI"
145 !
150 CDISP D$
155 !
160 DISP "END" @ END
165 !
170 FOR P=1 TO LEN(D$) !     ON CCODE subroutine
180 C=NUM(D$[P,P]) !         Check each code.
190 IF C<32 THEN GOTO 210 !  Found control code.
200 NEXT P
205 !
210 ON C+1 GOTO 220,230,250,270,...
215 !
220 GOTO 200 !               Skip defined codes.
225 !
230 CDISP "(1)" !            Code 1 routine.
240 RETURN
245 !
250 CDISP "(2)" !            Code 2 routine.
260 RETURN
265 !
270 CDISP "(3)" !            Code 3 routine.
280 RETURN
    .
    .
    .
```



As you can hear, the defined action for code 7 was taken (BEEP). When the CDISP attempted to display control code 3, the branch was taken and CHR$(3) was not displayed. The remainder of D$ was not displayed because CDISP was aborted at the point it found the code. If the remainder of D$ is to be displayed, the CDISP statement must be re-executed after removing the already displayed characters and the control code that caused the branch from the string.

The process of displaying from a buffer is a similar, but more powerful, method of implementing control codes in the display. It is discussed in "Displaying Buffers".

Suspending Control-Code Branching

The interpretation of control codes by the CDISP (and CPRINT) statements, not including the defined codes, can be stopped by executing the statement:

```
OFF CCODE
```

The result is to disable the branch. CDISP (and CPRINT) ignore all undefined control codes and do not abort; however, they still take the correct action when defined control codes are displayed (or printed).

To re-enable the branch all you need to do is execute another ON CCODE statement. The subsequent ON CCODE statement need not be the same one originally executed. Executing each additional ON CCODE statement cancels any previous ON CCODE statement, and no OFF CCODE statement need be executed in between.

Printing Control Codes

Another statement has been provided to allow user-defined control-code subroutines when using the HP-85 internal printer. The statement is:

```
CPRINT string
```

The statement resembles the CDISP statement in its action, but the characters are sent to the internal printer of the HP-85. The statement causes no action when executed on the 9915. The control codes that cause specific action when CPRINTed are:

Table 2-3. Control Codes Defined in CPRINT

Control Code	Name	Printer Action
0	Null	Ignored by the printer.
7	Bell	Short BEEP.
8	Backspace	Print the next character at the previous character position; no action was taken if the previous character was in the first column.
10	Linefeed	Advance one line.
12	Formfeed	Advance five lines.
13	Carriage Return	Print next character at first column of this line.

As with the CDISP statement, CPRINT will cause a branch if the character printed is a control code not in the above list and the ON CCODE statement has enabled the branch to occur.

The string to be printed can be a buffer, as with the CDISP and CWRITE statements. The use of buffers is discussed in the next section.

Displaying with Buffers

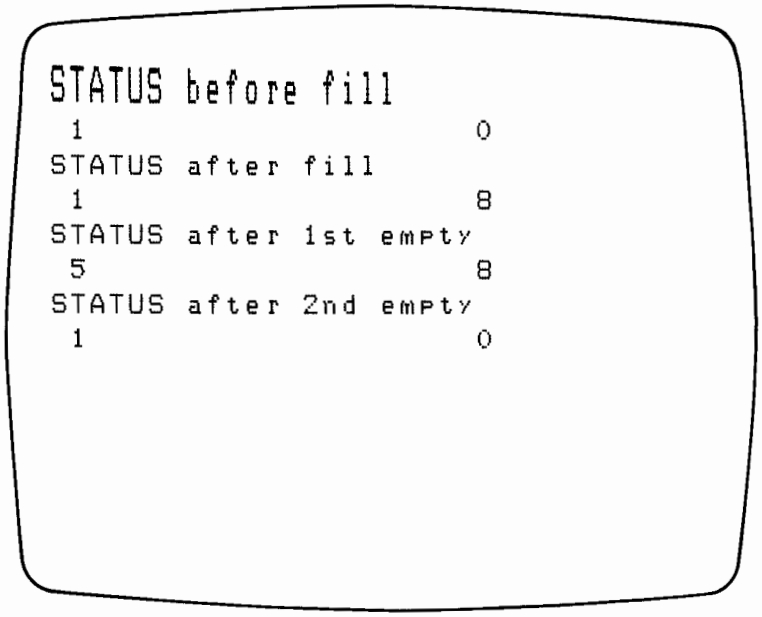
One of the most powerful uses of the CWRITE, CDISP and CPRINT statements involves the use of buffers. If you are not familiar with using buffers, this section contains a short review of the basics. For further information, see Section 8 of the *HP-85 I/O Programming Guide.*

Review of Buffer Pointers

The most important aspect of using buffers is the manner in which they are filled and emptied. The buffer is filled by adding a character to the string location specified by the fill pointer, after which the pointer is automatically incremented. The buffer is emptied in the reverse order; a character is read from the location specified by the empty pointer, and the pointer is incremented. These processes are shown below.

```
105 ! Example of buffer-pointer updating
106 !
110 DIM Z$[512]
120 IOBUFFER Z$ !                          Set up string as buffer.
125 !
130 DISP "STATUS before fill"
140 STATUS Z$,0 ; E,F @ DISP E,F !     Read/display pointers.
150 DISP
155 !
160 OUTPUT USING "#,8A" ; "12345678" ! Add 8 characters.
165 !
170 DISP "STATUS after fill"
180 STATUS Z$,0 ; E,F @ DISP E,F !     Read/display pointers.
190 DISP
195 !
200 ENTER Z$ USING "#,#4A" ; A$ !      Remove 4 characters.
205 !
210 DISP "STATUS after 1st empty"
220 STATUS Z$,0 ; E,F @ DISP E,F !     Read/display pointers.
230 DISP
235 !
240 ENTER USING "#,#4A" ; A$ !         Read last 4 characters.
245 !
250 DISP "STATUS after 2nd empty"
260 STATUS Z$,0 ; E,F @ DISP E,F !     Read/display pointers.
265 !
270 END
```

As you can see, the empty and fill pointers relieve the program of keeping track of filling and emptying the buffer. Also note what happens when the buffer becomes empty—the pointers are reset to their original values.



Displaying Buffers

As mentioned before, when an ON CCODE statement is in effect and a control code is found by CDISP (or CPRINT), program control is transferred to the location specified in the ON CCODE statement. This service routine must then determine which control character caused the branch. This is made much easier by using a buffer as the source of characters and examining the buffer pointers within the service routine. The routine must determine which character caused the branch, take the specified action, and re-execute the interrupted statement at the character following the interrupt-causing control character.

The following example illustrates how the program determines which character caused the branch and then how it takes the programmed action. Notice that the empty pointer is advanced after the branch is taken to avoid beginning the CDISP at the interrupt-causing control character.

```
105 ! Example of CDISP of a buffer
106 !
110 DIM Z$[520]                    Usable size is 512 characters.
120 ON CCODE GOTO 1010
130 IOBUFFER Z$
135 !
136 ! initialize serial I/O card
    .
    .
    .
300 TRANSFER 10 TO Z$ INTR !      Enter from serial I/O card.
310 CDISP Z$ !                    This statement re-executed.
    .
    .
    .
1005 !                            ON CCODE subroutine
1006 !
1010 STATUS Z$,0 ; E !            Read empty pointer,
1020 C=NUM(Z$[E,E]) !             then read control code.
1030 CONTROL Z$,0 ; E+1 !         Advance empty pointer.
1035 !
1036 !                            N-way branch on code.
1040 IF C>15 THEN GOTO 1060
1050 ON C GOTO 1100,1200,1300,...,2400,2500
1060 ON C-15 GOTO 2600,2700,2800,...,4000,4100
1095 !
1096 ! Routine for code 1 (=SOH)
    .
    .
    .
1190 GOTO 310
1195 ! Routine for code 2 (=STX)
    .
    .
    .
1290 GOTO 310
    .
    .
    .
4095 ! Routine for code 31 (=US)
    .
    .
    .
4190 GOTO 310
```

Notice that the TRANSFER...INTR statement sets up the interrupt mode of entering characters from the card. This allows the program to continue with its duties while the transfer takes place. It is also important to notice that the ON CCODE enables a GOTO instead of the GOSUB shown earlier. The GOTOs at the end of each control-code routine cause branches back to the CDISP statement, which then begins at the proper location.

It is important to note that if the buffer specifier is qualified in any way, the buffer-pointer updating feature is suspended, and the buffer is treated as an ordinary string. Examples of qualifying the buffer specifier are as follows.

```
100 CDISP Z$&"" !          Buffer-pointer updating is
100 CPRINT Z$[1] !         suspended by qualifying the
100 CWRITE Z$&K$ !         buffer specifier.
```

Chapter 3
Table of Contents

Keyboard Control

Chapter 3
Keyboard Control

Introduction

The front panel keys of the HP 9915 Modular Computer consist of the system keys (SELF TEST and AUTOSTART) and the eight special function keys (four function keys and one shift key). This chapter describes the statements, provided by the Program Development ROM, used to control HP-85 and 9915 keyboards.

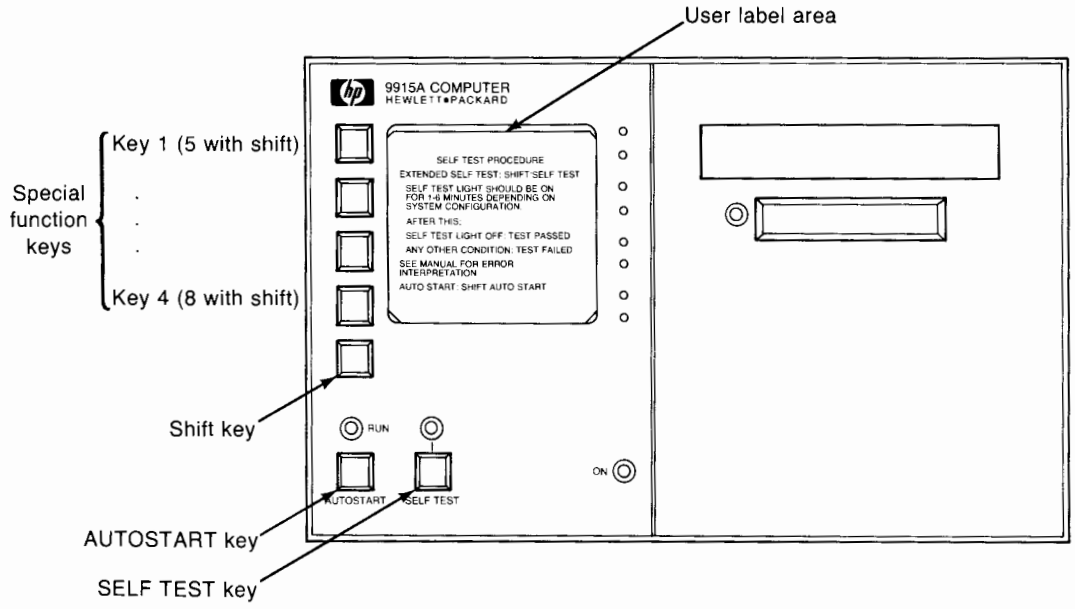


Figure 3-1. HP 9915 Front Panel Keys

Adding a more complete keyboard requires that your computer be equipped with the Operator Interface (Option 002). Compatible keyboards include the 98155A Accessory Keyboard and custom keyboards that you may design according to the specifications given in the *Operator Interface Technical Supplement.*

If you do not have a keyboard at this point, or if your system design includes a custom keyboard, consult the *Operator Interface Technical Supplement.* The supplement describes the hardware design of keyboards and gives a brief overview of the related programming techniques.

HP 9915 System Key Lockout

Even though the 9915 system keys are not programmable, their use can be enabled and disabled by the program with the statement:

```
ENABLE AS-ST mask
```

This statement resembles the ENABLE KBD statement provided by the HP-85 I/O ROM[1]. The value of mask determines whether these two keys are enabled or disabled. The mask values and their effects on the 9915 system keys are listed in the following table.

Table 3-1. ENABLE AS-ST Mask Values

Mask Value	Result
Any expression that rounds to 1	Enable both keys
Any expression that rounds to 0	Disable both keys

The keys are automatically enabled at power-on and AUTOSTART. The following program shows a typical use of the statement.

```
105 ! Example of disabling 9915 system keys
106 !
110 DISP "SELF TEST & AUTOSTART"
120 DISP " to remain enabled?"
130 DISP
140 DISP "CONT => no; Y => yes "
145 !
150 ON TIMER# 1,10000 GOTO 180 !  Allow 10 seconds.
160 M1=0 @ INPUT A$ !             Set default mask.
170 IF A$[1,1]="Y" THEN M1=1      Response "Y" enables.
180 OFF TIMER# 1
190 ENABLE AS-ST M1 !             Enable/disable keys.
200 .
    .
    .
```

Executing ENABLE AS-ST on the HP-85 will result in no action, since these keys are not implemented.

1 See Section 10 of the *HP-85 I/O Programming Guide.*

Special Function Keys

The 9915 special function keys are functionally identical to those of the HP-85 and the 98155A Accessory keyboards. These keys can be defined with statements such as:

```
100 ON KEY# 5,"ABORT" GOTO 450
125 ON KEY# 3 GOSUB 670
```

Pressing any special function key which has been defined in the program will cause the branch to the specified line after all statements on the current line. For further reference, see "Branching Using Special Function Keys" in Section 9 of the *HP-85 Owner's Manual and Programming Guide.*

Special function key definitions may be cancelled by executing the following statement for each key to be cancelled.

```
OFF KEY# key number
```

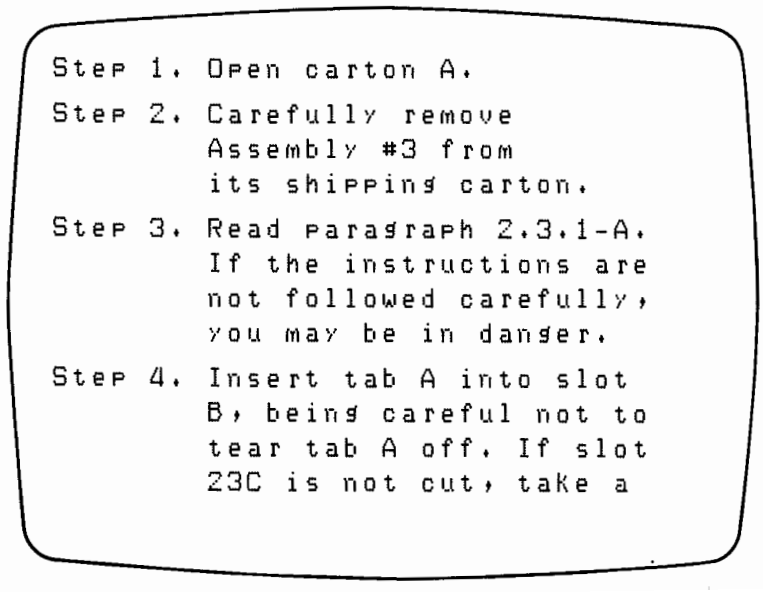
The key definitions can be changed by executing another ON KEY statement that specifies a different line number for the same key number. OFF KEY need not be executed before the re-defining statement is executed. The conditions under which the key definitions are cleared are when the program is SCRATCHed or changed, at AUTOSTART, and at power on.

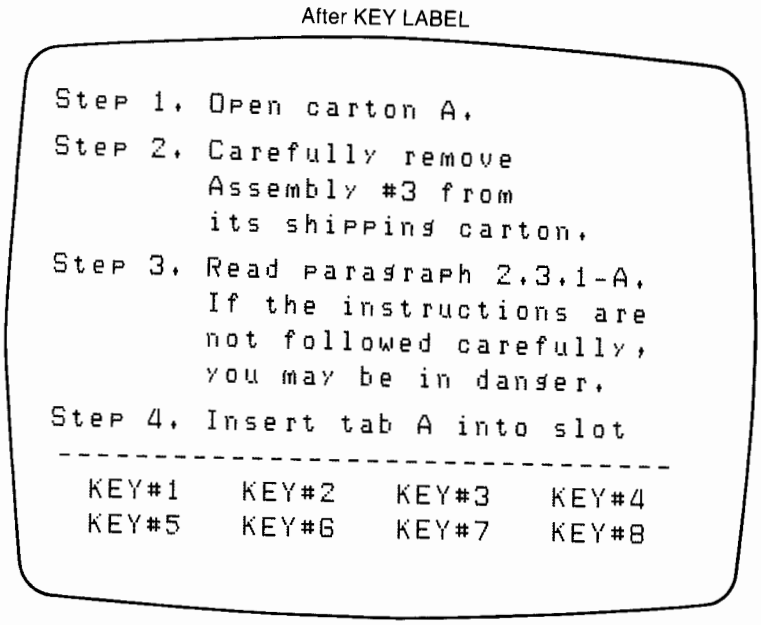
All key labels specified in the ON KEY statements can be displayed by executing:

```
KEY LABEL
```

The bottom three lines of the display window are used when this statement is executed (or key is pressed). These bottom lines are overwritten by the specified key definitions as shown below.

Before KEY LABEL





Enhanced Keyboard Control

The keyboard hardware continuously scans the keyboard looking for keystrokes. When one is detected, its code is calculated and the keyboard hardware signals that it has a valid code by setting an output line true. On the standard HP-85, this output signal is continuously monitored by the operating system. Thus, the operating system may be interrupted at any time by the keyboard[1]. There is no way to disable these interrupts from reaching the operating system on the standard HP-85.

The operating system responses to various keys during program execution are as follows[2]:

1. If the key pressed is a special function key defined in the program, the specified branch will be taken at the end of execution of the current line. If the special function key has not been been defined in the program, it will be ignored as program execution continues[3].
2. If the key is a program, system control or editing key, the program is paused and the defined action is taken. The program may or may not be automatically continued. If not, program execution may be resumed by pressing the CONT key.
3. If any other key is pressed, program execution is paused and the corresponding character is displayed.

The I/O ROM allows keyboard-lockout control with the ENABLE KBD statement[4]. Groups of keys can be selectively ignored by specifying an appropriate mask value in the statement. The operating system still gets interrupted by the keyboard hardware, but the disabled keys are ignored.

1 See "Some Background on Interrupts" in Section 9 of the *HP-85 I/O Programming Guide.*

2 See "Key Response During Program Execution" in Appendix B.

3 See "Branching Using Special Function Keys" in Section 9 of the *HP-85 Owner's Manual and Programming Guide.*

4 See Section 10 of the *HP-85 I/O Programming Guide.*

The Program Development ROM gives the program even more control of the keyboard by allowing:

- the service of keyboard interrupts to be assigned either to the program or to the operating system at any time during program execution (see "Controlling Keyboard Interrupts" and "Returning Control to the System").
- the 8-bit codes of the keys pressed to be stored as string data in an 80-character buffer which the program can read within the service routine (see "Controlling Keyboard Interrupts").
- the key buffer, or any other string, to be searched for the presence of system key codes (see "Finding System Keys").
- each key to produce any desired 8-bit code (see "Converting Key Codes").
- keyboard interrupts to be disregarded (see "Controlling Keyboard Interrupts" and "Use of KEY DOWN with ENABLE KBD").
- interrogation of the keyboard in real time to detect the presence and duration of any key press (see "Monitoring Keys in Real Time").

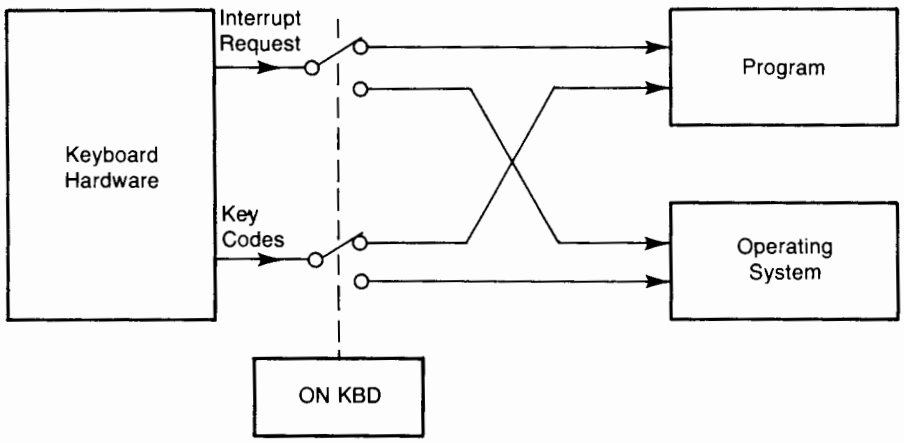


Figure 3-2. Directing Keyboard Interrupts to the Program

Controlling Keyboard Interrupts

Keyboard interrupts are directed to the program by executing statements such as:

```
100 ON KBD GOSUB 300
100 ON KBD GOTO 550
```

Execution of this statement establishes the location of the keyboard interrupt service routine, and then enables the EOL branch to be taken when the interrupt event occurs. The statement also directs the codes[1] of subsequent keys pressed to storage in an 80-character string buffer, hereafter referred to as the key buffer.

1 See "Character and Key Codes" in Appendix B for a list of the codes that each key produces.

The interrupting event is the first keystroke, at which the key buffer becomes "non-empty". Program control is transferred to the keyboard service routine, which will read (and automatically clear) the key buffer with the following string function:

KBD$

The service routine may choose either to interpret the keys or to ignore them entirely. When key processing is finished, control may be returned to the main program. If further keyboard interrupts are to be enabled, the ON KBD statement must be executed again. The following program shows a simple use of the KBD$ function in conjunction with the ON KBD statement.

```
105 ! Example of a keyboard service routine
106 !
110 ON KBD GOSUB 160
115 !
120 FOR I=1 TO 10 !                        Dummy main program.
130 WAIT 1000 @ DISP CHR$(13)
140 NEXT I
150 END
155 !                                       ON KBD service routine.
156 !
160 A$=KBD$ !                              Read/clear key buffer
165 !                                       and store in A$.
166 !
170 IF A$[1]=CHR$(139) THEN 150 !          END if RESET key.
180 DISP "Key buffer contents"
190 DISP "at EOL branch= ";A$ !            Display key codes
195 !                                       as character string.
200 DISP " "
210 DISP "Collecting further" !            Wait for more keys.
220 DISP "keystrokes for 3 sec."
230 WAIT 3000
240 DISP " "
250 DISP "New keystrokes= ";KBD$ !         Display new keys.
260 DISP " "
265 !                                       Re-enable EOL branch and
270 ON KBD GOSUB 160 @ RETURN !            RETURN on same line.
```

The example contains many subtle features of the statements used. When the interrupting event (the first keystroke) occurs during the program, execution of the current line is finished before the branch to line 160 is taken. This accounts for the delay between the first keystroke and the EOL branch being taken (up to approximately 1 second if a key is pressed while line 130 is executing).

This keyboard service request is automatically disabled from further interrupting the program when the branch is taken, even though the interrupt cause (the non-empty key buffer) may still be present. This disabling is necessary to ensure that the service routine will not be interrupted by the same request that called it. Acknowledging the service request does not stop subsequent key codes from being placed into the key buffer.

The function KBD$ reads the buffer, returns its contents, and then clears the buffer. Notice that the function was used in two different ways in the example. On line 160 it was used to read the buffer and transfer its contents into the string variable A$. This storage was necessary because the key codes were used more than once (remember that the buffer was cleared after being read). On line 250 the buffer contents were not stored because they were only used once and were not needed any further.

If the keyboard interrupt is to be re-enabled, the appropriate ON KBD statement must be executed again, as in line 270. If the service routine is a subroutine, the ON KBD and RETURN statements must be on the same program line. This technique is necessary to avoid the possibility of enabling further interrupts before this interrupt service routine has completed. The error generated when this technique is neglected is Error 18. It is due to the execution of more GOSUBs, without corresponding RETURNs, than the operating system allows.

Returning Keyboard Control to the System

If you want to return control of the keyboard to the system, all you need do is execute the statement:

```
OFF KBD
```

Further keyboard interrupt requests are directed back to the operating system until another ON KBD statement is executed. Since this statement also clears the key buffer, you may want to store its contents before executing the OFF KBD statement. The ON KBD branch is also disabled during INPUT statements, and at RESET, PAUSE, STOP, and AUTOSTART.

If another ON KBD statement is executed at a later time and a key is pressed before the line has been completely executed, the branch is taken at the end of that line.

Keyboard Buffer Overflow

Since the key buffer can only store 80 key codes, pressing more than 80 keys between calls to the string function KBD$ causes the key buffer to overflow. When the 81st keystroke occurs, control of keyboard interrupts is returned to the operating system. Error 118 is reported unless an ON ERROR statement is in effect[1]. If ON ERROR is in effect, the specified branch will be taken, and calls to the numerical functions ERRN and ERROM will return the values 118 and 8, respectively.

In either case, the key buffer retains only the first 80 key codes, which can be obtained by calling KBD$.

Finding System Keys

System keys are stored in the key buffer as codes ranging from 128 through 173[2]. A typical duty of the keyboard service routine is to search the key buffer for system keys. If you had to write this routine, it would probably look much like the following subroutine.

```
175 ! Example of searching for system keys
176 !
180 S=0 !                                  Return 0 if no key found.
190 Z$=KBD$ @ L=LEN(Z$) !                  Store keys @ # of keys.
200 P=1 !                                  Start search at Z$[1].
205 !
210 IF NUM(Z$[P,P])>=128 THEN S=P ! Keep system key position.
215 !
220 IF S>0 THEN GOTO 240 !                 Done if system key found,
230 P=P+1 @ IF P<=L THEN GOTO 210 ! ELSE next key code.
235 !
240 RETURN
```

Calling the numerical function FIND will yield the same result.

```
FIND ( string )
```

The string parameter can be any string expression or buffer. The numerical value returned by the function is the location in the specified string of the first key code greater than or equal to 128. If no such code is found, zero is returned. The function also finds system keys in buffers the same way as in strings, and the buffer pointers are totally ignored.

1 See "Error Testing and Recovery" in Section 13 of the *HP-85 Owner's Manual and Programming Guide.*

2 See "Key Response During Program Execution" in Appendix B.

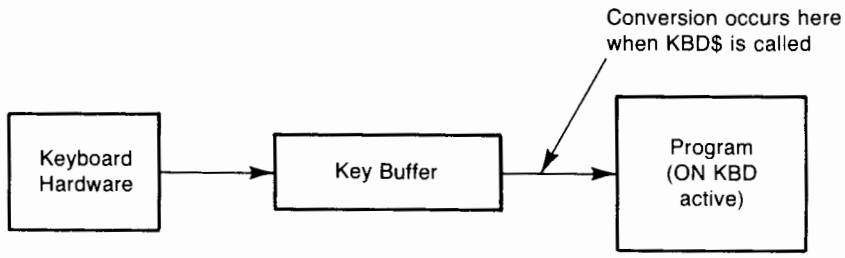
Converting Key Codes

Since you may want to redefine the codes produced by certain keys, a statement to do this is provided by the PD ROM.

```
100 KBD CONVERT INDEX ; K$
100 KBD CONVERT PAIRS ; X$
```

This statement very closely resembles the CONVERT statement provided by the I/O ROM except for two differences[1]. First, the IOBUFFER statement is not necessary, because the source is always the key buffer. Second, the direction is not specified because the keyboard is an input-only device. The conversion of the key codes is very powerful, as you will see if you are designing your own unique keyboard.

The conversion is made at the time that the key buffer is read. Thus, if a KBD CONVERT statement is in effect at the time the keystrokes are made but is not in effect at the time the buffer is read, no conversion is made. The following diagram may give you more insight into the process of key code conversions. A complete example of converting key codes is given in the *Operator Interface Technical Supplement.*



Interactions Between ON KBD and Other Statements

Certain statements disable the EOL branch of the ON KBD statement when they are executed. The INPUT statement temporarily returns control of the keyboard to the operating system[2], so that the user can enter the requested data. Satisfying the INPUT requirement(s) returns control of keyboard interrupts to the program.

Executing the ENABLE KBD statement with any mask[3] parameter while ON KBD is in effect cancels the ON KBD branch, but this time the operating system is not able to set up the ON KBD branch again.

NOTE

The ENABLE KBD statement must not be used in the same program that uses any ON KBD statements, due to the permanently disabling effect of the ENABLE KBD statement on the ON KBD branch.

---

1 See "Converting I/O Data" in Section 3 and "The Care and Feeding of Buffers" in Section 8 of the *HP-85 I/O Programming Guide.*

2 See "Key Response During Program Execution" in Appendix B.

3 See Section 10 of the *HP-85 I/O Programming Guide.* The mask definitions are also shown in Appendix B of this manual.

Precedence of ON KBD Branching

Since many devices can interrupt the program at one time, a precedence of servicing the interrupts needs to be kept. The keyboard interrupt has second priority in being processed, as shown in the following table. This table is also shown in Appendix B for your convenience:

Table 3-1. Branch Precedence Table

Branch Type	Select Code							
	3	4	5	6	7	8	9	10
ON ERROR	----------1----------							
ON KBD	----------2----------							
ON CCODE	----------3----------							
ON INTR	4	5	6	7	8	9	10	11
ON TIMEOUT	12	13	14	15	16	17	18	19
ON EOT	20	21	22	23	24	25	26	27
ON KEY	----------28----------							
ON TIMER (timer)	29 (#1)			30 (#2)			31 (#3)	

Monitoring Keys in Real Time

The keyboard hardware continuously scans the keyboard searching for keystrokes. This hardware can be interrogated at any time by the program to determine which key, if any, is currently being held down by calling the numerical function:

```
KEY DOWN
```

This function returns the following integers:

- −1 if no key is currently pressed (or is "down").
- the code of the key currently pressed.

This function operates independently; it does not interact with key codes being stored in the key buffer or any KBD CONVERT or KBD ENABLE statement in effect. If any key conversion is necessary, you must write the software to do so.

To use the KEY DOWN function in a program, the keyboard interrupt requests must either be redirected to the program or be ignored by the operating system. Thus, either ON KBD or ENABLE KBD is normally used with the KEY DOWN function.

Use of KEY DOWN with ENABLE KBD

An elementary use of the KEY DOWN function is shown below.

```
105 ! Example of KEY DOWN with ENABLE KBD
106 !
110 ON TIMER#1,10000 GOTO 150 !              END in 10 seconds.
115 !
120 ENABLE KBD 128 !                        Enable only RESET in
125 !                                       program exec. mode.
126 !
130 K=KEY DOWN @ CDISP CHR$((K<0)*33+K) ! Display blank or
140 GOTO 130 !                              Key-code character.
145 !
150 DISP @ DISP "END" @ BEEP
160 END
```

The keyboard has been disabled (except for the RESET key) so that keystrokes will not halt the running program. The program displays the character corresponding to the key code if any key is pressed at the time the function KEY DOWN is called. The process is repeated until the count of timer #1 reaches 10 seconds.

It is important to note that the KEY DOWN function has a built-in switch debounce mechanism[1]. For this reason, calling KEY DOWN while any key is pressed takes longer than calling the function when no key is pressed. Thus, the blanks are displayed in the above program more rapidly when no key is down than when the space bar is held down.

The main reason that you should be aware of this feature of the KEY DOWN function is that, since it checks the keyboard at discrete points in time, it may not be called often enough to detect all keystrokes that occur. The fact that calls to the function require less time when no key is pressed allows you greater assurance that you can monitor the keys without missing any keystrokes. Use of this tool in monitoring the keys should be made with the understanding of the possibility of missing keystrokes. However, since the response to the key is proportional to the duration of the keypress, it probably will not matter that the user may have to hold the key down a little longer waiting for the programmed response. This is the nature of analog keys.

1 See "Keyboard Hardware Operation" in Chapter 1 of the *Operator Interface Technical Supplement* for further details.

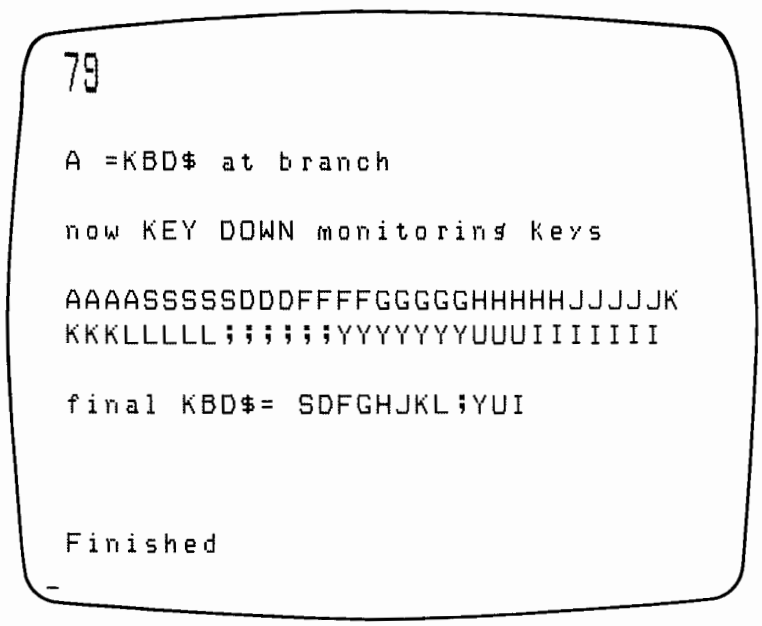
Use of KEY DOWN with ON KBD

The KEY DOWN function can also be used when an ON KBD branch is in effect, as shown in the following example.

```
105 ! Example of KEY DOWN with ON KBD
106 !
110 CCLEAR @ U=0 !                              U is a dummy counter.
120 ON TIMER# 1,5000 GOTO 160 !                 Start main timer.
130 ON KBD GOSUB 180
135 !
140 CCURSOR 0 @ CDISP VAL$(U) !                 Main loop.
150 U=U+1 @ GOTO 140
155 !
160 CCURSOR 448 @ DISP " Finished" !            Timer # 1 interrupt.
170 BEEP @ END
175 !
180 OFF TIMER# 1 !                              ON KBD routine.
190 ON TIMER# 2,5000 GOTO 270 !                 Stop main timer.
195 !                                           Start subr. timer.
200 CCURSOR 64
210 CDISP KBD$&" =KBD$ at branch"
215 !
220 CCURSOR 128
230 CDISP "now KEY DOWN monitoring keys"
235 !
240 CCURSOR (CCPOS DIV 32)*32+64 !              Skip one line.
250 K=KEY DOWN @ IF K<0 THEN GOTO 250 !         KEY DOWN loop.
260 CDISP CHR$(K) @ GOTO 250
265 !
270 CCURSOR (CCPOS DIV 32)*32+64  !             Timer# 2 interrupt.
280 CDISP "Final KBD$= "&KBD$
290 OFF TIMER# 2
300 ON TIMER # 1,2000 GOTO 160
310 RETURN !                                    With no ON KBD re-enable.
```

Type in the program, run it, and continue to type. Your final display might look like the following display.

Typical Display



Notice, after running the above program, that there were more characters displayed when KEY DOWN was monitoring the keyboard than when the final value of KBD$ was displayed. This shows that key codes are entered into the key buffer only when the keys are first pressed ("edge-triggered" buffer storage). Again, calling KEY DOWN to determine which key is currently being pressed is independent of the key codes being entered into the buffer.

Implementing Analog Keys

The most common use of the KEY DOWN function is to provide the program with feedback of the duration of a particular keystroke. Since the length of time that the key is held down is not stored in the key buffer, your program has to keep track of this information. The following program does not directly keep track of the duration of the keystrokes, but re-iterates the defined key action for as long as the key is held down.

```
105 ! Example of KEY DOWN used as an analog key
106 !
110 CCLEAR @ GCLEAR
120 SCALE 0,255,0,191 !                 256 by 192 raster screen.
130 X=127 @ Y=95 @ MOVE X,Y !           Begin at center.
140 P=1 @ PEN P !                       Default is white dots.
145 !
150 ENABLE KBD 128+32 !                 RESET and SFK's only in
155 !                                   program execution mode.
160 ON KEY# 1 GOSUB 250
165 !
170 K=KEY DOWN !                        Monitor keys, ignoring
180 IF K<156 OR K>162 THEN 170 !        all but cursor keys.
185 !
190 ON K-155 GOTO 200,210,220,220,220,230,240
195 !
200 X=X-1 @ PLOT X,Y @ GOTO 170 ! Left arrow key.
205 !
210 X=X+1 @ PLOT X,Y @ GOTO 170 ! Right arrow key.
215 !
220 GOTO 170 !                           Ignore ROLL and -LINE keys.
225 !
230 Y=Y+1 @ PLOT X,Y @ GOTO 170 ! Up arrow key.
235 !
240 Y=Y-1 @ PLOT X,Y @ GOTO 170 ! Down arrow key.
245 !                                   Pressing SFK #1 toggles
250 P=-P @ PEN P @ BEEP !               between draw and erase.
260 RETURN
```

As you can see, the defined function of each key monitored by KEY DOWN is continued for the duration of time it is down. You should be aware that if more keys are to be monitored in the program, the overall response to each key may be decreased.

Also notice that the special function keys may still be used. The keys still operate under interrupt control, but they cannot be recognized when another key is being held down. This implies that the cursor keys must be released to change the plotting mode from draw to erase (and back).

Chapter 4
Table of Contents

Program Storage and Retrieval

Chapter 4
Program Storage and Retrieval

Introduction

This chapter describes using the tape drive during program development, which greatly simplifies the processes of storing and retrieving programs. The tape can also be used to store programs permanently.

If other media are to be used for permanent program storage, the tape cartridge is used to emulate these other media during development. Initially, the program retrieval statements are used to access the tape. When programs are finalized, they are formatted and then transferred via an interface card to permanent storage locations. The program-retrieval statements are then re-directed to access the permanent storage device. Both the formatting and transfer processes are outlined in "Permanent Program Storage".

Figure 4-1 may help you visualize the options available for storing programs.

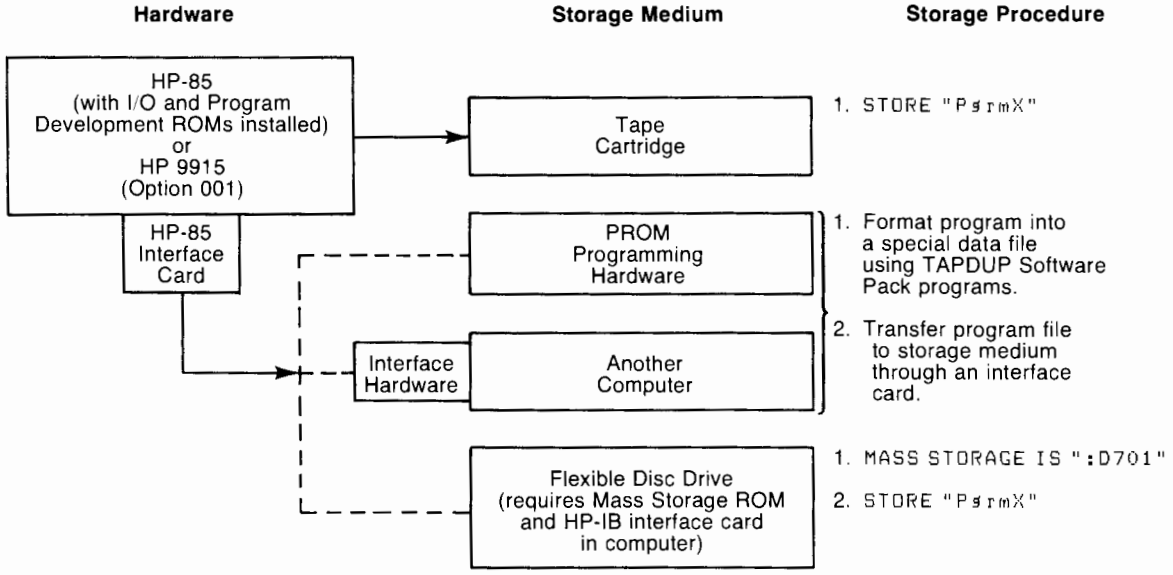


Figure 4-1. Program Storage Methods

The Program Development ROM provides the HP 9915 and HP-85 with the statements used to **retrieve** programs from internal PROM and from other computers. This chapter describes these additional statements.

Review of HP-85 Storage and Retrieval Operations

The standard HP-85 STORE command[1] is used to keep copies of programs on tape. An internal representation of the BASIC program in computer memory is stored on the tape.

The HP-85 LOAD command and CHAIN statement are the means of retrieving STOREd programs from the tape. The differences between the two operations are as follows.

- The CHAIN statement maintains all variables in the memory area declared in the COM statement(s). The LOAD command SCRATCHes memory before loading the program.
- The CHAIN statement maintains binary programs, but the LOAD command clears any binary program in memory.
- The CHAIN statement is programmable; the LOAD command is not. The CHAIN statement automatically runs the loaded program; the LOAD command does not.

Storing Programs During Development

Using the STORE command to keep copies of programs on tape simplifies development. You may use LOAD, CHAIN, and LOADBIN to retrieve the STOREd programs, or you may use PLOADGO, PCHAIN, and PLOADBIN provided by the PD ROM. The differences between these statements are described in the next section.

Program Retrieval Statements

If programs are to be stored permanently in PROM or in other computers, you should use the PLOADGO, PCHAIN, and PLOADBIN statements. These statements can also retrieve STOREd programs from tape. However, you can later use these statements to retrieve programs from the other storage devices by merely directing program requests to the desired device. The process of storing programs in PROM and other computers is outlined in "Permanent Program Storage".

1 See Section 11, "Using Tape Cartridges", of the *HP-85 Owner's Manual and Programming Guide*. Programs may also be stored on disc when the computer is equipped with a Mass Storage ROM. However, if programs are to be stored in PROM or in other computers, the programs must be STOREd on tape before being formatted. See "Formatting Program Files" for further information.

Specifying the Program Source

The PLOADGO, PCHAIN, and PLOADBIN statements are similar in effect to LOAD, CHAIN and LOADBIN. The "P" prefix implies that the request for a program will be directed to the device which has been specified by the most recent PROM IS statement. The default PROM IS device is the tape drive in the HP-85 and PROM in the 9915.

Either of the following statements may be used to specify the **tape drive** as the source of program information.

```
PROM IS TAPE
PROM IS -1
```

Either of the following statements may be used to specify **internal PROM** as the source of program information.

```
PROM IS PROM
PROM IS 0
```

The following **general form** of the PROM IS statement is used for specifying all other possible devices as the source of program information.

```
PROM IS device selector
```

The "device selector" parameter identifies the interface to be used to transfer the program requests and information. The device selector consists of an interface select code (and primary address, if necessary)[1]. The HP-IB and Serial interface cards are generally suggested for this purpose.

If an interface card is used to retrieve programs, the value of the interface select code determines the mode of program transfer. If an odd interface select code is specified, a binary protocol is used for the transfer. An even interface select code forces an ASCII[2] protocol. The protocols are further described in "Program Retrieval Operations".

1 Interface select codes range from 3 through 10. For more information on primary addresses, see the section pertaining to your interface card in the *HP-85 I/O Programming Guide*.

2 ASCII stands for American Standard Code for Information Interchange.

Loading and Running Programs

If a program is to be loaded and automatically run **without** maintaining the COM variables, use the statement:

```
PLOADGO name
```

The parameter "name" can be any string expression. If name is longer than six characters, only the first six are used.

The result of executing PLOADGO is very similar to that of executing LOAD except for the following differences.

- The program source is defined by the PROM IS statement.
- The program is automatically RUN when the load is finished.
- The PLOADGO statement is programmable.

As with the LOAD statement, any program currently in memory, including any binary program, is SCRATCHed before the specified program is loaded. Unlike the CHAIN statement, no COM variables are maintained. Using this statement allows you to delete binary programs, declare a different COM area, and change the option base.

If the loaded program uses any binary program keyword(s), the binary program must be in memory **at the time it is referenced**. Error 50 is generated if the binary program is not present when one of its keywords is used. Consequently, if the program to be loaded has been stored in its de-allocated state, the binary program referenced must be in memory **before** the BASIC program is loaded, allocated, and run by PLOADGO[1]. An example is given in "Changing Binary Programs".

If the specified program is not found on the current PROM IS device or is the wrong file type, the statement is aborted and the next program statement is executed. All other errors are discussed in the section called "Retrieval Errors".

1 The binary program is "referenced" when the program is allocated prior to being run. Allocated and de-allocated program states are discussed in "Memory Allocation".

Maintaining COM Variables

If any variables are to be maintained while changing programs, they must be declared in a COM statement. This common variable area is maintained when the following statement is used to load and run programs.

```
PCHAIN name
```

The parameter "name" can be any string expression. If name is longer than six characters, only the first six are used.

As with the CHAIN statement, either the programs must have matching COM areas or the program executing the PCHAIN must have no COM area. Thus, either the current program passes values to the program being loaded or the program being loaded defines a new COM area.

The source of the program is the device specified in the last PROM IS statement. If the specified program is not found on the current PROM IS device or is the wrong file type, the next program statement is executed. All other errors are discussed in the section called "Retrieval Errors".

Loading Binary Programs

Any binary program may be loaded from the device specified by the PROM IS statement. The resultant action is identical to that of executing the LOADBIN statement except that the program source is the current PROM IS device.

```
PLOADBIN name
```

The parameter "name" can be any string expression. If name is longer than six characters, only the first six are used.

Only **one** binary program can reside in memory at any time. Attempting to load a binary program when one is already in memory results in error 25.

Changing Binary Programs

To change binary programs, you must first clear computer memory. Executing PLOADGO, LOAD, SCRATCH, or AUTOSTART or cycling power clears both BASIC and binary programs **and variables** from memory. The desired binary and BASIC programs may then be loaded.

There are only two ways to change binary programs under program control. The first method is as follows. PLOADGO a program, automatically deleting both the current binary and BASIC programs. The desired binary program can then be loaded into memory by the new program (using PLOADBIN or LOADBIN). The following example illustrates the first method.

```
105 ! Program "OLDPGM".
106 !
110 PLOADBIN "OLDBIN"
     .
     .
     .
975 ! Now change binary (and consequently BASIC) programs.
976 !
980 PLOADGO "NEWPGM" ! Load new BASIC program in first.
```

The loaded BASIC program then loads the new binary program.

```
105 ! Program "NEWPGM".
106 ! This program is stored in the allocated state or does
107 ! not reference the binary program "NEWBIN", or both.
108 !
110 PLOADBIN "NEWBIN"
     .
     .
```

A problem arises if the BASIC program to be loaded is not stored in the allocated state[1] **and** references the binary program to be loaded. If the above method is attempted, the BASIC program is loaded into computer memory but cannot be allocated (and subsequently run) because of the missing binary program. Error 50 is generated and the program line(s) containing the reference(s) to the binary program cannot be listed.

1 The allocated and de-allocated program states are discussed in "Memory Allocation".

The solution to this dilemma is to use an "intermediate" BASIC program whose only purpose is to first load the binary program and then load the BASIC program which uses the binary program's keyword(s). The following example shows this second method.

```
      .
      .
      .
975  ! Changing the binary program with a de-allocated program.
976  !
980 PLOADGO "CHGBIN" ! This deletes binary and BASIC programs.
```

The intermediate program loads the binary and then "chains" the BASIC program into memory.

```
105 ! Program "CHGBIN".
106 !
110 PLOADBIN "NEWBIN" ! Load new binary program first.
120 PCHAIN "NEWPRG" ! CHAIN (and run) the new BASIC program.
```

The new BASIC program is free to declare a new COM variable area, if desired. If parameters are to be passed between programs, mass storage (tape or disc) may be used.

```
105 ! Program "NEWPRG".
106 !
110 ! This program may or may not declare a COM area.
      .
      .
      .
```

Unlike PLOADGO and PCHAIN, all errors (including errors 67 and 68) that occur as a result of executing the PLOADBIN statement are reported.

Permanent Program Storage

The introduction to this chapter briefly described the media used to store programs. To refresh your memory, the media are flexible discs, tape cartridges, internal PROM, and external computers. This section further describes using the latter three media for "permanently" storing programs.

Before final (or "permanent") storage of programs is made, there are two factors you may want to consider. First, you may want to reduce the amount of media space required to store your program(s). Second, you may want to secure your program(s) against being listed, copied, or edited. File security is discussed in Section 11 of the *HP-85 Owner's Manual and Programming Guide*.

Reducing Program Storage Space

The amount of media storage space required to store a program can be reduced in the following ways:

1. Remove all **unnecessary** comments, such as:

```
10 ! This is a comment.
20 REM This is also a comment.
```

   The *Tape Duplication and EPROM Programming Software Pack* provides the SHRINK and GETSAV programs which automatically remove the first (but leave the second) type of comment shown above. For further instructions, refer to the Software Pack manual.

2. Use multi-statement lines whenever possible. An example is:

```
100 BEEP @ DISP "INPUT A$" @ INPUT A$
```

   The "@" statement-concatenation operator requires only one byte of storage while using separate lines requires four bytes. However, be judicious when consolidating statements on one line. Remember that all interrupt requests (other than RESET, SELF TEST, and AUTOSTART) are only serviced at the end of program lines. The longer it takes to execute a program line, the longer the potential delay is in servicing the interrupt.

3. Store programs in the de-allocated state. The amount of media space saved is proportional to the number (and size) of variables used in the program. The next section describes a program's allocated and de-allocated states.

Memory Allocation

It was mentioned earlier that programs are STOREd on tape using an internal representation. Actually, there are two states of this internal-form program, known as the allocated and de-allocated states. The important difference between the two states is the file size required to store the program.

Memory is allocated for variables by the operating system when the program is run or when the INIT[1] key is pressed. When you STORE a program, the operating system automatically attempts to store the program in its allocated state. However, if the program contains a COM[2] statement or an error, it is stored in the de-allocated state. Programs containing COM areas are not stored in the allocated state because the COM variables would be overwritten when the program is loaded back into program memory. Inclusion of a COM statement in a program can be used to force de-allocated program storage.

There is a trade-off between program storage space and program allocation time when programs are stored in these two states. Programs stored in the allocated state require more storage space on the mass storage medium but do not have to be allocated before they are run. The converse is true of programs stored in the de-allocated state. Since the allocation time may be exactly offset by the additional time required to retrieve an allocated program, the decision factor may be storage space. You must make this decision based on empirical measurements.

The **difference in media space** required to store the program in the two states can be easily determined. First, de-allocate the program by "editing" any program line[3]. Now execute a LIST 9999. The number returned is the available computer memory (in bytes). Allocate the program by pressing INIT, followed by executing the LIST 9999 again. The difference between the two numbers is the media space saved by storing the program in its de-allocated state. The more variables used in the program, the greater the difference in storage requirements.

The **allocation time** of any program can be determined by again editing any line and then running the program. Since the program is de-allocated by the edit operation, the time between pressing the RUN key and the RUN light being turned on is the allocation time of the program.

1 See "Initializing a Program" in Section 6 of the *HP-85 Owner's Manual and Programming Guide.*

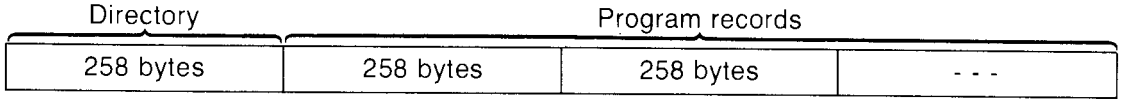
2 See "Declaring and Dimensioning Varibles" in Section 8 of the *HP-85 Owner's Manual and Programming Guide.*

3 LIST any program line(s), move the cursor into a line, and then press **END LINE**. The operating system then automatically de-allocates the program.

Formatting Program Files

Because of the automatic protocols used by the PD ROM retrieval statements, programs to be stored in internal PROM or in other computers must be formatted into special files before being transferred to the final storage medium. The IMAGE and TAPDUP programs provided by the *Tape Duplication and EPROM Programming Software Pack* can be used to format the desired program into the proper file type automatically. The software pack also provides two programs which are examples of transferring program records to PROM programming devices (DIOix and DIOxix). For examples of storing programs in other computers, see the *Networking Technical Supplement.* The program must be on tape prior to the formatting procedure. The following discussion describes the formatted program files generated by the IMAGE program.

The **complete storage file** must consist of a 258-byte directory followed by a series of 258-byte program records.



The directory contains thirty-two 8-byte entries with one entry for each program file, and a 2-byte cyclic redundancy check code (or CRCC)[2]. No empty records are permitted. Individual **directory entries** are as follows.

- **Bytes 0 through 5** contain the six-character program name (blanks are used as fill characters if the program name is five or fewer characters in length).
- **Byte 6** contains the number of 258-byte records in this program file (0 through 255).
- **Byte 7** contains the file-type code; a value of 32 (bit 5 set) signifies a BASIC program, and a value of 8 (bit 3 set) signifies a binary program file. Other files are not allowed[1].

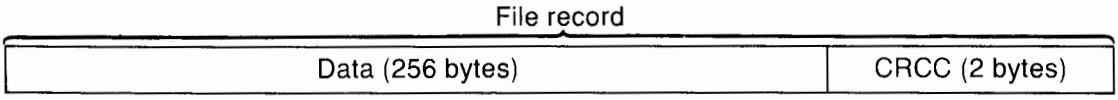
The following example shows the **8-byte directory entry** for a BASIC program called "Prog1" which is 3 records in length.

Name						Length	Type	
0101 0000	0111 0010	0110 1111	0110 0111	0011 0001	0010 0000	0000 0011	0010 0000	Binary representation
P	r	o	g	1	(space)	ETX	(space)	ASCII character

Each **file record** contains 258 bytes of data. The first 256 bytes are the actual program information (in internal form). The remaining two bytes constitute the CRCC used for verification that the transfer has been achieved without errors. Each record is 258 bytes in length **regardless** of the fact that each record may not be full of data. Thus, each program stored on the media begins on a 258-byte record boundary.

1 Even though data files are not allowed, data may be stored in programs as DATA statements, if necessary. See "The READ and DATA Statements" in Section 8 of the *HP-85 Owner's Manual and Programming Guide.*

2 The CRCC is the IBM SDLC Frame Check Sequence. It is automatically generated when one of the program retrieval statements is used: it is also available as the CHECKSUM$ function for use with string data. See the Syntax Reference description for further details.



Transferring Program Files

The program files, formatted and stored on tape by the IMAGE program, are sent to their final storage location(s) through an interface card. You must write the program that initializes the interface card and performs the transfer[1]. Understanding the automatic retrieval protocols may help you take into consideration some of the requirements of your transfer routine. These protocols are described in "Program Retrieval Operations". The rest of this section briefly describes data representations.

Since the data in computer memory is in binary form (data **bytes**), it may or may not be possible to transmit over the interface hardware you are using. However, this binary form is the easiest representation to work with, in terms of the transfer routine, **if** the hardware allows **both** control and non-ASCII characters to be transmitted as data (CHR$(0) through CHR$(31) and CHR$(128) through CHR$(255)).

An alternate data representation is to use the ASCII digits through 7 to represent the octal (base 8) value of each byte . Most interfaces are able to transfer ASCII characters easily. Keep in mind that this representation requires three ASCII characters and one delimiter character for each byte of binary data. The following table shows examples of different data representations.

Data Representation	Characters	
Decimal	129	13
Character	CHR$(129)	CHR$(13)
Binary	1000 0001	0000 1101
Octal	201	015
ASCII representation of octal digits	0011 0010 0011 0000 0011 0001 (delimiter)	0011 0000 0011 0001 0011 0101 (delimiter)

Binary data are easily changed to an octal representation. Assuming that the data byte to be changed is the third character of string D$, the following expression changes this binary character into a six-character ASCII-digit string.

```
O$ = OTO$( NUM( D$[3,3] ) )
```

When outputting the string-data program file, any data representation may be used; however, you must **ensure that the format of the data sent matches that of the data expected by the receiving device.**

1 If you are not familiar with interface card programming at this point, you may want to read Section 3 ("Formatted I/O Operations") in the *HP-85 I/O Programming Guide*

2 If you are not familiar with alternate number bases, see Section 5 ("Why Worry About Bits?") in the *HP-85 I/O Programming Guide.*

Program Retrieval Operations

The PLOADGO, PCHAIN, and PLOADBIN statements invoke different program retrieval routines, dependent on the current PROM IS device. Program retrievals from tape and internal PROM are handled automatically and need no further consideration. However, to transfer programs over different types of interfaces, you need to know more about these program retrieval protocols. Understanding these protocols may be helpful when you write the routine(s) to transfer the program records to final storage.

Program-File Requests

The actual retrieval process consists of a series of program-record requests (or prompts) from the computer, followed by the storage device returning the requested data. Since there are two protocols used for the transfer, the two prompts and the data formats are slightly different. The ASCII protocol is automatically invoked for all transfers that use an interface set to an even select code; the binary protocol is invoked when using an interface set to an odd select code. The data returned to the requesting computer is automatically entered byte-serially according to the selected protocol.

The first prompt sent to the storage device is a request for the directory record. The prompt, which specifies address zero, is OUTPUT to the PROM IS device through the interface card. The storage device must return the directory record in response to this prompt from the requesting computer.

The directory is then searched for the requested program name. If it is found, its position in the directory and the number of records it contains are noted. Its address in the program file is calculated accordingly.

If the program name is not found in the directory or if the file is the wrong type (errors 67 and 68, respectively), no further prompts for the program are sent and program execution continues with the next program statement[1].

All subsequent prompts for program records also include the address of the desired record. The storage device must respond by returning the data record byte-serially. The record contains 256 bytes of program information and the two-byte CRCC. If the CRCC generated during the transfer does not match the CRCC sent with the data record, the requesting computer re-issues the same prompt. Eight consecutive CRCC failures terminates the transfer with Error 116 (PROM or I/O read). Otherwise, the prompts for data from the requesting computer continue until all records for the specified program have been transferred.

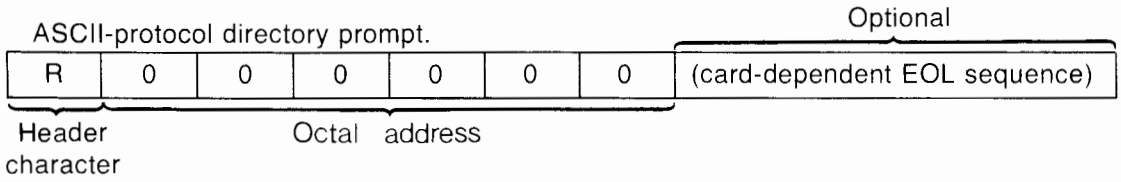
1 The PLOADBIN statement does not ignore these errors, as do PLOADGO and PCHAIN. Error 118 is reported if PLOADBIN does not find the program file in the directory.

The ASCII Protocol

The ASCII protocol is designed for interfaces that cannot transmit certain control codes. When using the ASCII protocol, the automatic prompt for program records sent to the PROM IS device is equivalent to the BASIC statement:

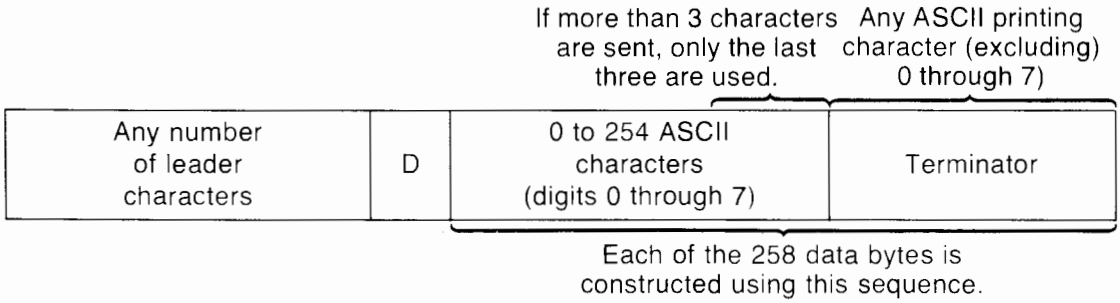
```
OUTPUT device selector USING "A,6A" ; "R",DTO$( address )
```

in which **address** is the six-digit octal address of the first byte of the data record to be returned. An example is shown below.



The data record from the storage device may be preceded by any number of leader characters. The start of valid data is indicated by the ASCII character "D". Each data byte is then formed by a sequence of ASCII digits preceded by any number of blanks (CHR$(32)) and terminated by any printing character **other than** the ASCII digits 0 through 7. Although a maximum of 254 digits will be accepted for each byte, only the one to three characters preceding the terminator are used to form the data byte. All control characters are ignored. If any CONVERT statement is in effect, all characters may be affected.

ASCII-protocol data record returned by the storage device.



Binary Protocol

The binary protocol is designed for maximum transfer rate. The prompt from the requesting computer consists of the ASCII "SOH" character (CHR$(1) or octal 001) followed by the high-order and low-order address bytes of the data to be returned and terminated by any card-dependent EOL sequence in effect. The equivalent BASIC statement is:

```
OUTPUT device selector USING "A,W" ; CHR$(1), address
```

in which **address** is the two-byte binary address of the first byte of data requested. An example is shown below.

Binary-protocol directory prompt.

Header character	Address		Optional
SOH (0000 0001)	0000 0000	0000 0000	End-of-line sequence (Card-dependent)

The program record may be preceded by up to 255 leader characters as long as the ASCII "STX" character (CHR$(2) or octal 002) is not among them. The start of program data is indicated by a single STX character followed by the 258-byte record. No terminator is required, but if one is sent, it is treated as the leader of the next record. An example is shown below.

Binary-protocol data record returned by the storage device.

	Header Character		2-byte CRCC
0 to 255 leader characters	STX (0000 0010)	258 bytes of data	

If any CONVERT statement is in effect, only the header characters (SOH and STX) are affected.

Retrieval Errors

The processes invoked by PLOADGO, PCHAIN, and PLOADBIN involve several steps. Errors can occur at different steps of the process and require explanation at this point.

The first step in retrieving a program is to request the directory from the PROM IS device. If the directory is returned successfully, it is checked for the requested program. If the program is not found by the PLOADGO or PCHAIN statement or is of the wrong type, the program continues as if the statement had never been executed. If the program is not found by the PLOADBIN statement or is of the wrong type, the error is reported (errors 67 and 68, respectively). If the directory is not returned by any of these statements, the error is reported. All of these errors can be trapped with the ON ERROR statement.

If the program is found in the directory and is of the correct type, errors can still occur during the transfer process. If any error does occur during the data transfer, it is reported and the program in memory may have been lost. The error is reported as if the statement was executed from the keyboard with no program in computer memory. For this reason, you should STORE all programs that use these statements before running them. There is no programmable method of recovery from this type of error.

Errors that commonly occur during the retrieval process are:

Error message	Cause
Error 110: I/O Timeout	The interface used for the retrieval has not received or is blocked from transmitting data for the time period specified by the ON TIMEOUT statement for this interface.
Error 114: No user PROM	Attempted to load a program from PROM in the HP-85.
Error 116: PROM or I/O read	Eight consecutive failures of a program to verify the CRCC when loaded from PROM, or PROM board is not installed properly in the 9915.
Error 124: ISC	Attempted to load program from interface select code not present in the computer.

Appendices
Table of Contents

Appendix A
Program Development ROM Syntax Reference

Syntax Conventions

Items enclosed in rounded boxes must be entered exactly as they appear in the box. Items in square boxes are parameters that can (or must) be used in the statement and are described in the text following the syntax drawing.

The lines connecting the boxes can only be followed in one direction when generating a statement. A statement element is optional if there is a valid path around it. Thus, any combination of elements can be used as long as the arrows are followed in the proper direction.

Parameter Definitions

numerical constant — a fixed number within the range of the computer (e.g., 3.1416, 8.854E-12).

string constant — a combination of alphanumeric characters enclosed within quotes (e.g.,"A0Z9", "Rubber duck").

numerical variable — an alphanumeric specifier that references a location in memory which can be used interchangeably with the number it references (e.g., A0, Z9).

string variable — an alphanumeric specifier, followed by "$", that references a memory location which can be used interchangeably with the string it references (e.g., A0$, Z9$).

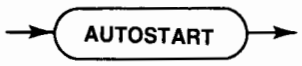
numerical expression — any combination of constants, variables, functions, and operators that can be evaluated by the computer to form a number (e.g., A*X↑2+B*X+C, A MOD 37).

string expression — any combination of constants, variables, functions, and operators which can be evaluated by the computer to form a string (e.g., "Line "&L$[3], VAL$(D)&UPC$(S$)).

buffer — a string variable that has been declared an IOBUFFER. Data characters are put into and taken from the buffer according to the fill and empty pointers, respectively. If the buffer is "qualified" in any way during an operation (such as B$[1]), the buffer pointers are ignored and it is treated as any other string.

AUTOSTART

Syntax



Examples

```
100 AUTOSTART
100 IF A$="POWER UP" THEN AUTOSTART
```

Parameters
none

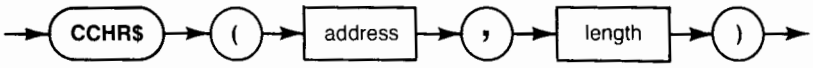
Action Taken
This **statement** is used to initiate the AUTOSTART procedure on both computers. All conditions are set to their power-on state, and a shortened version of SELF TEST is run on the 9915. All tests are successful if the SELF TEST light on the 9915 is turned off. Failure of any AUTOSTART test is indicated by the SELF TEST light remaining on; errors are also reported on the CRT, if installed. These tests are not run on the HP-85.

Both computers attempt to run the program called "Autost". Tape is first searched for the file; if it is found, it is loaded and run. If not found on tape, internal PROM is searched (only on the 9915). If the program is not found, error 48 is reported (on the 9915) and the computer enters the keyboard-interpret mode. Programs can be entered or requested from the keyboard.

Further References
Table of RESET Conditions (Appendix B), Interpreting AUTOSTART Test Results (Appendix C), Chapter 3 of the HP 9915 Installation Manual

CCHR$

Syntax



Examples

```
100 D$=CCHR$(32,32) !    Read all of 2nd display line.
100 C$=CCHR$(L1*32,L2) ! L1 is line #; L2 is #chars.
```

Parameters
address — any numerical expression.
length — any non-negative numerical expression.

Action Taken
This **function** returns a string of **length** characters beginning at **address**. Since address can be any numerical expression while actual display memory addresses range from 0 through 2047, all specified values of address are automatically mapped into this range. A negative address may be used; using a negative length will generate error 89.

All display memory locations are initialized to CHR$(13) at CCLEAR, AUTOSTART, and power-on.

Non-underlined characters have codes 0 through 127; underlined characters have codes 128 through 255. A non-underlined character can be underlined by adding 128 to its code.

Common Error	Cause
Error 89 : INVALID PARAM	Specified negative length.

Further References
How the Display Works, Reading the Display Contents

CCLEAR

Syntax

Examples

```
100 CCLEAR !                                  Blank and reset display.
100 IF NUM(K$[C,C])=146 THEN CCLEAR ! New CLEAR Key response.
```

Parameters

none

Action Taken

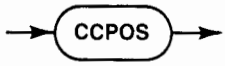
This **statement** "clears" display memory (sets all characters to CHR$(13)), sets the beginning of the display window to line 0, and positions the cursor to address 0. The statement is different from CLEAR in its window and cursor positioning and complete display memory clearing.

Further References

How the Display Works, Clearing the Screen

CCPOS

Syntax



Examples

```
100 C4=CCPOS !          Store cursor location.
100 CCURSOR CCPOS+32 ! Move cursor down one line.
```

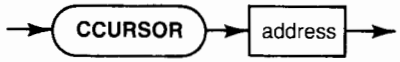
Parameters
none

Action Taken
This **function** returns the current cursor address. The returned value is an integer in the range 0 through 2047. The function can be used with the CCURSOR statement to provide **relative cursor addressing**.

Further References
How the Display Works, Sensing the Cursor Location, Moving the Cursor

CCURSOR

Syntax



Examples

```
100 CCURSOR 32*N !                  Move cursor to Nth line.
100 CCURSOR (CCPOS DIV 32)*32 ! Move to beginning of this line.
```

Parameter

address — any numerical expression.

Action Taken

This **statement** is used to change the cursor's location in display memory. Since **address** can be any numerical expression while actual display memory addresses range from 0 through 2047, all specified values of address are automatically mapped into this range. A negative address may be used.

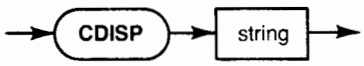
The cursor (underline) is not automatically displayed when its address is changed, but may be restored by executing a CDISP following the CCURSOR statement.

Further References

How the Display Works, Moving the Cursor

CDISP

Syntax



Examples

```
100 CDISP "" ! Restore cursor character after CCURSOR.
100 CDISP Z$ ! Z$ is a buffer.
```

Parameter
string — any string expression or buffer.

Action Taken
This **statement** is used to display characters beginning at the current cursor location. CDISP will not display characters off screen; if the location is not in the current window, the window will be moved so that the displayed character appears in the bottom line of the display window.

CDISPing control characters (codes 0 through 31) causes one of three possible actions: if the code is defined (0, 7, 8, 10, 12, and 13) the defined action is taken; if it is not defined it causes an EOL branch to the line number specified in the last ON CCODE executed. If no ON CCODE is in effect, no branch is taken and the character is ignored as the CDISP continues.

Table 2-2. Control Codes Defined with CDISP

Control Code	Name	Display Action
0	Null	Ignored by the display.
7	Bell	Short BEEP.
8	Backspace	Move the cursor back one position, but never off the display window.
10	Linefeed	Move the cursor down to the same position in the next line; if it is off the window, clear the line and roll the display until the cursor is in the bottom line.
12	Formfeed	Move cursor to the beginning of the next line and roll until that line is at the top of the window; clear all 16 lines in the window.
13	Carriage Return	Return the cursor to the beginning of the line it is in.

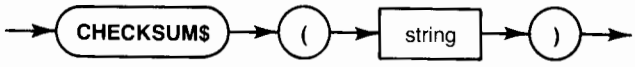
The destination of CDISP is not changed by the CRT IS statement.

If a buffer is used, the empty pointer is updated as characters are removed. However, if the buffer is qualified in any way (such as Z$[1] or Z$&""), the buffer is considered a string and the pointers are ignored.

Further References
Displaying Characters, Displaying Control Codes, Control-Code Branching

CHECKSUM$

Syntax



Examples

```
100 C$=CHECKSUM(A$) !                        Perform CRC on A$.
100 IF CHECKSUM(A$)#"A4" THEN GOTO 80 ! Retry transfer.
```

Parameter

string — any string expression or buffer.

Action Taken

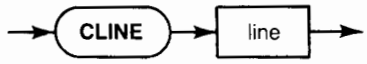
This **function** performs a cyclic redundancy check on the **string**, used for verifying program-file transfers to and from the computer. The statement returns a two-character string.

Further References

Formatting Program Files, Networking Technical Supplement

CLINE

Syntax



Examples

```
100 CLINE 32 !        Display lines 32 through 47.
100 CLINE CLPOS+1 !   Roll (up) one line, rel. to current pos.
```

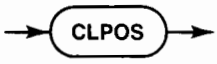
Parameter
line — any numerical expression.

Action Taken
This **statement** positions the display window (changes the beginning line of the display). Since **line** can be any expression while actual display lines range from 0 through 63, all values of line are automatically mapped into this range.

Further References
How the Display Works, Positioning the Display Window

CLPOS

Syntax



Examples

```
100 L=CLPOS !          Store current window position.
100 CLINE CLPOS-16 ! Roll (down) 16 lines.
```

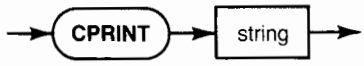
Parameters
none

Action Taken
This **function** returns the current display window position (the beginning line of the current display) as an integer from 0 through 63. It can be used with CLINE to provide **relative window positioning**.

Further References
How the Display Works, Sensing the Current Display Window

CPRINT

Syntax



Examples

```
100 CPRINT D$ !                                  Print buffer contents.
100 IF F1 THEN CPRINT D$ ELSE CDISP D$ ! F1 is print/disp flag.
```

Parameter
string — any string expression or buffer.

Action Taken
This **statement** is used to print character strings on the HP-85 internal printer and incorporates control-code interpretation. Printing begins at the current cursor location (this occurs in an internal buffer; when the buffer is full it is sent to the printer).

CPRINTing control characters (codes 0 through 31) will cause one of three possible actions: if the code is defined (see table), the defined action is taken; if it is not defined it causes an EOL branch to the line number specified in the last ON CCODE executed. If no ON CCODE is in effect, no branch is taken and the character is ignored as the CPRINT continues.

Table 2-3. Control Codes Defined with CPRINT

Control Code	Name	Printer Action
0	Null	Ignored by the printer.
7	Bell	Short BEEP.
8	Backspace	Print the next character at the previous character position; no action was taken if the previous character was in the first column.
10	Linefeed	Advance one line.
12	Formfeed	Advance five lines.
13	Carriage Return	Print next character at first column of this line.

The statement causes no action on the 9915.

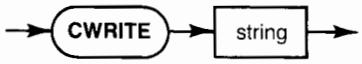
The destination of CPRINT cannot be reassigned by the PRINTER IS statement.

If a buffer is used, the empty pointer is updated as characters are removed. However, if the buffer is qualified in any way (such as Z$[1] or Z$&""), the buffer is considered a string and the pointers are ignored.

Further References
How the Display Works, Displaying Characters, Printing Control Codes

CWRITE

Syntax



Examples

```
100 CWRITE D$
100 CCURSOR 0 @ CWRITE "Line 0" @ CDISP "" ! Move onto screen.
```

Parameter

string — any string expression or buffer.

Action Taken

This **statement** is used to write characters (including control characters; codes 0 through 31) into display memory. Control codes are treated as any other character, unlike CDISP and CPRINT.

If a buffer is used, the empty pointer is updated as characters are removed and written into display memory. However, if the buffer is qualified in any way (such as Z$[1] or Z$&""), the buffer is considered a string and the pointers are ignored.

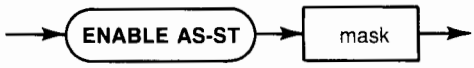
The destination of characters cannot be changed with the PRINTER IS statement.

Further References

How the Display Works, Displaying Characters, Simple Character Display

ENABLE AS-ST

Syntax



Examples

```
100 ENABLE AS-ST A#0 ! Enable (both keys) if A is non-zero.
100 ENABLE AS-ST 0   ! Disable keys.
```

Parameter
mask — any numerical expression that rounds to 0 or 1.

Action Taken
This **statement** provides lockout control of the AUTOSTART and SELF TEST front panel keys on the 9915. If the specified expression rounds to 1, both keys are enabled; if it rounds to 0, they are disabled. All other values of **mask** generate an error.

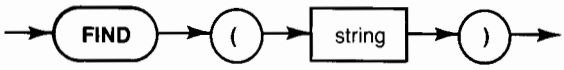
These keys cannot be enabled or disabled separately. These keys produce no key codes to be stored in the key buffer, so ON KBD has no effect on the operation of these keys.

Common Error	Cause
Error 89 : INVALID PARAM	Improper value of mask.

Further References
HP 9915 System Key Lockout

FIND

Syntax



Examples

```
100 P=FIND(A$) !                          Find 1st system key.
100 IF NOT FIND(A$) THEN GOTO 450 !  Skip system-key routines.
```

Parameter
string — any string expression or buffer.

Action Taken
This **function** searches a string or buffer to find character (or key) codes greater than 127 (equivalently, codes with bit 7 set, which are either system keys or underlined characters) and returns the location of the first one found. If none is found, zero is returned.

FIND works the same with buffers as with strings in that the buffer pointers are ignored.

Further References
Finding System Keys, Character and Key Codes (Appendix B)

KBD$

Syntax

Examples

```
100 K$=KBD$ !     Store key codes and empty key buffer.
100 Z$=Z$&KBD$ ! Append key codes to string (or buffer).
```

Parameters

none

Action Taken

This **function** returns the contents of the key buffer as a string and then automatically clears the buffer. Key codes are converted by a KBD CONVERT statement active at the time KBD$ is called, not by one active at the time of the keystroke.

The key buffer is 80 characters long. Pressing more than 80 keys between calls to KBD$ overflows this buffer. When the 81st key is pressed, control of keyboard interrupts is automatically returned to the operating system. The 81st and any further keystrokes are not stored in the key buffer. However, calling KBD$ returns the first 80 key codes.

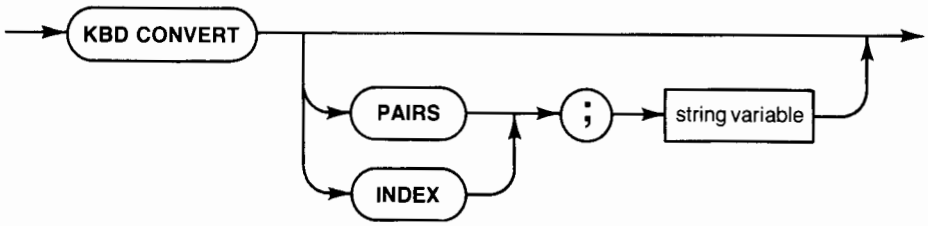
Common Error	Cause
Error 118 : KBD$ Overflow	Failure of program to service keyboard interrupts often enough.

Further References

Controlling Keyboard Interrupts, Keyboard Buffer Overflow, "Converting I/O Data" in the HP-85 I/O Programming Guide

KBD CONVERT

Syntax



Examples

```
100 A$=CHR$(11)&CHR$(9) @ KBD CONVERT PAIRS;A$ ! VT to HT.
100 C$="01234" @ KBD CONVERT INDEX;C$ !Control codes 0-4 to
                                       ASCII characters.
```

Parameter

string — any string variable or buffer.

Action Taken

This **statement** provides conversion of key codes **as they are read from the key buffer with KBD$**. Executing KBD CONVERT without "PAIRS" or "INDEX" turns off conversions until another KBD CONVERT statement is executed.

If PAIRS is specified, the action is to convert the first character to the second character, the third to the fourth, and so forth. If there are an odd number of characters in the conversion string, no conversion is made using the last character.

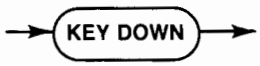
If INDEX is specified, the conversion is made using the original code as the index into the conversion string (or buffer). The first character of the conversion string is defined to be at position zero. If a key code's value is greater than or equal to the length of the specified conversion string, no conversion is made.

Further References

Converting Key Codes, and "Converting I/O Data" in the HP-85 I/O Programming Guide

KEY DOWN

Syntax



Examples

```
100 IF KEYDOWN THEN A=A+1 !    Increment duration counter.
100 K$=K$&CHR$(KEY DOWN) !     Collect key codes in K$.
```

Parameters

none

Action Taken

This **function** returns the code of any key currently being pressed; if no key is down when the function is called, −1 is returned. This function has no interaction with keys being stored in or read from the key buffer, and therefore is not affected by any KBD CONVERT statement.

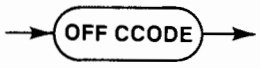
KEY DOWN is normally used while ON KBD or ENABLE KBD is in effect; otherwise, a running program might be PAUSEd when certain keys are pressed.

Further References

Extended Keyboard Control, Monitoring Keys in Real Time

OFF CCODE

Syntax



Examples

```
100 IF K$[P,P]=CHR$(173) THEN OFF CCODE ! Turn off codes.
100 OFF CCODE
```

Parameters

none

Action Taken

This **statement** disables the EOL branch set up by an ON CCODE statement. Subsequent ON CCODE statements can be executed and may use either the same or different branch locations.

Further References

Disabling Control-Code Branching, Displaying Control Codes, Printing Control Codes

OFF KBD

Syntax

Examples

```
100 K$=KBD$ @ OFF KBD !                    Store keys first,
100 IF K$[P,P]=CHR$(31) THEN OFF KBD !   Return control NOW!
```

Parameters

none

Action Taken

This **statement** returns control of keyboard interrupts to the operating system. It also clears the key buffer. If the key buffer contents are not to be lost they must be stored before executing this statement.

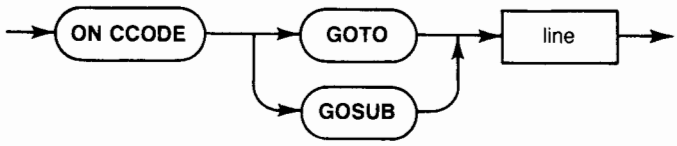
If another ON KBD statement is executed and any key is pressed before the rest of the line is executed, the specified branch is taken at the end of the line.

Further References

Controlling Keyboard Interrupts, Returning Keyboard Control to the System

ON CCODE

Syntax



Examples

```
100 ON CCODE GOSUB 480
100 ON CCODE GOTO 1000
```

Parameter

line — an integer from 1 through 9999.

Action Taken

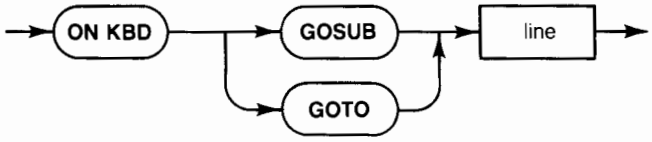
This **statement** establishes the location of the service routine for interrupts generated by CDISPing or CPRINTing control codes other than 0, 7, 8, 10, 12, and 13. When one of the other codes is found by CDISP or CPRINT, the statement is aborted and the branch is taken. If a GOSUB-type branch is specified, control RETURNs to the line **following** the line containing the CDISP or CPRINT.

Further References

Control-Code Branching, End-of-Line Branching (HP-85 I/O Programming Reference).

ON KBD

Syntax



Examples

```
100 ON KBD GOTO 480
100 ON KBD GOSUB 100 @ RETURN ! Re-enable and RETURN
                                on the same line.
```

Parameter
line — an integer from 1 through 9999.

Action Taken
This **statement** establishes the location of the service routine for interrupts generated by the keyboard. An 80-character key buffer is maintained by the operating system to store the code generated by each keystroke; an interrupt is generated as soon as this buffer becomes "non-empty".

As soon as the interrupt has been acknowledged (by taking the branch specified in the ON KBD statement), all further interrupts from the keyboard are disabled until another ON KBD statement is executed. This prevents the keyboard service routine from being interrupted by another keyboard service request before service of the first is complete.

If a GOSUB-type branch is specified, the RETURN statement must be in the **same** program line as the re-enabling ON KBD statement, as shown above. This prevents another GOSUB from being executed before the RETURN from this GOSUB has been made.

Keep in mind that while ON KBD is in effect, the RESET key is treated as any other key. If a RESET-like function is to be maintained, it must be done with software.

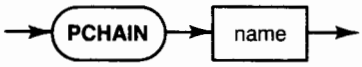
The front panel system keys of the 9915 do not produce key codes; they can only be enabled and disabled with the ENABLE AS-ST statement (see "HP 9915 System Keys" in Chapter 3).

Common Errors	**Cause**
Error 25 : GOSUB Nesting	Failure to execute ON KBD on the same line as RETURN.

Further References
Controlling Keyboard Interrupts, End-of-Line Branching (HP-85 I/O Programming Guide).

PCHAIN

Syntax



Examples

```
100 IF K=3 THEN PCHAIN A$ ELSE PCHAIN B$
100 PROM IS TAPE @ PCHAIN "NEWPRG" !      Change program source.
```

Parameter
name — any string expression.

Action Taken
This **statement** loads and runs the specified program from the current PROM IS device. If the specified file is not found or is the wrong type, the statement is skipped with no error; otherwise, errors are reported as usual.

If the interface select code of the PROM IS device is even, the ASCII protocol is used; if it is odd, the binary protocol is used.

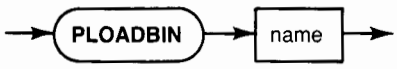
This statement is used to load programs when COM variables are to be passed to the program to be loaded. The COM areas of the requesting program and the program to be loaded must match. If no COM area is currently defined, the incoming program may define a new one. The program is run as soon as it is loaded properly into memory.

Common Errors	Cause
Error 32 : COM MISMATCH	Common areas are different.
Error 116 : PROM or I/O read	Eight consecutive CRCC failures to verify program transfer, or no internal PROM present.

Further References
Maintaining COM Variables, Program Retrieval Operations, Transfer Errors

PLOADBIN

Syntax



Examples

```
100 PLOADBIN "TAPDUP"
100 IF NOT B THEN PLOADBIN E$
```

Parameter
name — any string expression.

Action Taken
This **statement** loads the specified binary program from the device specified by the last PROM IS statement. The ASCII protocol is invoked for even interface select codes; the binary protocol is invoked for odd interface select codes.

There can only be one binary program resident in the computer at one time.

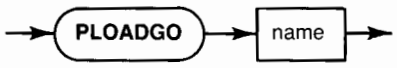
This statement does not ignore errors 67 and 68 as do the PLOADGO and PCHAIN statements.

Common Errors	Cause
Error 25 : TWO BIN PROGS	Tried to load a second binary program into memory.
Error 67 : FILE TYPE	Specified BASIC program.
Error 68 : FILE NAME	No file found in the directory with this name.

Further References
Loading Binary Programs, Changing Binary Programs, Program Retrieval Operations, Retrieval Errors

PLOADGO

Syntax



Examples

```
100 PLOADGO "NEWPRO"
100 IF A THEN PLOADGO A$ ELSE PLOADGO B$
```

Parameter
name — any string expression.

Action Taken
This **statement** loads the specified program from the device specified in the last PROM IS statement. This statement scratches all program information in the computer (including binary programs) before loading in the specified program. If the file is not found or is the wrong type, no error is reported. All other errors are reported as usual.

This statement provides the means to delete binary programs from memory, to allow a new COM variable area to be declared, and to change the option base.

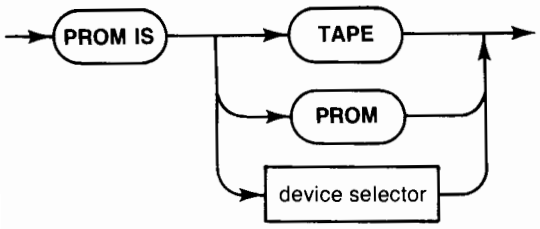
Common Errors

Error 114 : No user PROM	Attempted to load a program from PROM in the HP-85.
Error 116 : PROM or I/O read	Eight consecutive CRCC failures to verify a program record transfer, or PROM board is not installed properly.

Further References
Program Retrieval Statements, Program Retrieval Operations, Retrieval Errors

PROM IS

Syntax



Examples

```
100 PROM IS -1
100 PROM IS 701 ! Binary protocol.
```

Parameter
device selector — interface select code (and primary address, if necessary).

Action Taken
This **statement** redirects program-retrieval requests to the specified device.

Defaults are: PROM IS PROM (or 0) on the 9915
PROM IS TAPE (or −1) on the HP-85

If the interface select code of the PROM IS device is even, the ASCII protocol is used; if it is odd, the binary protocol is used.

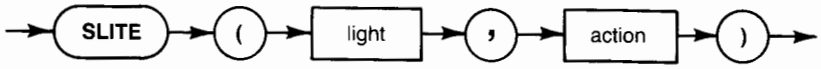
For program sources other than tape, the program must first be formatted into a special file by using a binary program. For more details, see "Formatting Program Files".

Common Errors	Cause
Error 124 : ISC	Specified an invalid select code.

Further References
Specifying the Program Source, Program Retrieval Operations

SLITE

Syntax



Examples

```
100 K9=SLITE(L,A)
100 A=-SLITE(L,A)
```

Parameters

light — any numerical expression.
action — any numerical expression.

Action Taken

This **function** turns the user lights on and off and returns the on/off status of the lights. This function can be used in expressions.

Table 2-1. SLITE Parameter Definitions

Specified action	Resultant action	Status returned
any expression that rounds to 1	turns specified light on	1
any expression that rounds to 0	no change (used to read current status of light)	±1
any expression that rounds to −1	turns specified light off	−1
any other value of action	no change in any light due to improper action value (0 also returned if improper light value specified)	0

Further References

Front Panel Lights, HP-85 User Lights

ST RESULT

Syntax

Examples

```
100 IF ST RESULT < 0 THEN X1=1 !              X1=1 implies 9915.
100 IF ST RESULT MOD 256 <> 0 THEN GOTO 200 !Failed SELF TEST.
```

Parameters
none

Action Taken
This **function** returns the result of the last AUTOSTART test. Using this function is the only way that the program can determine whether it is in a 9915 or in an HP-85.

Further References
Appendix C (Interpreting AUTOSTART Test Results)

Appendix B
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.

	POWER ON	RESET	SCRATCH	RUN	CHAIN	INIT	CONT
PROM IS	R (Tape on HP-85, EPROM on 9915)	R	—	—	—	—	—
SELF TEST	R (Test results)	—	—	—	—	—	—
SLITE	R (All off)	R	R	—	—	—	—
ON KBD	R (Disabled)	R	R	R	R	—	—
CRT Display	R (All memory blanked, CCURSOR = CLINE = 0)	R	—	—	—	—	—
ON CCODE	R (Disabled)	R	R	R	R	—	—
KBD CONVERT	R (Disabled)	—	—	—	—	—	—
ENABLE AS-ST	R (Keys enabled)	—	—	—	—	—	—
ENABLE KBD	R (All keys enabled)	R	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 _J.

EQUIVALENT FORMS				EQUIVALENT FORMS				EQUIVALENT FORMS				EQUIVALENT FORMS			
Char.	Binary	Octal	Dec	Char.	Binary	Octal	Dec	Char.	Binary	Octal	Dec	Char.	Binary	Octal	Dec
@^{c}	00000000	000	0	SPACE	00100000	040	32	@	01000000	100	64	`^{c}	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
D^c	00000100	004	4	$	00100100	044	36	D	01000100	104	68	d	01100100	144	100
E^c	00000101	005	5	%	00100101	045	37	E	01000101	105	69	e	01100101	145	101
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
H^c	00001000	010	8	(	00101000	050	40	H	01001000	110	72	h	01101000	150	104
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
K^c	00001011	013	11	+	00101011	053	43	K	01001011	113	75	k	01101011	153	107
L^c	00001100	014	12	,	00101100	054	44	L	01001100	114	76	l	01101100	154	108
M^c	00001101	015	13	-	00101101	055	45	M	01001101	115	77	m	01101101	155	109
N^c	00001110	016	14	.	00101110	056	46	N	01001110	116	78	n	01101110	156	110
O^c	00001111	017	15	/	00101111	057	47	O	01001111	117	79	o	01101111	157	111
P^c	00010000	020	16	0	00110000	060	48	P	01010000	120	80	p	01110000	160	112
Q^c	00010001	021	17	1	00110001	061	49	Q	01010001	121	81	q	01110001	161	113
R^c	00010010	022	18	2	00110010	062	50	R	01010010	122	82	r	01110010	162	114
S^c	00010011	023	19	3	00110011	063	51	S	01010011	123	83	s	01110011	163	115
T^c	00010100	024	20	4	00110100	064	52	T	01010100	124	84	t	01110100	164	116
U^c	00010101	025	21	5	00110101	065	53	U	01010101	125	85	u	01110101	165	117
V^c	00010110	026	22	6	00110110	066	54	V	01010110	126	86	v	01110110	166	118
W^c	00010111	027	23	7	00110111	067	55	W	01010111	127	87	w	01110111	167	119
X^c	00011000	030	24	8	00111000	070	56	X	01011000	130	88	x	01111000	170	120
Y^c	00011001	031	25	9	00111001	071	57	Y	01011001	131	89	y	01111001	171	121
Z^c	00011010	032	26	:	00111010	072	58	Z	01011010	132	90	z	01111010	172	122
[^{c}	00011011	033	27	;	00111011	073	59	[	01011011	133	91	{/^{s}	01111011	173	123
\$^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	~*^{s}	01111110	176	126
_^{c}	00011111	037	31	?	00111111	077	63	_	01011111	137	95	DEL+^{s}	01111111	177	127

Key Response During Normal 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 during 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			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

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.

Key Codes

Key	Response in ALPHA Mode	Response in Graphics Mode	Decimal Value
DELETE	←	←	168
STORE	→	→	169
LOAD	↑	↑	170
TRACE	↓	↓	171
AUTO	⊢	⊢	172
SCRATCH	⊣	⊣	173

Enable KBD Mask Parameter Definitions

In the keyboard input mode, the computer is temporarily halted awaiting operator response. This is the mode of operation for the INPUT statement. The four classes of key masks for the keyboard input mode are:

1. RESET
2. PAUSE
3. Special Function Keys and KEYLABEL key
4. Other keys (all remaining keys not in classes 1, 2, or 3)

The ENABLE KBD statement takes as a parameter the key mask desired for the appropriate operating mode. The upper four bits of the mask parameter specify the program execution keyboard mask. The lower four bits specify the keyboard input key mask. The bits of the mask parameter are shown below. Setting a bit **enables** the corresponding key(s), while clearing a bit **disables** the key(s). That is, only those keys having a bit set in the ENABLE KBD mask will respond.

Bit Number	Decimal Value	Operating Mode	Keys Masked
7	128	↑ Program Execution ↓	RESET
6	64		PAUSE
5	32		Special Function Keys and KEYLABEL
4	16		Other keys
3	8	↑ Keyboard Input ↓	RESET
2	4		PAUSE
1	2		Special Function Keys and KEYLABEL
0	1		Other keys

Branch Precedence Table

Branch Type	Select Code							
	3	**4**	**5**	**6**	**7**	**8**	**9**	**10**
ON ERROR	----------1----------							
ON KBD	----------2----------							
ON CCODE	----------3----------							
ON INTR	4	5	6	7	8	9	10	11
ON TIMEOUT	12	13	14	15	16	17	18	19
ON EOT	20	21	22	23	24	25	26	27
ON KEY	----------28----------							
ON TIMER (timer)	29 (#1)			30 (#2)			31 (#3)	

Appendix C
Interpreting AUTOSTART Test Results

The results of the AUTOSTART tests are reported on the front panel of the 9915 and on the HP-85 (and 9915, if installed) CRT displays. If all tests pass, the SELF TEST light of the 9915 is turned off. Independent of the success or failure of these tests, the computer searches for the "Autost" program file.

Using the Program Development ROM in the HP-85 does not enhance its self-test capability. It does, however, allow you to determine the AUTOSTART test results from the program. The **function**:

```
ST RESULT
```

must be called to determine these results. The **16-bit integer** returned by the function provides this information. It can also be used to determine which computer the program is running in, as the function returns a slightly different value on each computer.

The most important use of the ST RESULT function is in keeping track of test failures that might be missed by the computer user. Since both computers run the Autost program, if found, immediately after reporting test results, there is a possibility that a failing result of any test might be missed by the user. This is especially true on the HP-85, since there is no SELF TEST light.

For this reason, the ST RESULT function should be called by the Autost program, which may take the apropriate action (such as pausing the program) to ensure user awareness of any error found. The following program segment is an example of the routine that you can use to ensure any test failures are noticed by the user. All failures indicate that computer hardware needs to be inspected and should be reported to your service representative.

```
105 ! Example of a typical "Autost" program.
106 !
110 IF NOT ERRN THEN GOTO 200 ! All is OK, so continue.
120 DISP "Error number "&ERRN&" has occurred."
130 DISP @ DISP "If you wish to disregard this"
140 DISP "failure, press CONT."
150 BEEP 50,1000
160 PAUSE
165 !
200 DISP @ DISP "Program execution continuing"
    .
    .
    .
```

The definitions of each bit of the returned value are as follows.

HP-85

MSB LSB

1	1	1	1	1	1	1	1	8-Bit Error Code

HP 9915

MSB LSB

0	0	0	0	d11	d10	d9	d8	8-Bit Error Code

In which:
d11 = SP timer failure
d10 = SP interrupt failure
d9 = SP ROM failure
d8 = SP RAM failure

The error codes are interpreted as follows. Since the upper byte of the value returned on the HP-85 is all ones, it is interpreted as a negative integer by the computer. This tells the program that it is being executed in an HP-85. To isolate the error code, the value can be interpreted MOD 256.

The error codes returned by the function have the following meanings.

Error Code	Error Message Reported on the CRT Display	Error Cause
111	Error 111 : Invalid ROM set	I/O ROM missing
112	Error 112 : PD ROM	Bad PD ROM checksum
113	Error 113 : SP Self Test	SP self test failure
115	Error 115 : SP Timeout	SP failed to respond to CPU
200	Error 200 : SELF TEST	SP RAM test failure
201	Error 201 : SELF TEST	Op. system ROM #1 failure
202	Error 202 : SELF TEST	Op. system ROM #2 failure
203	Error 203 : SELF TEST	Op. system ROM #3 failure
204	Error 204 : SELF TEST	Op. system ROM #4 failure
205	Error 205 : SELF TEST	Graphics character ROM failure
206	Error 206 : SELF TEST	Graphics controller failure
207	Error 207 : SELF TEST	Tape controller failure
208	Error 208 : SELF TEST	Keyboard controller failure
208	Error 209 : SELF TEST	RAM refresh failure
255	(depends on error)	Error detected by option ROM other than PD ROM

Part No. 09915-90010
Microfiche No. 09915-99010

Printed in U.S.A.
First Edition January 1981