HEWLETT-PACKARD

I/O ROM

PROGRAMMING TECHNIQUES MANUAL

For the HP-75

HEWLETT
PACKARD

HP-75 I/O ROM

Programming Techniques Manual

January 1984

00075-90243

Printed in U.S.A.

Contents

How To Use This Manual

Please take a minute to read this introduction so that you can better understand how this manual is organized, and how to get the most utility from it. The HP-75 I/O ROM adds many new capabilities to your portable computer, opening a whole new world of applications. This manual is intended as both a **learning** and a **reference** tool. At first, you may use it to learn the fundamentals of I/O programming with your HP-75, and to become familiar with the many new statements and functions that the ROM provides. Later, as you develop your own I/O application programs, the manual will serve as a reference source.

Section 1 covers the installation of the ROM in your HP-75 Portable Computer and gives an overview of the Hewlett-Packard Interface Loop. It is assumed that you are familiar with HP-IL, but you may find the brief review to be helpful. Section 1 also covers the conventions that are used in defining the syntax of statements and functions throughout this manual. Please read the subsection "Syntax Guidelines" in section 1.

Sections 2 and 3 cover the fundamentals of I/O programming, and cover the capabilities of the `OUTPUT`, `ENTER`, and `IMAGE` statements. If I/O programming is new to you, sections 2 and 3 will get you started, and may contain all of the information that you need for most applications. Even if you are an accomplished I/O programmer, you should at least skim through these sections. The concepts presented are basic, but you still need to know how they are implemented for the HP-75.

Section 4 covers the `SENDIO`, `ENTIO$`, and `SEND` statements. These statements deal with the Hewlett-Packard Interface Loop on a message level and provide a wide spectrum of capabilities for the advanced I/O programmer. Section 5 covers several statements and functions that are useful in controlling HP-IL devices through the loop. These statements allow you to assign HP-IL addresses and device codes, to set up devices for remote control, and to identify and poll HP-IL devices.

The appendices provide some useful reference materials. Appendix A covers warranty and service information. Appendix B provides complete syntax definitions for all of the statements and functions covered in sections 1 through 5. Appendix C summarizes the HP-IL command mnemonics used in `SENDIO` and `ENTIO$` statements. In addition to the primary I/O functions covered in sections 1 through 5, the I/O ROM provides many useful support functions. Appendix D gives a complete list of these support functions, describing their operation and syntax. A list of errors and warnings is given in appendix E.

Section 1

Getting Started

The HP-75 I/O ROM gives the HP-75 the capability to communicate with any Hewlett-Packard Interface Loop (HP-IL) talker or listener device. This manual is for programmers who are experienced with the HP-75 and with HP-IL. Familiarity with HP-75 and HP-IL commands is assumed. Information on specific HP-IL commands can be found in the owner's manuals for HP-IL devices, and also in *THE HP-IL SYSTEM: An Introductory Guide to the Hewlett-Packard Interface Loop*, by Gerry Kane, Steve Harper, and David Ushijima, published by OSBORNE/McGraw-Hill, Berkeley, California, 1982. The complete functional, electrical, and mechanical specifications of the HP-IL interface system are given in *The HP-IL Interface Specification* (part number 82166-90017), Hewlett-Packard Company, 1982.

Installing and Removing the ROM Module

CAUTION

Be sure to turn off the HP-75 (press [SHIFT] [ATTN]) before installing or removing any module. If there are any pending appointments, type `alarm off` [RTN] in **EDIT** mode to prevent the arrival of future appointments (which would cause the computer to turn on). If the computer is on or if it turns itself on while a module is being installed or removed, it might clear itself, causing all stored information to be lost.

WARNING

Do not place fingers, tools, or other foreign objects into any of ports. Such actions could result in minor electrical shock hazard and interference with pacemaker devices worn by some persons. Damage to port contacts and internal circuitry could also result.

The HP-75 I/O ROM module can be plugged into any of the three ports on the front edge of the computer.

To insert the I/O ROM, orient it so that the label is right-side up (facing toward you), hold the computer with the keyboard facing up, and push in the module until it snaps into place. Be sure to observe the precautions described above during this operation.

To remove the module, use your fingernails to grasp the lip on the bottom of the front edge of the module and pull the module straight out of the port. Install a blank module in the port to protect the contacts inside.

Note: You may install the HP-75 VisiCalc® ROM and the I/O ROM concurrently, but the VisiCalc ROM must be installed in the rightmost port.

Translating LEX File Programs

Some of the capabilities of the HP-75 I/O ROM have been previously available in the form of LEX files. The I/O Utilities LEX file has been supplied with the *HP-75 I/O Utilities Solutions Book* (HP part number 00075-13013). The Autoloop LEX file has been available as the HP-75 Autoloop Users' Library program (HP part number 75-00104-6). The HP-75 I/O ROM supersedes these LEX files, providing new versions of the statements and functions they contain. To avoid conflicts between the old and new versions of these statements and functions, both LEX files must be purged from your HP-75 before you use the I/O ROM.

If you have written programs using statements and functions from the I/O utilities LEX file and/or the Autoloop LEX file, you can translate these programs so that they will run with the I/O ROM versions of the same statements and functions. The procedure follows:

1. Install the I/O ROM (turn off the computer first).
2. Load the LEX file(s) used in your original program.
3. Load the original program, then convert it to a TEXT file (refer to your *HP-75 Owner's Manual*).
4. Purge the LEX file(s).
5. Transform the program back to BASIC.

The translated program will run just as if it was originally written using the I/O ROM.

The Role of the Hewlett-Packard Interface Loop

The HP-75 I/O ROM provides several useful functions that enable your HP-75 Portable Computer to carry out Input/Output operations. However, an interface or hardware link is needed in order for a computer to communicate with its peripheral devices. The Hewlett-Packard Interface Loop (HP-IL) provides the link through which your HP-75 can communicate with the growing family of HP-IL devices. The HP-75 and all devices included in the interface loop are connected together in series, forming a communications circuit. Any information that is transferred among HP-IL devices is passed from one device to the next around the circuit. If the information is not intended for a particular device, the device passes the information on to the next device in the loop. When the information reaches the proper device, that device responds as directed. In this way, the computer can send information to and receive information from each device in the loop, according to the device's capability. All I/O operations are carried out through this interface loop.

VisiCalc is a registered trademark of VisiCorp.

A Brief Review of HP-IL

Before going further in this manual, you may find it helpful to review the fundamentals of HP-IL. This review covers the material necessary to understand the rest of this manual. Previous exposure to HP-IL is assumed. Users who feel sufficiently comfortable with HP-IL may skip this review.

HP-IL is an interface system in which devices are connected in a circular loop. Devices communicate with each other by sending messages around the loop. When a device sends or **sources** a message, each device in the loop examines the message, then passes it on to the next device. The message is passed around the loop until it returns to the original sender. All messages travel in the same direction around the loop.

HP-IL operates on a master-slave principle. One of the devices in the loop functions as loop **controller**. The controller has the responsibility of transmitting all commands to other devices in the loop. The HP-75 can function as loop controller. A device that can send data, but not commands, to other devices in the loop is called a **talker**. Although a device has talker capability, it will not actually send its data until commanded to do so by the controller. **Listeners** are devices with the capability to receive data from the loop. A listener will not receive data until commanded to do so by the controller.

Each HP-IL device can have one or more of the three basic capabilities: controller, talker, and listener. There can be any number of devices in the loop with controller, talker, or listener capabilities. Only one controller may be active at a time, and only one talker may be active at a time, but there may be more than one active listener. The controller device that was active when the system was turned on is called the **system controller**, and is in charge of the whole system. The HP-75 is always the system controller when used in the HP-IL loop. Figure 1-1 shows a typical HP-IL configuration:

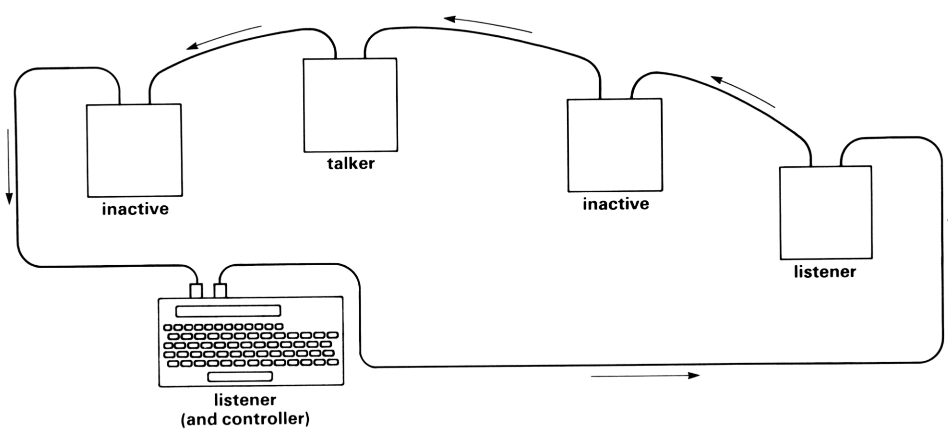

Figure 1-1. Hewlett-Packard Interface Loop

The system controller assigns an address to each device in the loop. It can direct commands to specific devices by using the device address. The address is a number from 0 to 30 or, with extended addressing, from 0 to 960.

Data and commands are sent around the loop as 11-bit **messages**. The first three bits of each message identify the type, or **group**, of the message. There are four groups of HP-IL messages: the command group, the ready group, the identify group, and the data/end group. In this discussion we will consider only command messages and Data Byte messages. The last eight bits are the actual content of the message. Thus, to send a command such as IFC (Interface Clear), a message would be sent out as follows: three bits identifying the message as a command message followed by eight bits with the command code for IFC (binary "10010000"). A Data Byte message consists of three bits identifying it as a Data Byte message followed by eight bits of data.

Each message is examined by every device in the loop. By examining the message, devices determine whether or not any further action is required. Action is indicated in a number of circumstances. Certain command messages, such as IFC, indicate action for all devices in the loop. Other command messages, such as LAD (Listen Address) and TAD (Talker Address), contain a device address. A device acts on the command only if the address in the command is the same as the address of the device. Some messages are processed only if the device is in an active state. Data Byte messages and DDL (Device Dependent Listener) messages are processed only by devices that are in an active listener state. The SDA (Send Data) message is processed only by a device that is an active talker.

An example of how all this works is as follows: Suppose the HP-75 controller wants to print a line on a printer. Assume that the printer has a device address of 2 and that all devices in the loop have inactive status. The controller first sends a LAD2 (Listen Address, Device 2) message around the loop. This puts device 2, the printer, into active listener status. The controller then sources the Data Byte messages. If the line to be printed is an 80-character line, 80 Data Byte messages are sent, followed by one message each for a **carriage-return** and a **line-feed** character. Once data transmission is complete, the controller sources the UNL (Unlisten) command message. This deactivates all listener devices in the loop, in this case, the printer.

Appendix C summarizes the HP-IL commands and their mnemonics.

Device Addresses

In order to distinguish among devices in the loop, each device must have an address — a number from 0 to 30. The system controller assumes the 0 address at power on, and then assigns addresses starting with 1 for the device next in order after the controller in the direction of information transfer. Each device in the loop stores its unique address internally.

Figure 1-2 shows how you can determine the direction of information transfer by noting the differences in the plugs on the HP-IL cables. It may be helpful to remember that information flows out of the computer through the large connector, around the loop, and back into the computer through the small connector. These connectors are labeled IN and OUT as shown in the figure.

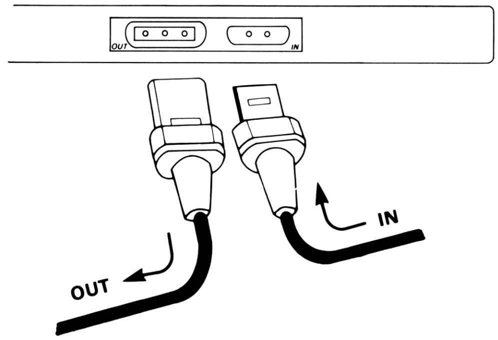

Figure 1-2. Connectors

Device Codes

Once your computer has assigned device addresses to the devices connected in the interface loop, you should assign a device code to each device. Most I/O operations require you to identify devices with device codes. Device codes may be one or two letters, a letter and a digit, or a digit and a letter. Examples of acceptable device codes are `T`, `TV`, `T1`, and `1T`. (A space used as the last character of a device code will be ignored; a space may not be used as the first character.) The letters of device codes may be entered in lowercase, but are converted internally to uppercase. The HP-75 I/O ROM provides two functions — `ASSIGN LOOP` and `AUTOLOOP` — that automatically assign device codes to all devices in the loop (refer to section 5). You may also assign device codes manually with the `ASSIGN IO` command (refer to your *HP-75 Owner's Manual*). When you specify a device code in a command, it must be preceded by a colon and enclosed in quotation marks, for example: `DISPLAY IS ':TV'`. You may also specify a device code by using the name of a string variable, for example: `DISPLAY IS A$` where `A$ = ':TV'`.

Syntax Guidelines

Instructions must be typed with proper **syntax** in order for the computer to understand their meaning. The following guidelines are used throughout this manual in defining the syntax of commands, statements, and functions:

`DOT MATRIX TYPE`	Words in dot matrix type may be keyed in using either lowercase or uppercase letters, but otherwise must be entered exactly as shown. Commands, statements, and functions entered in `lowercase` are converted internally to `UPPERCASE`.
italics type	Items in italics are the parameters you supply, such as the *filename* in the `PURGE` command.
' ', " "	Filenames and other character strings can be enclosed with single or double quotation marks and can be entered in lowercase or uppercase letters. Quoted filenames are converted to uppercase internally.
[ ]	Square brackets enclose optional items.
...	An ellipsis indicates that the optional items within the brackets may be repeated.
stacked items	When two or more items are placed one above the other, one (and only one) of them may be used.
or	When two or more items are separated by **or**, one or more instances of either or both items may be included.

Some examples may clarify the use of these symbols. The syntax of the PURGE command can be represented as follows:

```
       [ 'filename [:device code]' ]
PURGE  [ KEYS                      ]
       [ APPT                      ]
```

In this representation *filename* stands for the name of the file to be purged; *device code* for a valid HP-IL device code. The following statements are all valid:

```
PURGE 'DATA:D1'
PURGE KEYS
PURGE APPT
```

The brackets around : *device code* indicate that the colon and device code are both optional when you are specifying a filename. The outer set of brackets indicates that you may omit all parameters when using the `PURGE` command. Thus, the following statements are also valid:

```
PURGE 'DATA'
PURGE
```

Any parameter represented in this manual as a string in quotation marks (such as '*filename*') may be specified by either a quoted string expression or the name of a string variable that contains the equivalent expression. The following statements are equivalent to `PURGE1 'DATA'`:

```
10 A$='DATA'
20 PURGE A$
```

Section 2

Simple I/O Operations

The principal tools for using HP-IL to move data into and out of the computer are the OUTPUT and ENTER statements. These statements are the **core** of I/O operations. They are usually the fastest and easiest ways of getting data from the source to the destination in its final form. Many applications require no more than the proper use of OUTPUT and ENTER.

Simple OUTPUT and ENTER statements (as described in this section) use ASCII representation for all data. **ASCII** stands for **American Standard Code for Information Interchange**. It is a commonly used code for representing letters, numerals, punctuation, and special characters. The ASCII code provides a standard correspondence between binary codes that are easily understood by the computer and alphanumeric symbols that are easily understood by humans. A complete list of the characters in the ASCII set and their decimal code values is included in the *HP-75 Owner's Manual.*

When special formatting is desired, the OUTPUT USING and ENTER USING forms are very convenient. These forms are discussed in section 3.

Using Simple OUTPUT Statements

A simple OUTPUT statement may be used anywhere that a simple PRINT statement is proper. The OUTPUT statement (like the PRINT statement) contains a list of items to be output, but it also specifies one or more destination devices. You may use either the device code or the HP-IL address of a device in an OUTPUT statement. However, you must use device codes if you are specifying more than one output device. Only one device address may be specified in an OUTPUT statement. Here are some examples of properly syntaxed OUTPUT statements:

```
OUTPUT ':TV';'Hello'
OUTPUT 2 ;X
OUTPUT S1$;A$;B$
OUTPUT ':TV,:PR';X;Y;Z
OUTPUT ':PR';A(1);B(3);N$[2,7];
```

Notice that a semicolon is used to separate the device code(s) or device address from the output list. Semicolons are also used to separate items within the output list. Items in the output list may be numeric variables, numeric constants, string variables, or string constants. An end-of-line sequence (normally **carriage-return/line-feed**) is output after the last item in the output list unless the list is followed by a trailing semicolon.

The simple OUTPUT statement (with items in the output list separated by semicolons) uses the same compact-field output format as the simple PRINT statement. In each numeric output field the digits of a number are preceded by a space (if positive) or a minus sign (if negative), and followed by one space. String data is output with no leading or trailing spaces. Each field (numeric or string) is appended to the field before it. Obviously, compact-field output is inappropriate for many applications. Formatted output, using output images, is described in section 3.

Using Simple ENTER Statements

A simple ENTER statement may be used wherever an INPUT statement is proper. The ENTER statement (like the INPUT statement) contains a list of items to be entered, but it also specifies a device as the source. You may specify either the device code or HP-IL address of the source device in an ENTER statement, but there can be only one source. Here are some examples of properly syntaxed ENTER statements:

```
ENTER 'B1';X
ENTER S1$;A$,B$,C$
ENTER ':TP';X,Y,Z
ENTER 3 ;A(1),B(3),N$
```

Notice that a semicolon is used to separate the device code or device address from the enter list. Commas are used to separate items within the enter list. Items in the enter list may be numeric variables or string variables.

To use the ENTER statement effectively, it is important to understand what constitutes the beginning and ending of an entry into a variable. The simple ENTER statements just shown use a **free field format** for processing incoming characters. This format operates differently with string and numeric data.

Entering Numeric Data

The computer enters numeric values by reading the ASCII representations of those values. For example, if the computer reads an ASCII 1, then an ASCII 2, and finally an ASCII 5, it places the value one hundred twenty five into a numeric variable.Understanding the process that the computer uses to read a free field number can help you remove much of the mystery from I/O. Suppose your program has the statement:

```
ENTER ':TP';X,Y
```

Now assume that when this statement is executed, the following character sequence is received through the interface loop:

T	**U**	**E**	**S**	**D**	**A**	**Y**		**D**	**E**	**C**		**1**	**1**	**,**		**1**	**9**	**7**	**9**	**EOL**

The computer ignores all leading spaces and non-numeric characters, so the TUESDAY DEC characters do nothing. Then the 11 is read. Once the computer has started to read a number, a space or non-numeric character signals the end of that number. Therefore, the comma after the 11 causes the computer to place the value eleven into variable X and start looking for the next value. The space and comma in front of 1979 are ignored and the computer reads the 1979. Finally, the EOL (end-of-line) sequence causes the computer to place the value nineteen hundred seventy nine into variable Y and terminate the ENTER statement. The computer goes on to the next program line with X=11 and Y=1979.

Note: The HP-75 allows you to change the EOL (end-of-line) sequence with the ENDLINE statement (refer to the *HP-75 Owner's Manual*). The default **EOL** sequence is a two-character sequence consisting of a **carriage-return** followed by a **line-feed** character. In this manual **EOL sequence** refers to the current end-of-line sequence that you have set with the ENDLINE statement (unless otherwise noted). The symbol **EOL** is used to represent the end-of-line sequence in the examples.

The process just described can be easily summarized. When entering numeric data using free-field format, the computer:

1. Ignores leading spaces and non-numeric characters.
2. Uses numeric characters to build a number.
3. Terminates the building of a value when a trailing space or non-numeric character is encountered.
4. Inputs characters until an EOL sequence or End Byte message is encountered.

The discussion so far has referred to numeric and non-numeric characters without being specific. The digits `0` through `9` are always numeric characters. Also, the decimal point, plus sign, minus sign, and the letter `E` can be numeric if they occur at a meaningful place in a number. For example, assume that the following character sequence is read by an `ENTER` statement:

–	–	T	E	S	T		1	2	.	5	E	–	3

If a numeric value is being entered, the leading minus signs and the `E` in `TEST` will be ignored. They have no meaningful numeric value when surrounded by non-numeric characters. However, the characters `12.5E-3` will be interpreted as 12.5 x 10^{-3}. In this case, the minus sign and the exponent indicator (`E`) occur in a meaningful numeric order, so they are accepted as numeric characters.

Entering String Data

The computer enters string data by placing ASCII characters into a string variable. The process used for free-field entry is straightforward. All characters received are placed into the string until:

1. The string is full, or
2. An EOL sequence or End Byte message is received.

Assume that the computer is executing the statement:

```
ENTER ':TP';A$,B$,C$
```

The following character sequence is received:

H	E	L	L	O	EOL	EOL	T	H	E	R	E	EOL

The letters `HELLO` are placed into A$ when the first EOL sequence is encountered. Note that the EOL sequence itself is not placed into A$; it acts only as a terminator for the entry into A$. The entry into B$ begins. However, an EOL sequence is read immediately. This terminates the entry into B$, and B$ becomes the null string. Next, the entry into C$ begins. The characters `THERE` are placed into C$, terminated by the EOL sequence that follows those characters. With the enter list now satisfied and an EOL sequence detected at the end of the data, the computer will go on to the next program line.

Note: The current EOL sequence (specified with the `ENDLINE` statement) will act as a terminator and will not be entered into the string. If the current EOL sequence is **carriage-return/line-feed**, this sequence will terminate entry into a string variable and will notitself be entered. However, other potential end-of-line sequences (such as the **line-feed** character by itself) will not terminate entry and will be entered into the string. An End Byte message will terminate entry after its character has been entered into the string.

Another example can be used to show termination on a full string. This time, suppose the program contains the following statements:

```
DIM X$[3]
ENTER ':TP';X$
```

The following characters are sent to the computer:

B	O	Y	C	O	T	T	EOL

The computer places the characters BOY into X$, which fills the dimensioned length of 3. Then the computer continues to read the incoming characters until an EOL sequence is encountered. At that time, the ENTER statement is completed, and the computer goes on to the next program step with X$=BOY.

Section 3

Formatted I/O Operations

Although simple OUTPUT and ENTER statements work well for some I/O situations, there are times when more control over format is necessary. Perhaps a column of numbers with the decimal points in line is desired or an end-of-line sequence terminator is not wanted or expected. There are many reasons for desiring format control during I/O operations.

The format of information sent or received through interfaces is controlled by the use of **image specifiers**. These image specifiers can be placed in an IMAGE statement or can be included directly in an OUTPUT or ENTER statement. This section of the manual provides details on the meaning and use of image specifiers.

Formatted OUTPUT

An output image can control all major characteristics of output data, including spacing, appearance of the field, form of data representation, and use of end-of-line sequences. The HP-75 uses an output image when some form of the OUTPUT USING statement is encountered. There are two forms of this statement:

simplified syntax

10 IMAGE *output image*
20 OUTPUT ':*device code*' USING 10;*output list*

simplified syntax

OUTPUT ':*device code*' USING '*output image*';*output list*

The examples above show the general forms of the OUTPUT USING statement. Here are some specific examples:

```
10 IMAGE 'Total =',ZZ.D
20 IMAGE 5A,2X,17A
    :
60 OUTPUT ':B1' USING 10;C1,C2,C3
70 OUTPUT 2 USING 20;A$,B$
80 OUTPUT S3$ USING 'MDDD.DD';T(1),T(2)
90 OUTPUT ':TV,:PR' USING I$;N$,A
```

In the general forms, *device code* represents a list of one or more device codes (one for each output device). Each device code must be preceded by a colon. Commas separate the successive codes in the list (for example, ':D1,:D2,:D3'). The *device code* field can be occupied by the name of a string variable that contains the list of device codes. The symbol *output image* represents a proper list of image specifiers. The image specifier list may be a literal enclosed in quotation marks or the name of a string variable that contains the specifier list. The specifiers within the list must be separated by commas. The list of items to be output is represented by *output list*. You may use either commas or semicolons to separate items within the output list. All spacing is controlled by the image specifiers, so a semicolon has the same effect as a comma. As with the simple OUTPUT statement, the *output list* can contain numeric or string data (variables or constants), and a trailing semicolon will suppress the output of a final EOL sequence.

Note: You may substitute a valid HP-IL device address for the *device code* field in an OUTPUT statement; however, only one device address may be specified. If you want to specify more than one device, you must use device codes. If the intended destination device has already been addressed to listen, you may leave the *device code* field blank. Refer to appendix B for a complete definition of OUTPUT statement syntax.

Numeric Image Specifiers

The image specifiers in this group are used to control the format of numbers that are output. These image specifiers are the same as the PRINT image specifiers that may already be familiar to you. Since there are many numeric image specifiers, these specifiers are broken down into three categories in the following discussion. The categories are **digit specifiers, sign specifiers,** and **punctuation specifiers**.

Digit Specifiers. These are the image specifiers which form the digits of the number. They allow you to determine the number of digits before and after the decimal point, display or suppress leading zeros, and control the inclusion of exponent information.

Image Specifier	Meaning
d,D	Causes one digit of a number to be output. If that digit is a leading zero, a space is output instead. If the number is negative and no sign image has been provided, the minus sign will occupy one digit place. If any sign is output, the sign will **float** to a position just left of the left-most digit.
z,Z	Same as D, except leading zeros are output.
*	Same as Z, except leading zeros are replaced by asterisks.
e,E	Causes the number's exponent information to be output. This is a 5-character sequence including the letter E, the exponent sign, and three exponent digits.
k,K	Causes the number to be output in compact format. No leading or trailing spaces are output.

Sign Specifiers. These are the image specifiers used to control the output of sign information. Note that if no sign specifier is included in the image, negative numbers will use a digit position to output the minus sign.

Image Specifier	Meaning
s,S	Causes the output of a leading plus or minus sign to indicate the sign of the number.
m,M	Causes the output of a leading space for a positive number or a minus sign for a negative number.

Punctuation Specifiers. These are the image specifiers used to control the output of punctuation within a number, such as the inclusion of a decimal point.

Image Specifier	Meaning
.	Causes an American radix point to be output (a decimal point).
r,R	Causes a European radix point to be output (a comma).
c,C	Usually placed between groups of three digits. Causes a comma to be output to separate the groups of digits (American convention).
p,P	Same as C, except a period is used to separate the groups of digits (European convention).

It would be unrealistic to attempt examples of all possible combinations of these numeric image specifiers. The following examples show some of the many ways of combining these specifiers and the resulting output when numbers are sent to a typical printer. Additional examples for many of the specifiers can be found in the "Display and Printer Formatting" section of the HP-75 *Owner's Manual.*

Sample Statements		Printed Output
`OUTPUT ':PR' USING 'ZZZZ.DD' ; 30.336`		`0030.34`
`OUTPUT ':PR' USING '4Z.2D' ; 30.336`		`0030.34`
`OUTPUT ':PR' USING '4Z.2D' ; -30.336`		`-030.34`
`OUTPUT ':PR' USING '3DC3DC3D' ; 1E6`		`1,000,000`
`OUTPUT ':PR' USING '3DC3DC3D' ; 1.2345E4`		`12,345`
`OUTPUT ':PR' USING '3DC3DC3D' ; 1.2E9`	(Overflow Error)	`   ,   ,`
`OUTPUT ':PR' USING 'SZ.DDD' ; .5`		`+0.500`
`OUTPUT ':PR' USING 'MZ.DDD' ; .5`		` 0.500`
`OUTPUT ':PR' USING 'MD.DDD' ; .5`		`  .500`
`OUTPUT ':PR' USING 'Z.DDE' ; .00456`		`4.56E-003`

Notice in these examples that the image ZZZZ and the image 4Z mean the same thing. The same is true for the D and * specifiers. You can indicate the number of digits desired by simply placing that number in front of the specifier. The use of parentheses, as in 3(D), changes the meaning. The image 3D means "output one numeric quantity in a three-digit field." The image 3(D) means "output three numeric quantities, putting each in a one-digit field."

Be careful of overflow conditions when using these image specifiers. An overflow occurs when the number of digits required to accurately represent a number is greater than the number of digits allowed for in the image. If this happens, a warning is issued and something is output so that the program can continue. However, it is difficult to predict exactly what will be output. The output will probably bear little or no resemblance to the number that caused the overflow.

String Image Specifiers

The image specifiers in this group deal with the output of string characters. They can also be used in combination with the numeric image specifiers for spacing and labeling purposes. All of these image specifiers are the same as the PRINT image specifiers, which may already be familiar to you.

Image Specifier	Meaning
a,A	Causes the output of one string character. If all the characters in the current string have been used already, a trailing blank is output.
'*literal*' or "*literal*"	A **literal** is a string constant formed by placing text or in quotes, using a string function (such as CHR$), or a combination of the two. The character sequence specified is output when a literal image is encountered. When the literal is enclosed in quotes, the quotation marks themselves are not output. Literal images are commonly used for labeling other output.
x,X	Causes the output of one space.
k,K	Causes the string to be output in compact format. No leading or trailing spaces are output.

The following examples show some of the many ways of using these specifiers and the resulting output when the characters are sent to a typical printer. Additional examples for these specifiers can be found in the "Display and Printer Formatting" section of the *HP-75 Owner's Manual.*

Sample Statements **Printed Output**

```
OUTPUT ':PR' USING '5A,A' ; 'X','Y'                    X     Y
OUTPUT ':PR' USING 'K,3X,K' ; 'UNCLE','SAM'        UNCLE   SAM
OUTPUT ':PR' USING 'K,3X,K' ; 98.6,99.9            98.6   99.9

10 IMAGE 'TOTAL = ',3D,X,K
20 T=125 @ A$='CARS'
30 OUTPUT ':PR' USING 10 ; T,A$                 TOTAL = 125 CARS
```

Notice that the X and A image specifiers allow a number before them in the same fashion as the D, Z, and * specifiers. The K specifier works equally well with string data or numeric data. String and numeric image specifiers may be combined in the same image statement.

Literal images may be enclosed in either single or double quotation marks (' ' or " ") when included in an IMAGE statement. You may include a literal image directly in an OUTPUT statement provided that you do not use the same form of quotation marks to enclose both the literal and the whole *output image.* Thus, the following statements could be used:

```
50 OUTPUT ':PR' USING ' "Total=",K ' ;X
80 OUTPUT ':PR' USING " 'Total=',K " ;X
```

However, the statement OUTPUT ':PR' USING ' 'Total=',K ' ;X results in an error because the computer is not able to distinguish the nested quotation marks.

The End-of-Line Sequence Image Specifier

The end-of-line sequence image specifier controls the output of end-of-line sequences. An end-of-line sequence consists of one or more characters that are normally output after the last item in an output list. The default end-of-line sequence of the HP-75 is a two-character sequence: a **carriage-return** followed by a **line-feed**. You can change the normal **carriage-return/line-feed** EOL (end-of-line) sequence to any desired sequence of up to three characters by using the `ENDLINE` statement. This command can be executed either manually or in a program and is described in the *HP-75 Owner's Manual*. If an EOL sequence is output, it will be the current EOL sequence set by you or your program with the `ENDLINE` statement. The end-of-line sequence image specifier does not alter the EOL sequence, but simply causes one to be output.

Note: In this manual **EOL sequence** refers to the current end-of-line sequence that you (or your program) have established with the `ENDLINE` statement, unless otherwise noted. The symbol **EOL** is used in the examples to indicate the EOL sequence.

Image Specifier	Meaning
/	Causes the output of an EOL sequence. Often used for skipping lines in a printout.

The / may be placed anywhere in the image list and may have a number before it to indicate how many EOL sequences are desired. A typical use of the / image is shown by the statement:

```
OUTPUT ':PR' USING 'K,4/,K';A$,B$
```

If the destination is a printer, A$ is printed, followed by four blank lines, then B$ is printed. If A$="HI", B$="JOE", the character sequence is output as follows:

H	I	EOL	EOL	EOL	EOL	J	O	E	EOL

You can suppress the output of the final EOL sequence by ending the `OUTPUT` statement with a semicolon (;). For example, a semicolon could be added at the end of the above statement:

```
OUTPUT ':PR' USING 'K,4/,K';A$,B$;
```

The resulting output follows:

H	I	EOL	EOL	EOL	EOL	J	O	E

The string `HI` is printed and four lines are skipped. The string `JOE` is **not** printed, but is transmitted to the printer's buffer.

Note: A reference list of all `OUTPUT` image specifiers is given in appendix B under `IMAGE`.

Formatted ENTER

Using ENTER statements with image specifiers gives you a high degree of control in two areas:

1. Accurately describing to the computer what the incoming data looks like and what should be done with it.
2. Precisely specifying what conditions constitute the end point of the ENTER statement itself.

This discussion deals with data formatting images first, then presents the terminator images. The HP-75 uses an enter image when some form of ENTER USING statement is encountered. There are two forms of this statement:

simplified syntax

10 IMAGE *enter image*
20 ENTER ':*device code*' USING 10;*enter list*

simplified syntax

ENTER ':*device code*' USING '*enter image*';*enter list*

The examples above show the general forms of the ENTER USING statement. Here are some specific examples:

```
10 IMAGE 2(A),K
20 IMAGE 5D,2X,3D
      ⋮
60 ENTER ':B2' USING 10;A$,B$,X
70 ENTER ':TP' USING 20;I,J
80 ENTER S2$ USING '%,8A,/,K';Q$,R$
90 ENTER ':TP' USING I$;N$,A
```

The general forms use the same type of symbols that were used to represent the OUTPUT statement. In the ENTER statement, *device code* stands for the device code of the device from which the data is to be entered, *enter image* for the list of image specifiers, and *enter list* for the list of variables to be entered. Note that the ENTER statement will accept only one device code, and that you may use string variables in place of the *device code* and/or *enter image* fields. As with simple ENTER statements, the *enter list* must contain either string or numeric variables. You can't enter into a constant.

Note: You may substitute a valid HP-IL device address for the *device code* field in an ENTER statement. If the intended source device has already been addressed to talk, you may leave the *device code* field blank. Refer to appendix B for a complete definition of ENTER statement syntax.

Data Images

The image specifiers in this group are used to indicate what the computer should do with the incoming stream of data. The basic choices are:

1. Use characters to build a numeric variable.
2. Place characters into a string variable.
3. Skip over a number of characters.

Note: A reference list of all ENTER image specifiers is given in appendix B under IMAGE.

Numeric Image Specifiers. These specifiers are used to control the input of numeric characters, including digits, sign, exponent, and punctuation. You may precede any of these specifiers (except K) with a number from 1 to 255. In an ENTER image 5D and DDDDD both mean "enter five characters to be used in building a number."

Image Specifier	Meaning
d,D z,Z * . s,S m,M	These specifiers all accept one character to be used in building a number. The incoming characters do not have to follow the specified format, there just has to be the right number of characters. The six different specifiers are provided so that your program can document the expected format of the characters, and so that ENTER and OUTPUT statements can share the same IMAGE statement, if desired.
c,C	This specifier also accepts one character to be used in building a number. However, if a C is present **anywhere** in a number's image, all commas will be ignored while the number is being entered. Without this specifier, a comma would terminate numeric entry.
e,E	Accepts five characters to be used in building a number. The five characters may be exponent information, but do not have to be.
k,K	Enters data into a numeric variable using free-field format (explained in section 2).
r,R	Accepts one digit and treats all commas (,) as radix symbols (to accept numeric input in European format).
p,P	Accepts one digit and ignores all periods (to accept numeric input in European format).

String Image Specifiers. These specifiers are used to enter characters into string variables. You may precede the A specifier (but not the K) with a number from 1 to 255. In an ENTER image 4A and AAAA both mean "enter four characters into a string variable."

Image Specifier	Meaning
a,A	Enters one character into a string variable.
k,K	Enters data into a string variable using free-field format (explained in section 2).

Some examples are in order. Suppose the following character sequence is received by the computer:

1	2	3	4	H	E	L	L	O	EOL

Either of the following ENTER statements can be used to enter a numeric variable followed by a string variable:

```
ENTER ':TP' USING '4D,5A';X,Y$
ENTER ':TP' USING 'Z.DD,5A';X,Y$
```

Notice that any numeric image that accepts four characters will properly enter the 1234. String data can be entered with an nA image if n (the number of characters) is known, or with a K if the number of characters is unknown.

Suppose instead that the incoming data was:

1	,	2	3	4	H	E	L	L	O	EOL

The ENTER image would now have to include a C for the entire **1234** to be entered. For example:

```
ENTER ':TP' USING 'C4D,K';X,Y$
ENTER ':TP' USING 'DDDDC,5A';X,Y$
```

Notice that the C does not have to appear at the same place in the image as the comma does in the incoming data. However, the comma is counted as a character.

Skipping Unwanted Characters. The following specifiers can be used with incoming numeric or string data to skip over any characters that you do not want to include in the input. You may precede the X specifier with a number from 1 to 255. In an ENTER image 3X and XXX both mean "skip three spaces."

Image Specifier	Meaning
x,X	Causes one character to be skipped.
/	Causes the computer to skip characters until the next terminator is received. The normal terminators are the current EOL sequence (defined with the ENDLINE statement) or the End Byte message.

The X specifier should only be used when you have a good understanding of the structure of the incoming data, but can be very useful in formatting operations. For example, suppose that text is being entered from a remote computer that sends a line number at the beginning of every string. You know that the line number information always appears in the first eight characters of each string, and you don't want these line numbers in your data. The following format could be used to strip off the line numbers:

```
ENTER ':TP' USING '8X,K';A$
```

The / specifier is used to demand a terminator (either the current EOL sequence or an End Byte message) before going on to the next variable. To see the effect of this specifier, assume that the incoming data is as follows:

1	2	3	H	I	EOL	B	Y	E	EOL

Note: The normal terminators are the current EOL sequence and the End Byte message. The / specifier will cause the ENTER statement to skip to whichever terminator occurs first. The operation of this specifier is affected by the use of terminator images (refer to the following subsection). If you have used a terminator image to redefine the active terminators, the / specifier will cause a skip to the first recognized terminator.

Using the statement:

```
ENTER ':TP' USING '3D,K';Y,A$
```

causes Y to get the value 123 and A$ the value HI. However, if the statement:

```
ENTER ':TP' USING '3D,/,K';Y,A$
```

is used, then Y gets the value 123 and A$ becomes BYE. The / specifier causes the computer to skip all characters after Y is satisfied until it receives the EOL sequence. The entry into A$ begins with the first character after the EOL sequence. Without the / specifier, the entry into A$ begins as soon as the 3D field is exhausted.

Terminator Images

Terminators (normally the current EOL sequence and the End Byte message) serve in two roles for the ENTER statement. If a terminator is received in a field of data (before the variable is otherwise satisfied), it will serve as a **field terminator** and will terminate entry into the variable. The ENTER statement will begin entry into the next variable. Once all variables have been satisfied, a terminator will serve as a **statement terminator** and will terminate the ENTER statement. Indeed, a statement terminator is normally **required** in order to go on to the next statement in the program. The terminator that terminates the ENTER statement can be the same one that satisfied the last variable. Note that terminators are not required to satisfy a variable. Data entry into a variable can be ended by satisfying an image list, by filling a dimensioned string variable, or by the free-field entry of a trailing blank or non-numeric character into a numeric variable.

You can redefine the active terminators by using a **terminator image**. By using the appropriate terminator image specifier, you may eliminate the current EOL sequence, the End Byte message, or both as statement terminators. You may also establish the ETO (End Of Transmission — OK) message as a terminator. The terminator image specifiers, and their various combinations, are listed in the following table:

Image Specifier	Meaning
#	Eliminates the current EOL sequence as a terminator. When this specifier is present, the ENTER statement terminates only on an End Byte message.
!	Eliminates the End Byte message as a terminator. The ENTER statement terminates only on an EOL sequence.
%	Establishes the ETO (End Of Transmission — OK) message as a terminator. The ENTER statement terminates on an ETO message, End Byte message, or an EOL sequence.
#! or !#	Both the current EOL sequence and the End Byte message are eliminated as terminators. No terminator is required. The ENTER statement terminates when the last variable is satisfied.
#% or %#	Eliminates the current EOL sequence as a terminator, but establishes the ETO message. The ENTER statement terminates on an ETO message or an End Byte message.
!% or %!	Eliminates the End Byte message as a terminator, but establishes the ETO message. The ENTER statement terminates on an ETO message or an EOL sequence.
#!% (any order)	Eliminates EOL sequence and End Byte message as terminators. ENTER statement terminates only on an ETO message.

Most data entry situations do not require the use of terminator images. If you are entering data from a device that outputs the **carriage-return** and **line-feed** characters after each data item, the ENTER statement will terminate on this EOL sequence (provided that **carriage-return/line-feed** is the current EOL sequence). In most other cases, the ENTER statement will correctly terminate when an End Byte message is received. Normally, it is not necessary to specify which terminator to use, since the ENTER statement will terminate on the first one received. However, terminator images do give you the flexibility to handle certain specialized applications.

If you want the ENTER statement to terminate only on an End Byte message, you can suppress the current EOL sequence as a terminator by including the # specifier at the begining of the image list. The following statement will terminate **only** when an End Byte message is received:

```
ENTER ':E1' USING '#,K,5D';A$,B1
```

> **Note:** Terminator image specifiers must be listed first in the ENTER image list (before the first comma). You cannot precede them with a number.

The ! specifier suppresses the End Byte message as a terminator. The following statement will terminate **only** when the current EOL sequence is received:

```
ENTER ':E2' USING '!,4D,5A';X,Y$
```

Eliminating the Statement Terminator Requirement. Normally, the `ENTER` statement must see the current EOL sequence or an End Byte message at the end of the incoming data before the program can go on to the next statement (the ETO message may be specified as an alternative terminator). If there is no statement terminator at the end of the data, a record overflow error will result. You can use the `#!` (or `!#`) image specifier to eliminate the requirement for a statement terminator. This specifier eliminates the EOL sequence and End Byte message as terminators, and causes the `ENTER` statement to terminate when the last variable is satisfied. In the following example, the `ENTER` statement terminates after the variable Y is satisfied.

```
ENTER ':E1' USING '#!,4D,6D';X,Y
```

If 10 numeric characters are received, the two variables are satisfied and the statement terminates.

> **Note:** The `K` and `/` specifiers override the `#!` (or `!#`) specifier. If a `K` or `/` is present in an `ENTER` image, a terminator is required **for that field.**

Using the ETO Message As a Statement Terminator. If you are unable to use either an EOL sequence or an End Byte message to terminate an `ENTER` statement, you may use the `%` specifier to establish the ETO (End Of Transmission — OK) message as an alternative statement terminator. The following statement terminates when an EOL sequence, End Byte message, **or** an ETO message is received:

```
ENTER ':D3' USING '%,K,5A';A$,B$
```

You may combine the `#` or `!` specifiers with `%` to suppress the EOL sequence or End Byte message as a terminator, while establishing the ETO message. The following statement will terminate on either an End Byte message or an ETO message:

```
ENTER ':D3' USING  '#%,K,5A';A$,B$
```

If you want **only** an ETO message to terminate your statement, specify `#!%`:

```
ENTER ':D3' USING '#!%,K,5A';A$,B$
```

There's Always an Exception. Not all terminator problems are a proper job for terminator images. Consider the example of a name field (string) followed by an age field (numeric). Suppose that the names are variable in length and separated from the age by a comma. If the age came first, this would not be a problem since the comma would end the entry into the numeric variable. But since the string data is entered first in this example, the task is a bit trickier. You could input the entire record into a temporary string variable, then use the `POS` function and string subscripts to extract the name and age fields. This hypothetical situation emphasizes the importance of knowing the nature of the data you are trying to enter. Some problems are handled by terminator images, and some are solved by different means, but all require thought by the programmer.

Changing the Size of the ENTER Buffer

The ENTER statement receives data into a reserved area in memory called the ENTER buffer. This buffer is also used by other statements that enter data (for example, ENTIO$ and ADDRESS). The default size of this buffer is 256 bytes. Thus, the ENTER statement reads up to 256 bytes into this buffer, then places this data into the appropriate variables when the statement is terminated. You can change the size of the ENTER buffer with the IOSIZE statement. If an ENTER statement receives more than 256 bytes (or the size set with IOSIZE) before a terminating condition is reached, an error will result.

The IOSIZE statement allows you to set any ENTER buffer size from 1 to 24,575 bytes. The general form of this statement is:

IOSIZE *buffer size*

where *buffer size* is a number from 0 to 24,575 (a zero or negative value sets the default size of 256 bytes). You should set IOSIZE to be at least the maximum expected record size plus one byte.

A Word of Advice About Images

Choosing the proper image for your application can often mean the difference between success and failure for your program. However, considering the wide range of peripheral devices and the near-infinite variety of possible data formats, it is understandably difficult to pick just the right image. Even experienced programmers will go through a period of trial-and-error before finding the perfect combination of image specifiers.

There is an old, but true, saying in the world of computers: "You can't program a computer to do something that you don't know how to do yourself." This is an appropriate sentiment for formatted I/O. If you don't know exactly what character sequence needs to be output or what an incoming sequence contains, it is very unlikely that you will know eactly what image specifiers to use.

Deciding on an exact character sequence for an output is simply a matter of definition. You know what data is generated by your program, so all you need to do is pick a desirable form for its output. The primary caution here is to avoid image overflow conditions.

But how can you determine the exact nature of the incoming data when you can't get it into the computer to study? Fortunately, there is a way to inspect a totally unknown character sequence. Any sequence of bytes, including potential terminators, can be entered with the #!,nA image (where n is the number of characters to read). For example, the statement:

```
ENTER ':D1' USING '#!,10A';A$
```

will read 10 bytes as the equivalent ASCII characters. You may then use the HEX$ function (refer to appendix D) to convert these ASCII characters to a hexadecimal representation. Once you know the exact nature of the incoming data, the job of choosing image specifiers will be much simpler.

Section 4

Sending and Receiving HP-IL Messages

The HP-75 I/O ROM provides enhanced versions of the SENDIO statement and ENTIO$ function that are compatible with the SENDIO and ENTIO$ of the *HP-75 I/O Utilities Solutions Book.* A SEND statement, similar in syntax to the HP Series 80 SEND statement, is also provided for software compatibility. All three instructions enable you to source individual HP-IL messages. The SENDIO statement allows you to send commands and data to specified HP-IL devices. The ENTIO$ function allows you to send commands to a specified device and return data as the value of the function. The SEND statement allows you to send any HP-IL message. To use SENDIO, ENTIO$, and SEND successfully, you must follow HP-IL protocol. A full discussion of HP-IL protocol is beyond the scope of this manual. Refer to the following sources for a complete discussion of HP-IL protocol:

- Kane, Gerry, et al. *THE HP-IL SYSTEM: An Introductory Guide to the Hewlett-Packard Interface Loop.* Osborne/McGraw-Hill, Berkeley, California, 1982.
- Hewlett-Packard Company. *The HP-IL Interface Specification.* HP part number 82166-90017, 1982.

The SENDIO Statement

The SENDIO statement is used to send commands and data to HP-IL devices. SENDIO can be issued from the HP-75 keyboard or executed in a BASIC program. The general form of this statement is:

simplified syntax

SENDIO ':*device code*' , '*command list*' , '*data list*'

The three parameters are string expressions. The *device code* parameter is a list of one or more device codes, each representing a device that will receive HP-IL commands or data. The *command list* is a list of HP-IL commands to be executed, separated by commas. The commands may be specified in the form of HP-IL command mnemonics. The commands that you may use in a SENDIO *command list* are listed in appendix C. The *data list* is a character string to be transmitted as data. Any of the three parameters may be specified with either a literal enclosed in quotation marks or the name of a string variable that contains the quoted string. A complete definition of the syntax of the SENDIO statement is given in appendix B.

Most of the time, SENDIO will be used to activate a device as a listener. The device to be activated can be specified with either the *device code* parameter or the *command list:*

- Use the *device code* parameter when you know what device code has been given to the intended device. You can specify one or more device codes in this parameter (for example: ':D1' or ':PR,:TV'). You can send a LAD (Listen Address) message to the specified device(s) either by leaving the *command list* null, or by specifying LAD# in the *command list*. (Only one LAD# command is needed, even if more than one device code is specified.) LAD# can be used in combination with other HP-IL commands, and it may appear anywhere in the *command list.*
- Use the *command list* when the HP-IL address of the intended device is known. To do this, specify LADn, where n is the HP-IL address of the device. This will cause a LAD message to be sent to device n regardless of what appears in the *device code* field. You may have any number of LADn commands within a single SENDIO statement, and you may have both LADn and LAD# in the same SENDIO.

The following SENDIO statement sends the string HELLO to the devices named D1 and D2, and also to the devices with addresses 5 and 6:

```
SENDIO ':D1,:D2','LAD#,LAD5,LAD6','HELLO'
```

It is not necessary to supply values for all three parameters. If you wish to omit a parameter, you must specify a null string. The following example of SENDIO sends no commands, but sends the string DATA to any devices in the loop that already have active listener status:

```
SENDIO '','','DATA'
```

You may substitute the name of a string variable for any of the three parameters, as long as you have defined the variable. In the following example, the SENDIO statement sends the string DATA to the devices named PR and TV. (Leaving the *command list* null generates a LAD# command.)

```
10 A$ = ':PR,:TV'
20 SENDIO A$,'','DATA'
```

The SENDIO statement processes parameters from left to right. Processing proceeds as follows:

1. If the *device code* parameter has been specified, SENDIO determines the HP-IL address of the specified device. This device address is used when processing the *command list.* If more than one device is specified in the *device code* field, SENDIO determines the address of each device. If the *device code* field is null, then no action is taken in this step.
2. The *command list* is processed. Commands are sent one at a time through the loop. RFC (Ready For Command) messages are sent automatically after each command.
3. After all commands are sent, the data specified in the *data list* is sent around the loop, one character at a time. If a listener device sends an NRD (Not Ready For Data) message, transmission of data is terminated. You can recover from this condition by using the SEND? function (refer to the subheading "Resuming Data Transmission With SEND?").
4. After all commands and data have been sent, the UNT (Untalk) and UNL (Unlisten) messages are sent around the loop, deactivating all talker and listener devices. If you want the talker and listener devices to remain active, you can suppress the automatic UNT/UNL by including a TL+ anywhere in the *command list.*

You can use SENDIO to send HP-IL commands around the loop without sending data. For example, you can use the following SENDIO statement to address the loop:

```
SENDIO '','AAU,AAD1',''
```

The AAU (Auto Address Unconfigure) command clears all device addresses in the loop, then the AAD1 (Auto Address) command automatically readdresses the devices in the loop starting with address 1. AAD1 should appear last in the *command list.*

Resuming Data Transmission With SEND?

If a device in the loop sends an NRD message while SENDIO is transmitting data, the transmission terminates. You can resume transmission from the point of interruption by using the the SEND? function.

SEND? is a function that requires no parameters. It returns an integer value representing the position in the *data list* of the character **after** the last one that was successfully sourced in the last SENDIO statement. If the *data list* in the last SENDIO was null, or if the last SENDIO was successfully completed, SEND? returns a 0. (If a device in the loop sends an NRD message after the last character was sent, SEND? will return a value equal to the length of the string plus one).

The following program is an example of how to use SEND?. The program will send the characters I love my HP-75 to the fourth device in the loop:

```
10 A$ = 'I love my HP-75'
20 SENDIO '','LAD4',A$
30 IF SEND? = 0 THEN GOTO 50
40 SENDIO '','LAD4',A$[SEND?]
50 END
```

If the first SENDIO (statement 20) successfully transmits the entire string, SEND? will return a value of zero. This will cause a branch to statement 50, completing the program. Suppose that an NRD message is received after the SENDIO in statement 20 sends the m in my. SENDIO will stop transmitting at this point. The SEND? function returns a value of 9, since the m is the eighth character in the data list (and the last one successfully sent). In this case, statement 40 is executed before the program ends. In statement 40, SENDIO sends a substring of A$ that starts at the ninth position. The substring has the value y HP-75.

If the SENDIO in statement 20 successfully sends the entire string and the device in the loop then sends an NRD message, the value of SEND? will be the length of the string plus one. Statement 40 will be executed, but will send the null string. Thus, the program sends the complete string I love my HP-75 in any event.

SENDIO Restrictions

SENDIO causes the HP-75 to become active as a talker. Therefore, although it is possible to issue TAD (Talker Address) commands with SENDIO, doing so will cause more than one talker to become active in the loop. You should **not** use SENDIO to address devices as talkers since this will result in a deadlock condition.

If DISPLAY IS or PRINTER IS devices have been assigned for the HP-75, the talkers will automatically be deactivated even if TL+ is specified in the *command list.* Although TL+ will stop SENDIO from automatically deactivating listeners, HP-75 I/O operations not related to SENDIO may cause deactivation when DISPLAY IS or PRINTER IS devices are in use.

The ENTIO$ Function

The ENTIO$ function is used to receive data from other HP-IL devices. In contrast to SENDIO, which is a **statement**; ENTIO$ is a **function**, and returns a character string value. The string returned is the data transmitted by the specified HP-IL device. The general form of the ENTIO$ function is:

simplified syntax

ENTIO$(' :*device code* ', '*command list*')

The two parameters are string expressions. The *device code* parameter is a list of one or more device codes. The *command list* consists of one or more HP-IL commands, separated by commas. The commands may be specified in the form of HP-IL command mnemonics. The commands that you may use in an ENTIO$ *command list* are listed in appendix C. Both parameters may be specified with either a literal enclosed in quotation marks or the name of a string variable that contains the quoted string. You may specify the null string for either of the parameters, but not both. A complete definition of ENTIO$ syntax is given in appendix B.

Most of the time, ENTIO$ will be used to activate a device as a talker. The device to be activated can be specified with either the *device code* parameter or the *command list:*

- Use the *device code* parameter when you know what device code has been given to the device. You can talk or listen address the specified device by including TAD# or LAD# in the *command list.* If you leave the *command list* null, TAD#, SDA is automatically generated. The TAD# and LAD# commands may be used in combination with any other HP-IL commands, and may appear anywhere in the *command list.* If TAD# is specified in the *command list*, only one device code may be specified (otherwise an error will result).
- Use the *command list* when the device's HP-IL address is known. To do this, specify TADn or LADn, where n is the HP-IL address of the device. This will send a TAD or LAD message to device n regardless of what appears in the *device code* field. Both TADn and LADn may be used in conjunction with other HP-IL commands within a single ENTIO$ instruction. You may also combine TADn or TAD# with LADn or LAD# in the same ENTIO$.

The following example shows how you might use the ENTIO$ function in a BASIC statement:

```
70 A$ = ENTIO$(':D1','TAD#,SDA')
```

The ENTIO$ function addresses the device named D1 as the talker, then sends an SDA (Send Data) message. The data sent by device D1 is returned by the ENTIO$ function as the value of A$.

With ENTIO$, either the *device code* parameter or the *command list* may be null, but not both. If null strings are specified for both parameters, an error results (see appendix E).

ENTIO$ processes parameters from left to right, as does SENDIO. Note, however, that ENTIO$ does not have a data field. This is because ENTIO$ causes the HP-75 to become active as a controller and a listener only; it can transmit commands and receive data, but it cannot send data. Processing proceeds as follows:

1. If the *device code* parameter has been specified, ENTIO$ determines the device addresses in the loop. These device addresses are used when processing the *command list.* If the *device code* field is null, then no action is taken in this step.
2. The *command list* is processed. Commands are sent one at a time through the loop. RFC (Ready For Command) messages are automatically sent after each command.
3. Data is collected from the loop. The value returned by the ENTIO$ function will be the data collected in this step. Data collection terminates when one of the following conditions is met:
   - An End Of Transmission message is received. The ETO (End Of Transmission — OK) message will terminate data collection unless an ET- command is included in the *command list.* The ETE (End Of Transmission — Error) message will always result in termination.
   - The number of Data Byte messages exceeds the limit set with the SZ= command. The default value is either 256 bytes or the value set with IOSIZE. The HP-75 sources an NRD message if the limit is exceeded.
   - A logical end-of-record character or sequence is received. If this occurs, an NRD message is sourced. Refer to the subheading "Defining Logical End-of-Record" for more details.

   End-of-line sequences are treated as data by ENTIO$. If EOL sequences are received, they are included in the string returned by the ENTIO$ function.
4. UNT (Untalk) and UNL (Unlisten) messages are sent around the loop to deactivate all talker and listener devices. If you want the talkers and listeners to remain active, you can suppress the automatic UNT/UNL by including the TL+ command in the *command list.*

The SZ= command is used to set the maximum number of bytes that the ENTIO$ function will read. If no SZ= command is included in the *command list*, the maximum number of bytes will be the current size of the ENTER buffer. The default size is 256 bytes. You can set the size of the ENTER buffer to any value from 1 to 24,575 bytes with the IOSIZE statement (refer to section 3). If a SZ= command is included in an ENTIO$ *command list*, the specified size overrides the ENTER buffer size set with IOSIZE **for that** ENTIO$ **only**. The maximum size that you may set with the SZ= command is 32,767 bytes (unless DA- is also specified in the *command list*). The syntax is SZ=XXXXX where XXXXX is a decimal number in the range 1 to 32767.

The DA- command prevents the ENTIO$ function from reading any data into the computer. ENTIO$ returns the null string if DA- is included in the *command list*; however, data will be transmitted from the talker to any active listeners in the loop. If SZ= is not specified, the maximum number of bytes transmitted will be the current value of IOSIZE (default = 256). If both DA- and SZ= are included in the *command list*, sizes up to 999,999,999 bytes may be set. The syntax is SZ=XXXXXXXXX where XXXXXXXXX is a number in the range 0 to 999999999. If SZ=0 is specified, an unlimited number of bytes will be transferred from the talker to any active listeners. SZ=0 cannot be specified unless DA- is also specified.

An example may clarify this. In the following statement ENTIO$ addresses device 1 as the talker and devices 2 and 3 as listeners, then causes the talker to send its bytes to the listeners:

```
120 B$ = ENTIO$ ('','TAD1,LAD2,LAD3,DA-,SZ=0,SDA')
```

The SZ=0 command negates the size limit on the number of bytes to be read. The DA- command causes ENTIO$ to return no data (the null string is returned for B$). Thus, the SDA command in the above statement causes the talker to send as many bytes as it has to send, and listeners 2 and 3 to receive the transmitted data.

Defining Logical End-of-Record

You can define a character or sequence of characters to serve as a logical end-of-record during transmission. When the logical end-of-record is received, transmission will be terminated. The data that has been collected up to the point of termination will be returned by ENTIO$. You can define the logical end-of-record by including one of the following commands in the ENTIO$ *command list*:

TR$ You can specify the current EOL sequence as a logical end-of-record by including TR$ in the *command list.*

TR: You can specify any ASCII character as a logical end-of-record by including TR:XX in the *command list*, where XX is the hexadecimal representation of the ASCII character number (you cannot specify a null value for XX).

TR[You can specify any desired string of up to six characters as a logical end-of-record by including TR[*string*] in the *command list.* Note that the string is delimited with brackets rather than quotation marks. You cannot include the] character in the string. If the string includes quotation marks, they must not be the same form (single or double) that is used to delimit the *command list* itself.

TR! You can use the End Byte message as a logical end-of-record by including TR! in the *command list.*

Here is an example of how you might use logical end-of-record: Suppose that the data you are receiving consists of lines of text with a **line-feed** character separating each line. Rather than having ENTIO$ return 256-character strings with embedded **line-feed** characters, you may wish to treat each text line as a logical record. To accomplish this, you would simply include TR:0A within the command list. This command establishes the **line-feed** character (ASCII decimal code 10, hexadecimal 0A) as the logical end-of-record. Each time ENTIO$ is executed, it will return a string containing just one line of text. The **line-feed** character will be included in the string.

Enhanced Printing Control

You can have an EOL sequence inserted into the data string automatically each time an End Byte message is received from the talker. If you include a CL+ command in the *command list*, a **carriage-return/line-feed** sequence will be inserted after each End Byte message. If you use the EL+ command instead, the current EOL sequence (established with the ENDLINE statement) will be inserted. Suppose that you want to receive readings from an HP-IL device that transmits Data Byte messages followed by End Byte messages, then print the readings on a printer. If these transmissions were printed as received, the readings would all be on one line with no spacing. Specifying EL+ will cause the current EOL sequence to be inserted after each reading, thus allowing each reading to be printed on a separate line.

ENTIO$ Restrictions

The ENTIO$ function will return the null string unless either SDA, SST, SDI, SAI, AADn, or ID:00 appears as the last command in the *command list.* These commands should not appear in the *command list* except as the last command. If one of these commands occurs as other than the last command, it will cause the transmission to begin, but the transmission will be terminated after **one** message is sent.

If DISPLAY IS or PRINTER IS devices have been assigned for the HP-75, the talker will automatically be deactivated even if TL+ is specified in the *command list.* Although TL+ will stop ENTIO$ from automatically deactivating listeners, HP-75 I/O operations not related to ENTIO$ may cause deactivation when DISPLAY IS or PRINTER IS devices are in use.

The SEND Statement

Most I/O applications can be performed most easily by using either the OUTPUT and ENTER statements, or SENDIO and ENTIO$. However, the HP-75 I/O ROM also provides the SEND statement, which allows you to send **any** HP-IL message or sequence of messages. This provides enhanced capability for the advanced user. The syntax of the SEND statement appears to be rather complex due to its versatility:

simplified syntax

```
SEND [[ CMD byte number ] [ DATA byte number  [EOL] ] [ END byte number  [EOL] ]
            byte string  ]        byte string  ]          byte string

        [IDY byte number] [RDY byte number] [DDL byte number] [DDT byte number]

        [SAD byte number] [LISTEN byte number] [TALK byte number]

        [GTL] [RMO] [NRE] [LLO] [CIF] [LPD] [MLA] [MTA][SDC] [UNL] [UNT]]...
```

The SEND statement enables the HP-75 to source individual HP-IL messages. You can send any combination of the bracketed items listed in the above syntax representation, *in any order* (consider the representation to be one continuous line). Since the SEND statement deals with individual messages, a discussion of HP-IL messages and how to specify them follows.

Each HP-IL message is defined by 11 bits: three **control bits** and eight **data bits**. HP-IL messages are separated into four groups according to their control bits:

- Command group: These messages convey instructions from the controller and are monitored by all HP-IL devices (including idle devices).
- Ready group: These messages provide special-purpose communication between the controller and one or more devices, and are generally used to coordinate the transfer of instructions and data.
- Identify group: These messages enable devices to request service from the controller. Any device can modify these messages to indicate a service request condition to the controller.
- Data/end group: These messages convey data between active devices (possibly including the controller). Any device can modify these messages to indicate a service request condition to the controller.

The SEND statement allows you to specify messages from each of these four groups by including the appropriate message indicators and qualifiers. An example of a message **indicator** is CMD, which indicates a command message. Message **qualifiers** specify a specific message, and include the *byte number* and *byte string.*

Sending Command Group Messages

Certain command message indicators — GTL, RMO, NRE, LLO, CIF, LPD, MLA, MTA, SDC, UNL, and UNT — require no qualifiers. You may include any combination of these indicators in a SEND statement, and you may include them in combination with other indicators. These indicators (except CIF, RMO, MLA, and MTA) cause the SEND statement to send the HP-IL commands with the corresponding mnemonics (refer to appendix C). The CIF indicator causes SEND to send the IFC (Interface Clear) message. The RMO indicator causes SEND to send the REN (Remote Enable) message. The MLA indicator causes SEND to send no message, while MTA causes SEND to send the UNT message. In the following example, the SEND statement sends the HP-IL command messages UNT (Untalk), UNL (Unlisten), and REN (Remote Enable):

```
30 SEND UNT UNL RMO
```

> **Note:** The HP-75 automatically sends an RFC (Ready For Command) message after each command message sent by the SEND statement.

You may specify **any** HP-IL command message with the CMD message indicator. The specific command is indicated by either a *byte number* or *byte string.* A CMD *byte number* is a number in the range 0 through 255 (modulo 256) that represents the eight **data bits** of the command message. The *byte number* for the NRE (Not Remote Enable) message is 147, representing the bit pattern "10010011". The following SEND statement sends the NRE message:

```
70 SEND CMD 147
```

You may specify more than one command *byte number* in a CMD field, separating the successive numbers with commas. The following statement sends the UNT and UNL messages (UNT is command number 95 and UNL is command number 63):

```
90 SEND CMD 95,63
```

You may also use a *byte string* to specify a series of HP-IL commands in a CMD field. Each ASCII character in a *byte string* indicates the command that has the *byte number* equivalent to its ASCII decimal code. The following statement also sends the UNT and UNL messages:

```
110 SEND CMD '_?'
```

The underscore (_) has ASCII decimal code 95, representing the UNT message. The question mark (?) has decimal code 63, representing the UNL message. Note that capital and lower case letters specify different bytes when used in a *byte string*. You may use the CHR$ function to include characters that cannot be generated directly from the keyboard.

The DDL and DDT message indicators may be used to specify Device-Dependent Listener and Device-Dependent Talker messages having number 0 through 31 indicated by *byte number* (modulo 32). More than one *byte number* may be specified in a DDL or DDT field.

The SAD message indicator is used to specify a Secondary Address message having an address in the range 0 through 31 indicated by byte number (modulo 32). More than one *byte number* may be specified in an SAD field.

The LISTEN message indicator is used to specify LADn (Listen Address) messages. Addresses are indicated by byte numbers in the range 0 through 31 (modulo 32). The device at the specified address becomes a listener — except that 31 clears all devices from listener status. More than one LADn message may be specified in a LISTEN field. The following SEND statement sets up the devices at addresses 2, 3, and 5 to listen:

```
50 SEND UNT UNL LISTEN 2,3,5
```

You can now send the string ABC to these devices with the following OUTPUT statement:

```
60 OUTPUT ; 'ABC'
```

The HP-75 automatically becomes the talker when the OUTPUT statement is executed. You need not specify device codes in the OUTPUT statement since you have already addressed the intended devices to listen.

The TALK message indicator is used to specify a TADn (Talk Address) message. The address n is indicated by a *byte number* in the range 0 through 31 (modulo 32). The device at the specified address becomes a talker — except that 31 clears all devices from talker status. Only one TADn message may be specified in a TALK field. The following SEND statement addresses device 3 as the talker:

```
30 SEND UNT UNL TALK 3
```

You may now enter data from device 3 with the ENTER statement. To enter data as a string:

```
40 ENTER ; A$
```

The HP-75 automatically becomes a listener when the ENTER statement is executed. You need not include a device code in the ENTER statement since the intended device has already been addressed to talk. Once the ENTER statement is completed, you should remove talker status from device 3 with UNT or MTA.

Note: You should be careful when using the SEND statement to address talkers. The HP-75 will automatically become a talker when you execute an OUTPUT or PRINT statement. If a device in the loop has been addressed as a talker with SEND, there will be two active talkers.

Sending Ready and Identify Group Messages

Ready group messages are specified with the RDY message indicator. Identify group messages are specified with the IDY message indicator. In either case the message sent will have the data bits set according to a *byte number* in the range 0 through 255 (modulo 256). More than one *byte number* may be specified in an RDY or IDY field.

Sending Data/End Group Messages

Data/End group messages are specified with the DATA and END message indicators. You may use either a *byte number* field or a *byte string* to specify the actual Data Byte message or End Byte message. The *byte number* field may contain several byte numbers each indicating the ASCII character code of one character in a string. Byte numbers have the range 0 through 255 (modulo 256). A *byte string* results in a series of Data Byte messages that transfer the characters defined by the string. The following statements both send the Data Byte messages that transfer the string ABC (A, B, and C have the ASCII decimal codes 65, 66, and 67):

```
40 SEND DATA 65,66,67
70 SEND DATA 'ABC'
```

The inclusion of an EOL indicator in a DATA or END field causes the current EOL sequence (defined with the ENDLINE statement) to be transmitted as a sequence of Data Byte messages. The following statement addresses device 2 as a listener, sends the string HELLO, and sends the current EOL sequence:

```
90 SEND UNT UNL LISTEN 2 DATA 'HELLO' EOL
```

If device 2 is a printer, the EOL sequence will normally cause HELLO to be printed (provided the current EOL sequence is **carriage-return/line-feed**).

Appendix B gives a complete definition of the syntax of the SEND statement.

Application Programs

The following programs exemplify some typical I/O applications using `OUTPUT`, `ENTER`, `SENDIO`, and `ENTIO$`.

An HP-75/HP Series 80 Interface

The following programs allow you to set up an interface between the HP-75 and an HP Series 80 Personal Computer using HP-IL. The HP Series 80 computer must have an HP-IL module and an I/O ROM installed. The Series 80 HP-IL module must be set in the non-controller mode and have a select code of 9. There are two programs involved: one for the HP-75 and one for the HP Series 80 machine. The programs assume that the HP Series 80 machine has been assigned the device code `C1`.

Instructions:

1. Key in each program to the appropriate machine.
2. Run the programs concurrently.
3. The HP-75 starts out as the talker, the HP Series 80 as the listener.
4. The prompt `MESSAGE :` will appear on the display of the talker.
5. Key in the message to be sent and press the return key. The message will appear on the display of the listener.
6. To exchange the talker and listener functions, precede the message with a `*`.
7. To stop the programs, precede the message with a `\`.
8. Go to step 4 unless the last message began with a `\`.

HP-75 Program Listing:

```
10 DIM A$[256]                              Dimensions the string.
20 INPUT 'MESSAGE : '; A$                   Inputs message.
30 OUTPUT ':C1' ; A$                        Sends message.
40 IF A$[1,1]='*' THEN 70                   Change talkers?
50 IF A$[1,1]='\' THEN END                  Terminate communications?
60 GOTO 20
70 ENTER ':C1' ; A$                         Enters message.
80 DISP USING 120 ; A$                      Displays message.
90 IF A$[1,1]='*' THEN 20                   Change talkers?
100 IF A$[1,1]='\' THEN END                 Terminate communications?
110 GOTO 70
120 IMAGE 'HP SERIES 80-->HP-75 :',K
130 END
```

HP Series 80 Program Listing:

`10 DIM A$[256]`	Dimensions the string.
`20 ENTER 9;A$`	Enters message.
`30 DISP USING 130 ;A$`	Displays message.
`40 IF A$[1,1]="*" THEN 70`	Change talkers?
`50 IF A$[1,1]="\" THEN END`	Terminate communications?
`60 GOTO 20`	
`70 DISP "MESSAGE : ";`	
`80 INPUT A$`	Inputs message.
`90 OUTPUT 9;A$`	Sends message.
`100 IF A$[1,1]="*" THEN 20`	Change talkers?
`110 IF A$[1,1]="\" THEN END`	Terminate communications?
`120 GOTO 70`	
`130 IMAGE "HP-75-->HP SERIES 80 :",K`	
`140 END`	

An HP-75/Modem Interface

This program allows communication between the HP-75 and another mainframe through an HP-IL modem. The HP-75 functions as though it is a terminal while the program is running. The program assumes that the device code MO has been assigned to the modem.

Instructions:

1. Turn on the modem.
2. Dial the number for the computer on the telephone.
3. Place the phone handset into the modem.
4. When the carrier light comes on, run the program.
5. The HP-75 now functions as a terminal. From this point on, the procedure depends on the computer to which you are connected. Do what you would normally do to communicate with the computer from a terminal.

Program Listing:

`10 WIDTH INF`	Sets large width.
`20 CLEAR ':MO'`	Clears the modem buffers.
`30 SENDIO ':MO','UNL,REN,LAD#','`*parameters*`' @ SENDIO ':MO','NRE',''`	Remote enables the modem.
`40 K$=KEY$ @ IF K$ #'' THEN GOSUB 80`	Gets the key.
`50 E$=ENTIO$(':MO','UNL,TAD#,SDA')`	Gets input from modem.
`60 DISP E$;`	Displays input.
`70 GOTO 40`	
`80 SENDIO ':MO','UNL,LAD#',K$`	Sends the key.
`90 RETURN`	

Note: The *parameters* field in line 30 of the program is used to specify the parity and protocol for your application. Refer to your modem manual for further information.

Obtaining Readings From a Multimeter

In this program the HP-75 triggers the HP 3468A Multimeter to take 10 voltage readings (one every 10 seconds), receives the data from the multimeter as a string, and outputs each voltage reading to the printer. The program assumes that the device codes `E1` and `P1` have been assigned to the multimeter and the printer, respectively.

Instructions:

1. Turn on the multimeter, printer, and HP-75. Assign the appropriate device codes.
2. Run the program.

Program Listing:

`10 REMOTE ':E1'`	Sets meter to remote mode.
`20 FOR F = 1 TO 10`	
`30 SENDIO ':E1','LAD#','F1RAT2'`	Sets meter to read voltage.*
`40 A$ = ENTIO$(':E1','TAD#,SDA')`	Gets reading from meter.
`50 OUTPUT ':P1' USING '"Voltage = ",K';A$;`	Outputs reading to printer.
`60 WAIT 10`	Wait 10 seconds.
`70 NEXT F`	
`80 LOCAL ':E1'`	Returns meter to local mode.
`90 END`	

The `OUTPUT` statement (line 50) ends with a semicolon (`;`) to suppress the output of a final EOL sequence. Without the final semicolon, the printer will skip a line after each reading because the voltmeter itself sends **carriage-return/line-feed** after each reading. The `REMOTE` and `LOCAL` statements (lines 10 and 80) are covered in section 5. These statements leave the multimeter addressed to listen. If this causes problems in a program, use `SENDIO` or `SEND` to send the UNL (Unlisten) command.

* The string `F1RAT2` consists of HP 3468A Multimeter command codes (refer to the *HP 3468A Multimeter Operator's Manual*). The function code `F1` specifies **DC Volts**. The range code `RA` specifies **Autorange**. The command code `T2` specifies the **Single Trigger** mode.

Section 5

Other HP-IL Statements and Functions

The HP-75 I/O ROM provides several statements and functions that allow you to automatically assign the loop, select remote or local control of HP-IL devices, check the device ID and accessory ID of HP-IL devices, and conduct serial and parallel polls.These statements and functions are described in this section.

Assigning the Loop

The I/O ROM provides two statements — `ASSIGN LOOP` and `AUTOLOOP ON/OFF` — that enable you to automatically assign device codes to all devices in the loop. You need not assign device codes individually with `ASSIGN IO`. Two functions — `DEVADDR` and `DEVNAME$` — allow you to quickly determine the device address or device code of a specified device. The `ADDRESS` function addresses the loop and returns the number of devices in the loop.

The `ASSIGN LOOP` and `AUTOLOOP ON/OFF` Statements

When you execute the `ASSIGN LOOP` statement, device codes are automatically assigned to all devices in the loop. For each HP-IL device `ASSIGN LOOP` uses the Accessory ID to determine its class, then assigns a two-character device code. Each device code consists of a letter indicating the class of the device followed by a numeral indicating its occurrence within the class. The characters used to indicate the device classes are:

A	Analytical Instrument
B	HP-IB Device
C	Controller
D	Display
E	Electronic Instrument
G	Graphic Device
I	Interface
K	Keyboard Device
M	Mass Storage Device
O	General Device
P	Printer
U	Unknown Class
X	Extended Class

The first display device found would be assigned the device code D1; the third electronic instrument, E3, and so forth. Device codes are assigned in this manner for all classes except "B" (HP-IB Devices). Refer to "Assigning HP-IL Addresses and Device Codes to HP-IB Devices" for information about this class.

The AUTOLOOP statement automatically executes ASSIGN LOOP when the HP-75 is turned on. You may turn this feature on or off by executing AUTOLOOP ON or AUTOLOOP OFF. When AUTOLOOP is in the on state, device codes are assigned to all devices in the loop each time the computer is turned on. The computer "beeps" to indicate that the assignment has been made. AUTOLOOP sends the LPD (Loop Power Down) command when you turn the computer off. AUTOLOOP remains in the on state until you execute AUTOLOOP OFF.

Assigning HP-IL Addresses and Device Codes to HP-IB Devices

When used in "translator" mode, the HP 82169A HP-IL/HP-IB Interface allows you to control HP-IB devices from HP-IL, and vice-versa. (In "mailbox" mode, the interface transfers only **data** between HP-IL and HP-IB.) When the HP 82169A HP-IL/HP-IB Interface is connected in the loop with an HP-75 as the controller, you can assign **HP-IL** addresses for the **HP-IB** devices connected to the interface. The interface must be the last device in the loop, must be in "translator" mode, and must use default addressing (refer to the *HP 82169A HP-IL/HP-IB Interface Owner's Manual*). When the HP-75 assigns addresses to the loop, the interface receives its appropriate address, then reserves all higher numbered HP-IL addresses for the HP-IB devices connected to it. If, for example, the interface is the fifth (and last) device in the loop, it is assigned HP-IL address 5 and reserves HP-IL addresses 6 through 30 for HP-IB devices. You must then set the address switches of each HP-IB device to one of the available addresses.

Once device addresses have been assigned, you can use ASSIGN LOOP or AUTOLOOP to assign device codes. The ASSIGN LOOP statement (or AUTOLOOP) assigns a device code to each HP-IL device in the loop **including** the HP 82169A HP-IL/HP-IB Interface. The interface is assigned a device code of the "I" (Interface) class (for example, I1). Next, ASSIGN LOOP assigns a device code for each of the HP-IL addresses reserved by the interface for HP-IB devices. The first character of each device code is B (indicating an HP-IB Device). The second character of each device code indicates the corresponding address. Addresses 2 through 9 are assigned the device codes B2 through B9. (There can be no device code B1 because the interface itself occupies one address.) Letters are used to represent device addresses above 9. Device addresses 10 through 30 are assigned the device codes BA through BU (address 10 is assigned device code BA, address 11 is assigned BB, and so forth).

Now let's consider a specific configuration. The following devices (in order) are connected in the loop with the HP-75 as the controller: an HP 82161A Digital Cassette Drive, an HP 82162A Thermal Printer, an HP 3468A Multimeter, and the HP 82169A HP-IL/HP-IB Interface. An HP 82905B Printer is connected to the HP-IB side of the interface. The HP-IL devices are assigned addresses 1 through 4. The interface reserves addresses 5 through 30 for HP-IB devices. The ASSIGN LOOP statement assigns the device codes M1, P1, E1, and I1, respectively, for the cassette drive, thermal printer, multimeter, and interface. ASSIGN LOOP assigns the device codes B5 through BU for the reserved addresses (5 through 30). However, the reserved addresses and device codes do not yet correspond to any device. You must set the address switches of the HP 82905B Printer to the address that corresponds with the desired device code. (The owner's manual of each HP-IB device gives the procedure for setting the address switches.) For example, if you set the address to 5, the HP-IB printer will have the device code B5. If you set the address to 10, the device code will be BA. Note that each HP-IB device must have a unique address greater than that of the interface, and that a maximum of 30 devices (HP-IL and HP-IB) may be assigned.

The DEVADDR and DEVNAME$ Functions

The DEVADDR and DEVNAME$ functions operate on the device code or address of a device, allowing you to determine one if you know the other. The DEVADDR function accepts a device code as its argument and returns the address of the specified device. The DEVNAME$ function accepts a device address as its argument and returns the device code as a string. In the following examples assume that the printer has address 5 and the device code P1.

The DEVADDR function can be used in the following BASIC statement:

```
30 A1 = DEVADDR (':P1')
```

DEVADDR will return a value of 5 for A1.

The DEVNAME$ function can be used in the following statement:

```
70 A$ = DEVNAME$ (5)
```

DEVNAME$ will return a value of :P1 for A$.

The ADDRESS Function

The ADDRESS function allows you to quickly determine the number of devices in the loop. The function addresses all devices in the loop and returns a number. ADDRESS causes the controller to assume address 0, then addresses the devices in the loop starting with address 1. Once all addresses have been assigned, the ADDRESS function returns a value equal to the number of devices in the loop (the address of the last device). The ADDRESS function might be used in a BASIC statement as follows:

```
70 X = ADDRESS
```

If there are 15 devices in the loop, the ADDRESS function will address the loop and return the value **15** for X.

Note: If you have already assigned device codes for the devices in the loop, use caution when using the ADDRESS function. ADDRESS will cause no problems as long as you have not added or removed any devices from the loop. However, if you have added or removed devices, the addresses assigned by the ADDRESS function will not agree with the original addresses. This will invalidate the device code assignments.

Remote and Local Control of HP-IL Devices

The HP-75 I/O ROM provides four statements — REMOTE, LOCAL, LOCAL LOCKOUT, and TRIGGER — that allow you to select either remote (through the loop) or local (front panel) control of HP-IL devices.

The `REMOTE` Statement

With the `REMOTE` statement you can set up HP-IL devices for remote control. The general form of this statement is:

simplified syntax

`REMOTE ':`*device code*`'`

You may specify one or more device codes in a `REMOTE` statement, or you may omit the *device code* parameter. If you do not specify a device code, the `REMOTE` statement sends a REN (Remote Enable) message to all devices in the loop.Individual devices will go into the remote state once they are addressed to listen. If device codes are specified, the `REMOTE` statement sends out the UNL and REN messages, then addresses the specified devices to listen. Thus, the devices specified in the *device code* parameter are set up for remote control. Remote mode disables a device's front panel controls except for the power switch and the remote-mode override control (the LOCAL button). In remote mode HP-IL data bytes are interpreted by the device as remote control commands. The following statement sets devices `E1` and `E2` to remote mode:

```
30 REMOTE ':E1,:E2'
```

A device will respond to the REN message only if it has been designed with HP-IL remote control capability. Once a device has been set up for remote control, the functions that can be controlled remotely by the HP-IL controller depend on the design of the device. For example, the HP 3468A Multimeter allows you to control its range settings remotely.

> **Note:** The `REMOTE` statement (also the `LOCAL` and `TRIGGER` statements) leave HP-IL devices addressed to listen. You may remove listen-addressed status by sending the UNL (Unlisten) command with `SENDIO` or `SEND`.

The `LOCAL` Statement

With the `LOCAL` statement you can return HP-IL devices from the remote state to local control.The general form of this statement is:

simplified syntax

`LOCAL ':`*device code*`'`

The *device code* parameter is optional, and one or more device codes may be specified. If device codes are specified, the `LOCAL` statement sends out the UNL message, addresses the specified devices to listen, then sends the GTL (Go To Local) message. The GTL message returns the devices to local control, but leaves them remote enabled and addressed to listen. The devices will return to remote mode when next addressed to listen. The following statement returns `E1` and `E2` to local control, but leaves them remote enabled:

```
50 LOCAL ':E1,:E2'
```

If the `LOCAL` statement is used without parameters, the NRE (Not Remote Enable) message is sent. This removes remote enabled status from all devices in the loop. The following statement returns all devices to local control and removes remote enabled status:

```
50 LOCAL
```

The LOCAL LOCKOUT Statement

The LOCAL LOCKOUT statement enables you to lock out the front panel remote-mode override control (the LOCAL button) on a device that is in the remote state.This prevents an operator from returning to local control at a critical time during remote operation. The statement has no parameters:

```
LOCAL LOCKOUT
```

The LOCAL LOCKOUT statement sends the LLO (Local Lockout) message. To establish local lockout for devices E1 and E2 you could use the following sequence of instructions:

```
10 REMOTE ':E1,:E2'
20 LOCAL LOCKOUT
```

Only those devices that have been designed with local lockout capability will respond to the LLO message. You can return a device from the local lockout state to local control with the LOCAL statement.

The TRIGGER Statement

You can use the TRIGGER statement to initiate operation of devices that are designed to respond to the GET (Group Execute Trigger) message. The general form of this statement is:

simplified syntax

TRIGGER ':*device code*'

You may specify one or more device codes in the *device code* parameter, or you may leave it blank. If you do not specify a device code, the GET message is sent. All devices that have already been addressed to listen will receive the GET message. If device codes are specified, the TRIGGER statement sends the UNL message, addresses the specified devices to listen, then sends the GET message. The following statement causes devices E1, E2, and E3 to initiate operation:

```
80 TRIGGER ':E1,:E2,:E3'
```

The response of an HP-IL device to the GET message depends on the design of the device. The TRIGGER statement simply initiates the operation of several devices at (approximately) the same time. For example, several temperature measuring instruments could be periodically triggered with this statement.

The possible remote control applications using the REMOTE, LOCAL, LOCAL LOCKOUT, and TRIGGER statements are obviously numerous. However, since the response of an individual device to these statements depends on the design of the device, specific applications are beyond the scope of this manual. The remote control characteristics of individual HP-IL devices are covered in the owner's manuals for those devices. For general information about remote and local control, refer to *THE HP-IL SYSTEM: An Introductory Guide to the Hewlett-Packard Interface Loop*, by Gerry Kane, Steve Harper, and David Ushijima, published by OSBORNE/McGraw-Hill, Berkeley, California, 1982.

Checking the Device ID or Accessory ID of HP-IL Devices

The HP-75 I/O ROM provides two functions — `DEVID$` and `DEVAID$` — that enable you to check the device ID or accessory ID of HP-IL devices. Only one device at a time may be specified in either function.

Device ID

The `DEVID$` function allows you to check the device ID of an HP-IL device. The general form of this function is:

`DEVID$ (':`*device code*`')`

`DEVID$` addresses the specified device as the talker and sends the SDI (Send Device ID) message. The device sends its device identification, and `DEVID$` returns this identification as a string. The device identification that a device sends is usually an ASCII string consisting of a two-letter manufacturer's code, a five-character model number, model revision, and any additional information included by the manufacturer of the device. In the following example `DEVID$` is used to determine the device identification of an HP 3468A Multimeter that has been assigned the device code `E1`.

```
40 A$ = DEVID$ (':E1')
```

The `DEVID$` function returns the device identification `HP3468A` as the value of A$.

Accessory ID

The `DEVAID$` function allows you to check the accessory ID of an HP-IL device. The general form of this function is:

`DEVAID$ (':`*device code*`')`

`DEVAID$` addresses the specified device as the talker and sends the SAI (Send Accessory ID) message. The talker sends its accessory identification and `DEVAID$` returns this identification as a string. The accessory identification is usually a single byte in which the most-significant four bits designate the device class (for example, printer, mass-storage device, etc.) and the least-significant four bits indicate a specific device. Since `DEVAID$` returns a character string, this eight-bit byte is represented as an ASCII character. In the following example `DEVAID$` is used to determine the acessory identification of an HP 82161A Digital Cassette Drive that has been assigned the device code `M1`.

```
70 B$ = DEVAID$ (':M1')
```

The `DEVAID$` function returns the ASCII character `θ` as the value of B$.

> **Note:** Certain characters (for example, the Greek letters) may not be printable with your printer. Thus, the `DEVID$`, `DEVAID$` and `SPOLL$` functions may return strings that contain characters that do not appear in a printout. However, all characters will appear on the display.

Polling HP-IL Devices

The HP-75 I/O ROM provides three functions that enable you to conduct polls of HP-IL devices. The SPOLL and SPOLL$ functions are used in serial polls. The PPOLL function is used to conduct parallel polls.

Serial Polling

The SPOLL and SPOLL$ functions both conduct a serial poll of a specified device. These functions differ in the way they represent the results of the poll.

The general form of the SPOLL function is:

SPOLL (' :*device code*')

The SPOLL function sends the SST (Send Status) message to the specified device. The device responds by sending back one or more status bytes. The value returned by the SPOLL function is the first status byte, represented as a number. In the following example SPOLL is used to conduct a serial poll of an HP 82162A Thermal Printer that has been assigned the device code P1:

```
140 X = SPOLL (':P1')
```

If the printer sends the status bytes “00100000” and “01100000”, SPOLL returns 32 (the decimal value of the first byte) as the value of X.

The SPOLL$ function conducts a serial poll of a specified device, like SPOLL, but returns the result as a character string. The general form of this function is:

SPOLL$ (' :*device code*')

The SPOLL$ function sends the SST message to the specified device. The device responds by sending back one or more status bytes. The value returned by the SPOLL$ function is a string of ASCII characters representing the status bytes. Suppose that SPOLL$, rather than SPOLL, is used to conduct the serial poll of the previous example:

```
190 D$ = SPOLL$ (':P1')
```

The SPOLL$ function converts the status bytes “00100000” and “01100000” to the ASCII characters with the equivalent decimal codes (32 and 96). The string returned for D$ is “ `”. Note that the first character in the string is CHR$(32), a blank space.

Parallel Polling

The `PPOLL` function conducts a parallel poll of those devices in the loop that have been configured for parallel polling. The `PPOLL` function sends the IDY (Identify) message. All devices that are to be polled must be capable of responding to this message. Each device in the poll sets one bit of the parallel poll response byte according to its configuration. The `PPOLL` function has no parameters, and returns a number representing the response byte.

Each device to be polled must be configured for parallel polling before you execute the `PPOLL` function. Each device is configured by sending the appropriate PPEn (Parallel Poll Enable) message to the device with the `SENDIO` statement. The PPEn message configures a device to set the one of the eight **data bits** (D0 through D7) of the parallel poll response byte, and also specifies whether the device is to set the bit if service is requested or if service is not requested.

> **Note:** Normally, each device will specify its own exclusive bit in the response byte, allowing you to poll up to eight devices at once. It is possible to assign more than one device to each bit of a parallel poll response byte. If you do, you can poll more than eight devices. However, if two or more devices share a bit that has been set, you will not be able to tell which device set it.

The **PPEn** message enables a device to respond to an IDY message, and defines the response according to the value of **n**, an integer from 0 to 15. The following table lists the configurations set by PPEn messages from PPE0 to PPE15. Note that PPE0 through PPE7 specify that the configured device is to set the designated bit of the response byte (D0 through D7) if service **is not** requested. The messages PPE8 through PPE15 specify that the device is to set the designated bit if service **is** requested.

> **Note:** In a parallel poll response, a device will set its assigned bit to a "1" if the condition specified in the table exists. Otherwise the bit will be left unchanged. Also, **control bit** C0 will be set if service is requested by any device in the poll.

Parallel Poll Response to an IDY Message

Enable message:	Designates bit...	Device sets that bit if...
PPE0	D0	service is not requested.
PPE1	D1	
PPE2	D2	
PPE3	D3	
PPE4	D4	
PPE5	D5	
PPE6	D6	
PPE7	D7	
PPE8	D0	service is requested.
PPE9	D1	
PPE10	D2	
PPE11	D3	
PPE12	D4	
PPE13	D5	
PPE14	D6	
PPE15	D7	

An example will show how to configure the loop. Suppose that there are two devices in the loop, a printer at address 1, and a digital cassette drive at address 2. You should start by setting the loop to an initial condition by executing the following `SENDIO` statement:

```
SENDIO '','UNL,PPU',''
```

The UNL (Unlisten) command prevents unwanted devices from responding to the subsequent commands. The PPU (Parallel Poll Unconfigure) command resets any existing parallel-polling configuration. Remember that `SENDIO` automatically sends an RFC (Ready For Command) message after each command. You may now start configuring the devices, one at a time, for the parallel poll. The following statement will configure the first device (the printer):

```
SENDIO '','LAD1,PPE13,UNL',''
```

The `LAD1` command addresses device 1 to listen. `PPE13` specifies that the addressed device should use bit D5 of the parallel poll response byte, and should set that bit to a "1" if service is requested. The UNL command unlistens the printer so that it will ignore further commands.

You may now configure another device. The following statement configures device 2 (the cassette drive) to set bit D7 of the response byte to a "1" if service is **not** requested:

```
SENDIO '','LAD2,PPE7,UNL',''
```

Once you have configured the desired devices for parallel polling, you may execute the `PPOLL` function as often as you want. The IDY message will be sent out each time you execute `PPOLL`, and each device will assert one bit of the response byte according to the configuration. The `PPOLL` function will return a number representing the response byte. You could poll devices 1 and 2 (configured above) by executing the following statement:

```
40 X = PPOLL
```

Device 1 will set bit D5 of the response byte if it needs service, and device 2 will set bit D7 if it does not need service (according to the above configuration). The value of X will be a number that represents the response byte. If the response byte is "10100000", `PPOLL` will return the value 160.

For further information on parallel polling, refer to *THE HP-IL SYSTEM: An Introductory Guide to the Hewlett-Packard Interface Loop*, by Kane, Harper, and Ushijima.

Appendix A

Owner's Information

CAUTIONS

Do not place fingers, tools, or other objects into the plug-in ports. Damage to plug-in module contacts and the computer's internal circuitry may result.

Turn off the computer (press [SHIFT] [ATTN]) before installing or removing a plug-in module.

If a module jams when inserted into a port, it may be upside down. Attempting to force it further may result in damage to the computer or the module.

Handle the plug-in modules very carefully while they are out of the computer. Do not insert any objects in the module connector socket. Always keep a blank module in the computer's port when a module is not installed. Failure to observe these precautions may result in damage to the module or the computer.

Limited One-Year Warranty

What We Will Do

The HP-75 I/O ROM is warranted by Hewlett-Packard against defects in materials and workmanship affecting electronic and mechanical performance, but not software content, for one year from the date of original purchase. If you sell your unit or give it as a gift, the warranty is transferred to the new owner and remains in effect for the original one-year period. During the warranty period, we will repair or, at our option, replace at no charge a product that proves to be defective, provided you return the product, shipping prepaid, to a Hewlett-Packard service center.

What Is Not Covered

This warranty does not apply if the product has been damaged by accident or misuse or as the result of service or modification by other than an authorized Hewlett-Packard service center.

No other express warranty is given. The repair or replacement of a product is your exclusive remedy. **ANY OTHER IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS IS LIMITED TO THE ONE-YEAR DURATION OF THIS WRITTEN WARRANTY.** Some states, provinces, or countries do not allow limitations on how long an implied warranty lasts, so the above limitation may not apply to you. **IN NO EVENT SHALL HEWLETT-PACKARD COMPANY BE LIABLE FOR CONSEQUENTIAL DAMAGES.** Some states, provinces, or countries do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you.

This warranty gives you specific legal rights, and you may also have other rights which vary from state to state, province to province, or country to country.

Warranty for Consumer Transactions in the United Kingdom

This warranty shall not apply to consumer transactions and shall not affect the statutory rights of a consumer. In relation to such transactions, the rights and obligations of Seller and Buyer shall be determined by statute.

Obligation to Make Changes

Products are sold on the basis of specifications applicable at the time of manufacture. Hewlett-Packard shall have no obligation to modify or update products once sold.

Warranty Information

If you have any questions concerning this warranty, please contact an authorized Hewlett-Packard dealer or a Hewlett-Packard sales and service office. Should you be unable to contact them, please contact:

- In the United States:

Hewlett-Packard
Personal Computer Group
Customer Support
11000 Wolfe Road
Cupertino, CA 95014

Toll-Free Number: (800) FOR-HPPC (800 367-4772)

- In Europe:

Hewlett-Packard S.A.
150, route du Nant-d'Avril
P.O. Box CH-1217 Meyrin 2
Geneva
Switzerland
Telephone: (022) 83 81 11

Note: Do *not* send units to this address for repair.

- In other countries:

Hewlett-Packard Intercontinental
3495 Deer Creek Rd.
Palo Alto, California 94304
U.S.A.
Telephone: (415) 857-1501

Note: Do not send units to this address for repair.

Service

Hewlett-Packard maintains service centers in most major countries throughout the world. You may have your unit repaired at a Hewlett-Packard service center any time it needs service, whether the unit is under warranty or not. There is a charge for repairs after the one-year warranty period.

Hewlett-Packard products are normally repaired and reshipped within five (5) working days of receipt at any service center. This is an average time and could vary depending upon the time of year and the work load at the service center. The total time you are without your unit will depend largely on the shipping time.

Obtaining Repair Service in the United States

The Hewlett-Packard United States Service Center for battery-powered computational products is located in Corvallis, Oregon:

Hewlett-Packard Company
Service Department

P.O. Box 999
Corvallis, Oregon 97339, U.S.A.

or

1030 N.E. Circle Blvd.
Corvallis, Oregon 97330, U.S.A.

Telephone: (503) 757-2000

Obtaining Repair Service in Europe

Service centers are maintained at the following locations. For countries not listed, contact the dealer where you purchased your unit.

AUSTRIA
HEWLETT-PACKARD Ges.m.b.H.
Kleinrechner-Service
Wagramerstrasse-Lieblgasse 1
A-1220 Wien (Vienna)
Telephone: (0222) 23 65 11

BELGIUM
HEWLETT-PACKARD BELGIUM SA/NV
Woluwedal 100
B-1200 Brussels
Telephone: (02) 762 32 00

DENMARK
HEWLETT-PACKARD A/S
Datavej 52
DK-3460 Birkerod (Copenhagen)
Telephone: (02) 81 66 40

EASTERN EUROPE
Refer to the address listed under Austria.

FINLAND
HEWLETT-PACKARD OY
Revontulentie 7
SF-02100 Espoo 10 (Helsinki)
Telephone: (90) 455 02 11

FRANCE
HEWLETT-PACKARD FRANCE
Division Informatique Personnelle
S.A.V. Calculateurs de Poche
F-91947 Les Ulis Cedex
Telephone: (6) 907 78 25

GERMANY
HEWLETT-PACKARD GmbH
Kleinrechner-Service
Vertriebszentrale
Berner Strasse 117
Postfach 560 140
D-6000 Frankfurt 56
Telephone: (611) 50041

ITALY
HEWLETT-PACKARD ITALIANA S.P.A.
Casella postale 3645 (Milano)
Via G. Di Vittorio, 9
I-20063 Cernusco Sul Naviglio (Milan)
Telephone: (2) 90 36 91

NETHERLANDS
HEWLETT-PACKARD NEDERLAND B.V.
Van Heuven Goedhartlaan 121
NL-1181 KK Amstelveen (Amsterdam)
P.O. Box 667
Telephone: (020) 472021

NORWAY
HEWLETT-PACKARD NORGE A/S
P.O. Box 34
Oesterndalen 18
N-1345 Oesteraas (Oslo)
Telephone: (2) 17 11 80

SPAIN
HEWLETT-PACKARD ESPANOLA S.A.
Calle Jerez 3
E-Madrid 16
Telephone: (1) 458 2600

SWEDEN
HEWLETT-PACKARD SVERIGE AB
Skalholtsgatan 9, Kista
Box 19
S-163 93 Spanga (Stockholm)
Telephone: (08) 750 2000

SWITZERLAND
HEWLETT-PACKARD (SCHWEIZ) AG
Kleinrechner-Service
Allmend 2
CH-8967 Widen
Telephone: (057) 31 21 11

UNITED KINGDOM
HEWLETT-PACKARD Ltd
King Street Lane
GB-Winnersh, Wokingham
Berkshire RG11 5AR
Telephone: (0734) 784 774

International Service Information

Not all Hewlett-Packard service centers offer service for all models of HP products. However, if you bought your product from an authorized Hewlett-Packard dealer, you can be sure that service is available in the country where you bought it.

If you happen to be outside of the country where you bought your unit, you can contact the local Hewlett-Packard service center to see if service is available for it. If service is unavailable, please ship the unit to the address listed above under Obtaining Repair Service in the United States. A list of service centers for other countries can be obtained by writing to that address.

All shipping, reimportation arrangements, and customs costs are your responsibility.

Service Repair Charge

There is a standard repair charge for out-of-warranty repairs. The repair charges include all labor and materials. In the United States, the full charge is subject to the customer's local sales tax. In European countries, the full charge is subject to Value Added Tax (VAT) and similar taxes wherever applicable. All such taxes will appear as separate items on invoiced amounts.

Computer products damaged by accident or misuse are not covered by the fixed repair charges. In these situations, repair charges will be individually determined based on time and materials.

Service Warranty

Any out-of-warranty repairs are warranted against defects in materials and workmanship for a period of 90 days from date of service.

Shipping Instructions

Should your unit require service, return it with the following items:

- A completed Service Card, including a description of the problem.
- A sales receipt or other proof of purchase date if the one-year warranty has not expired.

The product, the Service Card, a brief description of the problem, and (if required) the proof of purchase date should be packaged in adequate protective packaging to prevent in-transit damage. Such damage is not covered by the one-year limited warranty; Hewlett-Packard suggests that you insure the shipment to the service center. The packaged unit should be shipped to the nearest Hewlett-Packard designated collection point or service center. Contact your dealer for assistance. (If you are not in the country where you originally purchased the unit, refer to "International Service Information" above.)

Whether the unit is under warranty or not, it is your responsibility to pay shipping charges for delivery to the Hewlett-Packard service center.

After warranty repairs are completed, the service center returns the unit with postage prepaid. On out-of-warranty repairs in the United States and some other countries, the unit is returned C.O.D. (covering shipping costs and the service charge).

Further Information

Service contracts are not available. Circuitry and designs are proprietary to Hewlett-Packard, and service manuals are not available to customers. Should other problems or questions arise regarding repairs, please call your nearest Hewlett-Packard service center.

When You Need Help

Hewlett-Packard is committed to providing after-sale support to its customers. To this end, our customer support department has established phone numbers that you can call if you have questions about this product.

Product Information. For information about Hewlett-Packard dealers, products, and prices, call the toll-free number below:

(800) FOR-HPPC
(800 367-4772)

Technical Assistance. For technical assistance with your product, call the number below:

(408) 725-2600

For either product information or technical assistance, you can also write to:

Hewlett-Packard
Personal Computer Group
Customer Support
11000 Wolfe Road
Cupertino, CA 95014

Appendix B

Syntax Reference Guide

This appendix provides syntax definitions for the statements and functions described in sections 1 through 5 of this manual. The syntax representations in this appendix follow the format described in section 1 (refer to the subheading "Syntax Guidelines").

ADDRESS

Syntax

```
ADDRESS
```

Sample Statement

```
70 A1 = ADDRESS
```

Actions Taken

Addresses all devices in the loop, starting with 1, and returns a value equal to the number of devices (the address of the last device).

Related Statements

```
ASSIGN LOOP
AUTOLOOP ON/OFF
```

ASSIGN LOOP

Syntax

```
ASSIGN LOOP
```

Actions Taken

Causes two-character device codes to be assigned to each device in the loop. The first character (a letter) indicates the class of the device. The second character (a numeral) indicates the occurrence of the device. The following letters are used to indicate device class:

A	Analytical Instrument
B	HP-IB Device
C	Controller
D	Display
E	Electronic Instrument
G	Graphic Device
I	Interface
K	Keyboard Device
M	Mass Storage Device
O	General Device
P	Printer
U	Unknown Class
X	Extended Class

Note: Class "B" (HP-IB Devices) is treated differently. Refer to "Assigning HP-IL Addresses and Device Codes to HP-IB Devices" in section 5.

Related Statements

```
ADDRESS
AUTOLOOP ON/OFF
```

AUTOLOOP ON/OFF

Syntax

```
         ON
AUTOLOOP
         OFF
```

Actions Taken

Device codes are assigned to all devices in the loop each time the computer is turned on if AUTOLOOP is in the on state. A "beep" indicates that the assignment has been made. Device codes are assigned following the same rules used by ASSIGN LOOP. Also, AUTOLOOP sends the LPD (Loop Power Down) message when the computer is turned off. AUTOLOOP remains in the on state until an AUTOLOOP OFF command is executed.

Related Statements

ADDRESS
ASSIGN LOOP

DEVADDR

Syntax

```
DEVADDR (':device code')
```

Sample Statements

```
30 B1 = DEVADDR (':D1')
70 X = DEVADDR (A$)
```

Parameters

device code — a valid HP-IL device code. You may substitute the name of a string variable that contains the desired device code.

Actions Taken

Returns the HP-IL address of the specified device.

Related Statements

```
DEVNAME$
```

DEVAID$

Syntax

```
DEVAID$ (':device code')
```

Sample Statement

```
40 B$ = DEVAID$ (':D1')
```

Parameters

device code — a valid HP-IL device code. You may substitute the name of a string variable that contains the desired device code.

Actions Taken

Addresses the specified device as the talker and sends the SAI (Send Accessory ID) message. The talker sends its accessory identification, and DEVAID$ returns this identification as a string. The accessory identification is usually a single byte, and is represented as an ASCII character.

Related Statements

DEVID$

DEVID$

Syntax

```
DEVID$ (':device code')
```

Sample Statement

```
40 A$ = DEVID$ (':P3')
```

Parameters

device code — a valid HP-IL device code. You may substitute the name of a string variable that contains the desired device code.

Actions Taken

Addresses the specified device as the talker and sends the SDI (Send Device ID) message. The device sends its device identification, and DEVID$ returns this identification as an ASCII character string (including any **carriage-return/line-feed** characters sent by the device).

Related Statements

DEVAID$

DEVNAME$

Syntax

```
DEVNAME$ (device address)
```

Sample Statements

```
60 A$ = DEVNAME$ (15)
90 C$ = DEVNAME$ (A1)
```

Parameters

device address — a valid HP-IL device address (0 through 30).

Actions Taken

Returns the device code of the specified device.

Related Statements

```
DEVADDR
```

ENTER

Syntax

ENTER	[' :*device code* ' / *device address*]	[USING	' *image list* ' / *line number*]	[; [*variable*][, *variable*]...]

Sample Statements

```
70 ENTER ':TP' USING A$;X,Y,Z
90 ENTER C$;N(I),Z$
120 ENTER ':D1' USING 30;A$
150 ENTER USING 30;A$
170 ENTER ;B$
```

Parameters

device code — a valid HP-IL device code. You may substitute the name of a string variable that contains the desired device code.

device address — a valid HP-IL device address (0 through 30).

image list — a string expression that contains a valid set of image specifiers. The expression can be either a list of image specifiers enclosed in quotation marks or the name of a string variable that contains a list of image specifiers.

line number — the line number of an IMAGE statement that contains a valid set of image specifiers.

variable (numeric or string) — the name of a variable intended as a destination of the ENTER operation.

Actions Taken

Inputs bytes from the specified device; uses those bytes to build a number or string; places the result into a BASIC variable.

When USING is not specified, free-field format is used. A free-field entry into a string places incoming bytes into the variable until the current EOL (end-of-line) sequence or an End Byte message is received, or the string is full. Terminating sequences are not placed into the destination string. A free-field entry into a numeric variable ignores leading blanks and non-numeric characters. Entry into a numeric variable is terminated by the first trailing blank or non-numeric character.

When USING is specified, input operations are formatted according to the image specifiers used. Image specifiers may be enclosed in quotation marks and placed in the ENTER statement, contained in a string variable named in the ENTER statement, or placed in an IMAGE statement referenced by the ENTER statement. For detailed information on image specifiers, refer to "Formatted ENTER" in section 3.

ENTER requires either the current EOL sequence or an End Byte message to terminate the statement after the variable list has been satisfied. If no EOL sequence or End Byte message is detected, an error will be issued. This requirement can be removed by using #! as the first image specifier. For more detailed information on statement terminators, refer to “Formatted ENTER”.

Related Statements

IMAGE

ENTIO$

Syntax

```
ENTIO$ ('[:device code[,:device code]...]','[command[,command]...]')
```

Sample Statements

```
30 A$ = ENTIO$ ('','TAD1,SDA')
170 X$ = ENTIO$ (':D1','TAD#,SDA')
230 B$ = ENTIO$ (':D3','')
```

Parameters

device code — a valid HP-IL device code. You may substitute the name of a string variable that contains the desired device code.

command — a valid HP-IL command mnemonic (refer to appendix C). You may substitute the name of a string variable that contains the list of commands.

Actions Taken

ENTIO$ is a function that returns a character string value. ENTIO$ is usually used to address an HP-IL device as a talker, then return the data received from that device as the value of the function. Only one device may be addressed as a talker, but one or more listeners may be addressed.

ENTIO$ processes parameters from left to right. If a *device code* parameter has been specified, ENTIO$ determines the corresponding device address in the loop. If TAD# is specified in the *command* field, only one device code may be specified. If the *device code* field is the null string, no action is taken in this step.

Next, the list of HP-IL commands in the *command* field is processed. A TAD# or LAD# command causes the device specified in the *device code* field to be addressed as a talker or listener, respectively. If no device code is specified, TAD# and LAD# are not valid in the command list. The TADn and LADn commands contain HP-IL device addresses. A TADn or LADn in the command list causes the device with address n to be addressed as a talker or listener. ENTIO$ returns the null string unless the last command in the *command* field is SDA, SST, SDI, SAI, AADn, or ID:00. The data sent by the active talker in response to the ready group command is returned as the result of the ENTIO$ function. If the *command* field is the null string, ENTIO$ automatically generates TAD#,SDA.

Either the *device code* field or the *command* field can be the null string, but not both.

Related Statements

SENDIO

IMAGE

Syntax

IMAGE *specifier* [, *specifier*]...

Sample Statements

```
10 IMAGE 'Total =',4D.DD
100 IMAGE #,K,2X,K
```

Parameters

specifier — a valid OUTPUT or ENTER image specifier. These specifiers are listed below. Refer to section 3, "Formatted I/O Operations", for detailed descriptions.

Summary of OUTPUT Image Specifiers

Image	Meaning
a,A	Output one string character
c,C	Output a comma separator in a number
d,D	Output one digit character; blank for leading zero
e,E	Output exponent information; five characters
k,K	Output a variable in free-field format
m,M	Output number's sign if negative, blank if positive
p,P	Output a period separator in a number
r,R	Output a European radix point (comma)
s,S	Output number's sign, plus or minus
x,X	Output one blank
z,Z	Output one digit character, including leading zeros
' '," "	Output a literal (enclosed in quotation marks)
*	Output one digit character; asterisk for leading zero
.	Output an American radix point (decimal point)
/	Output the current EOL sequence

Summary of ENTER Image Specifiers

Image	Meaning
a,A	Demand one string character
c,C	Demand one character for a numeric field; allows commas to be skipped over
d,D	Demand one character for a numeric field
e,E	Demand five characters for a numeric field
k,K	Enter a variable in free-field format
m,M	Demand one character for a numeric field
p,P	Demand one digit and ignore all periods
r,R	Demand one digit and treat comma as radix symbol
s,S	Demand one character for a numeric field
x,X	Skip one character
z,Z	Demand one character for a numeric field
*	Demand one character for a numeric field
.	Demand one character for a numeric field
/	Demand the current EOL sequence
#	Eliminate the current EOL sequence as a terminator
!	Eliminate the End Byte message as a terminator
%	Establish the ETO (End Of Transmission — OK) message as an alternative terminator

Related Statements

ENTER...USING
OUTPUT...USING

IOSIZE

Syntax

IOSIZE *buffer size*

Sample Statement

```
IOSIZE 500
```

Parameters

buffer size — an integer representing the desired buffer size (range: 0 to 24,575 bytes). A zero or negative value specifies the default value of 256 bytes.

Actions Taken

Sets the size of the ENTER buffer to the specified value. Controls the maximum number of bytes to be read by a statement or function that causes input of data (ENTER, ENTIO$, ADDRESS, etc.) If *buffer size* is exceeded, a record overflow error will result. A SZ= command in an ENTIO$ command list will override the value of IOSIZE for that ENTIO$ statement only.

Related Statements

```
ENTER
ENTIO$
```

LOCAL

Syntax

```
LOCAL [' :device code[, :device code]... ']
```

Sample Statements

```
100 LOCAL
220 LOCAL ':D1'
330 LOCAL ':B1,:B2,:B3'
```

Parameters

device code — a valid HP-IL device code. You may substitute the name of a string variable that contains the desired device code(s).

Actions Taken

LOCAL addresses the specified device(s) to listen and sends the GTL (Go To Local) message. The specified devices are returned to local mode, but remain remote enabled. LOCAL leaves devices addressed to listen.

If no device code is specified, LOCAL sends the NRE (Not Remote Enable) message. This returns devices to local control and removes remote enabled status.

Related Statements

```
LOCAL LOCKOUT
REMOTE
TRIGGER
```

LOCAL LOCKOUT

Syntax

```
LOCAL LOCKOUT
```

Sample Statements

```
50 LOCAL LOCKOUT
LOCAL LOCKOUT
```

Action Taken

Sends LLO (Local Lockout) command. Locks out LOCAL button on front panel of devices in remote mode. Devices can be returned to local control only by a GTL or NRE message (refer to the `LOCAL` command).

Related Statements

```
LOCAL
REMOTE
TRIGGER
```

OUTPUT

Syntax

```
OUTPUT [ ':device code[,:device code]...' ] [USING 'image list' ]
       [ device address                   ] [      line number   ]

       [;expression[,expression][;expression]...]
```

Sample Statements

```
40 OUTPUT ; A$
70 OUTPUT ':TV' USING A$ ; X,Y,Z
90 OUTPUT C$ ; N(I);Z$
120 OUTPUT ':D1' USING 30 ; A$
```

Parameters

device code — a valid HP-IL device code. You may substitute the name of a string variable that contains the desired device code(s).

device address — a valid HP-IL device address (0 through 30). Only one device address may be specified. Use device codes if more than one device is to be specified.

image list — a string expression that contains a valid set of image specifiers. The expression can be either a list of image specifiers enclosed in quotation marks or the name of a string variable that contains a list of image specifiers.

line number — the line number of an `IMAGE` statement that contains a valid set of image specifiers.

expression (string or numeric) — any string expression or numeric expression intended to be output. Expressions may be constants or variables and may be separated by commas or semicolons.

Actions Taken

Outputs bytes to the specified device(s); bytes may be string or numeric.

When `USING` is not specified, and output items are separated by semicolons, compact format is used. A compact output of a string expression causes it to be sent with no leading or trailing blanks. A compact output of a numeric quantity causes it to be sent with one trailing blank and one leading sign character (blank if positive, minus sign if negative).

When `USING` is specified, output operations are formatted according to the image specifiers used. Image specifiers may be enclosed in quotes and placed in the `OUTPUT` statement, contained in a string variable named in the `OUTPUT` statement, or placed in an `IMAGE` statement referenced by the `OUTPUT` statement. For detailed information on image specifiers, refer to "Formatted `OUTPUT`" in section 3.

OUTPUT sends the current EOL (end-of-line) sequence after the last item in the OUTPUT list. This sequence can be changed with the ENDLINE statement, and defaults to **carriage-return/line-feed**. The EOL sequence can be suppressed by using ; after the last variable. For more detailed information on statement terminators, refer to "Formatted OUTPUT".

Related Statements

IMAGE

PPOLL

Syntax

```
PPOLL
```

Sample Statements

```
310 X=PPOLL
620 P9=PPOLL
```

Actions Taken

PPOLL is a function that returns the results of a Parallel Poll operation. Sends an IDY (Identify) message. Devices capable of responding each assert one bit of the parallel poll response byte.

Related Statements

```
SPOLL
SPOLL$
```

REMOTE

Syntax

`REMOTE` [' `:`*device code*[`,` `:`*device code*]... ']

Sample Statements

```
50 REMOTE ':D1'
130 REMOTE S1$
190 REMOTE
```

Parameters

device code — a valid HP-IL device code. You may substitute the name of a string variable that contains the desired device code(s).

Actions Taken

If no device code is given, `REMOTE` sends the REN (Remote Enable) message. Devices do not go into remote mode until they are addressed to listen.

If device codes are specified, `REMOTE` sends the UNL (Unlisten) and REN messages, then addresses the specified devices to listen. Devices are left addressed to listen.

Related Statements

```
LOCAL
LOCAL LOCKOUT
TRIGGER
```

SEND

Syntax

```
SEND [[CMD  byte number [, byte number]...
            byte string                    ]

      [DATA byte number [, byte number]...  [EOL]]
            byte string

      [END  byte number [, byte number]...  [EOL]]
            byte string

      [IDY byte number [, byte number]...] [RDY byte number [, byte number]...]

      [DDL byte number [, byte number]...] [DDT byte number [, byte number]...]

      [SAD byte number [, byte number]...] [LISTEN byte number [, byte number]...]

      [TALK byte number] [GTL] [RMO] [NRE] [LLO] [CIF] [LPD] [MLA]

      [MTA] [SDC] [UNL] [UNT]]...
```

Note: The above bracketed items may be included in any order. They may be repeated as many times as desired, with one exception: `EOL` may be included only once in a `DATA` or `END` field.

Sample Statements

```
100 SEND CMD 'U?%' DATA 'Hello'
200 SEND CMD A$ SAG 14,18 DATA X$
300 SEND MTA UNL LISTEN 6,14 DATA 'ABC'
```

Parameters

byte number — a number that specifies the actual message to be sent. Byte numbers for the `CMD`, `DATA`, `END`, `IDY`, and `RDY` message indicators represent bits D0 through D7 of the message, and have the range 0 through 255 (modulo 256). Byte numbers for the `DDL`, `DDT`, `SAD`, `LISTEN`, and `TALK` message indicators have the range 0 through 31 (modulo 32).

byte string — a string of ASCII characters that specify a series of messages. Each character represents a message having the *byte number* equivalent to its ASCII character code.

Actions Taken

CMD	Sends list of commands specified by *byte number*. Each *byte number* specifies bits D0 through D7 of the command message. A *byte string* may be substituted for a list of byte numbers. Each character in the string specifies the command with the *byte number* equivalent to its ASCII character code.
DATA	Sends list of Data Byte messages with bits D0 through D7 specified by *byte number.* A *byte string* may be substituted for a list of byte numbers. Each character specifies the bit pattern with the *byte number* equivalent to its ASCII decimal code. ASCII character strings may be sent exactly as specified in quotes. Inclusion of EOL causes the current EOL sequence to be sent.
END	Sends End Byte message, but otherwise same as DATA.
IDY	Sends identify message having bits set according to byte number.
RDY	Sends ready message having bits set according to byte number.
DDL	Sends Device-Dependent Listener message having number 0 through 31 indicated by byte number (modulo 32).
DDT	Sends Device-Dependent Talker message having number 0 through 31 indicated by byte number (modulo 32).
SAD	Sends Secondary Address message having address 0 through 31 indicated by byte number (modulo 32). Associates this secondary address with the primary address of the preceding command message, indicating an extended address.
LISTEN	Sends LADn (Listen Address) message to device n, the address specified by a byte number in the range 0 through 31 (modulo 32). Makes device n a listener, except that 31 clears all devices from listener status.
TALK	Sends TADn (Talk Address) message to device n, the address specified by a byte number in the range 0 through 31 (modulo 32). Makes device n a talker, except that 31 clears all devices from listener status.
GTL	Sends GTL (Go To Local) message.
RMO	Sends REN (Remote Enable) message.
NRE	Sends NRE (Not Remote Enable) message.
LLO	Sends LLO (Local Lockout) message.
CIF	Sends IFC (Interface Clear) message.
LPD	Sends LPD (Loop Power Down) message.
MLA	Sends no message.
MTA	Sends UNT (Untalk) message.
SDC	Sends SDC (Selected Device Clear) message.
UNL	Sends UNL (Unlisten) message.
UNT	Sends UNT (Untalk) message.

SEND?

Syntax

```
SEND?
```

Sample Statements

```
30 C1 = SEND?
80 B$ = A$[SEND?]
```

Actions Taken

Returns an integer value representing the position in the string of the character that was unsuccessfully sourced in the last SENDIO statement. Returns a value of 0 if the SENDIO *data list* was null, or if the last SENDIO statement was successfully completed.

Related Statements

SENDIO

SENDIO

Syntax

SENDIO '[:*device code*[, :*device code*]...]', '[*command*[, *command*]...]', '[*data*]'

Sample Statements

```
30 SENDIO ':D1,:D2','LAD#,LAD5','DATA'
50 SENDIO '','LAD1,LAD2','HI'
90 SENDIO '','','BYE'
```

Parameters

device code — a valid HP-IL device code. You may substitute the name of a string variable that contains the desired device code(s).

command — a valid HP-IL command mnemonic (refer to appendix C). You may substitute the name of a string variable that contains the list of commands.

data — a string expression to be sent out by SENDIO.

Actions Taken

SENDIO sends commands and data to HP-IL devices. SENDIO can be executed from the keyboard or in a program. Listener devices may be addressed by including either device codes or device addresses in a SENDIO statement.

SENDIO processes parameters from left to right. One or more device codes may be included in the *device code* field. If device codes are specified, SENDIO determines the HP-IL address of each specified device. If the *device code* field is null, no action is taken.

A single LAD# command in the command field causes all devices specified in the *device code* field to be addressed as listeners. The LAD# command may be used in combination with other HP-IL commands, and may appear anywhere in the *command* field. Listener devices may also be addressed by including LADn commands in the *command* field. Any number of LADn commands may be included, and they may be used in combination with other HP-IL commands, including LAD#. SENDIO should not be used to address talkers.

Once all commands in the command field have been sent, the string expression in the *data* field is sent out over the loop.

One or two of the quoted parameters may be the null string, but not all three.

Related Statements

ENTIO$
SEND?

SPOLL

Syntax

```
SPOLL (':device code')
```

Sample Statements

```
50 P = SPOLL (B$)
250 IF SPOLL (':D1') > 63 THEN GOTO 750
```

Parameters

device code — a valid HP-IL device code. You may substitute the name of a string variable that contains the desired device code.

Actions Taken

Polls a device in the loop by sending the SST (Send Status) message. Returns a number representing the first status byte sent by the polled device.

Related Statements

```
PPOLL
SPOLL$
```

SPOLL$

Syntax

```
SPOLL$ (':device code')
```

Sample Statements

```
40 S$ = SPOLL$ (B$)
90 E$ = SPOLL$ (':D1')
```

Parameters

device code — a valid HP-IL device code. You may substitute the name of a string variable that contains the desired device code.

Actions Taken

Polls a device in the loop by sending the SST message. Returns a string of ASCII characters representing the status bytes sent by the polled device.

Related Statements

```
PPOLL
SPOLL
```

TRIGGER

Syntax

TRIGGER [' :*device code* [, :*device code*]... ']

Sample Statements

```
70 TRIGGER ':D1,:D2'
190 TRIGGER S1$
250 TRIGGER
```

Parameters

device code — a valid HP-IL device code. You may substitute the name of a string variable that contains the desired device code(s).

Actions Taken

Sends the Group Execute Trigger command (GET).

If no device code is given, the GET command is sent. All devices that have already been addressed to listen will receive the GET command.

If a device code is specified, the UNL (Unlisten) command is sent, followed by the LAD (Listen Address) of the specified device(s). The GET command is then sent. Devices are left addressed to listen.

Related Statements

```
LOCAL
LOCAL LOCKOUT
REMOTE
```

Appendix C

HP-IL Commands

Summary of HP-IL Commands

The following is a list of HP-IL command mnemonics for the commands that you may use in a SENDIO or ENTIO$ *command list*. Although SENDIO and ENTIO$ do not recognize the mnemonics of other HP-IL commands, you may include other commands in a *command list* by using extended HP-IL command capability.

> **Note:** The commands CL+, DA-, EL+, ET-, SZ=, TR!, TR*, TR:, and TRC may be included in a *command list* for either ENTIO$ or SENDIO; however, only ENTIO$ will recognize them.

n	Represents a one byte non-negative integer.
AADn	Auto Address: addresses the loop starting with initial address n (0-30).
AAU	Auto Address Unconfigure: resets addresses of the loop to the unassigned state.
AEPn	Auto Extended Primary: assigns primary address n (0-30) to extended address group.
AESn	Auto Extended Secondary: assigns secondary address starting with n (0-30).
AMPn	Auto Multiple Primary: assigns primary addresses to all devices starting with n (0-30).
CL+	The CL+ command inserts **carriage-return/line-feed** in the incoming string after each End Byte message received during ENTIO$ data collection.
DA-	The DA- command prevents the ENTIO$ function from reading any data into the HP-75. ENTIO$ returns the null string if DA- is in the *command list*. However, up to 256 Data Byte messages (or the number set with IOSIZE) will be transmitted from the talker to any active listeners in the loop. If a SZ= command is used to specify a size, that size will take precedence over IOSIZE. If SZ=0 is specified, there is no size limit on the number of Data Byte messages that the talker can send.
DCL	Device Clear: clears all devices in the loop.
DDLn	Device Dependent Listener: sends the Device Dependent Listener command denoted by n (0-31).
DDTn	Device Dependent Talker: sends the Device Dependent Talker command denoted by n (0-31).
EDN	Enable Device Sourcing NRD: enables devices to source own NRD messages.
EL+	The EL+ command inserts the current EOL sequence in the incoming string after each End Byte message received during ENTIO$ data collection (similar to CL+).
ET-	The ET- command disables ENTIO$ termination by an ETO (End Of Transmission - OK) message received from an HP-IL device. ENTIO$ will terminate only when the logical end-of-record is detected, size is exceeded, an ETE (End Of Transmission - Error) message is received, or the [ATTN] key is pressed.

GET — Group Execute Trigger: sets listeners to begin device operation.

GTL — Go To Local: returns listen addressed devices to local control, but leaves them remote enabled. Devices will return to remote mode when next addressed to listen.

IAA — Illegal Auto Address: sent to determine if there are too many devices in the loop.

IEP — Illegal Extended Primary: basically a no-op.

IES — Illegal Extended Secondary: sent to determine if there are too many devices in the loop.

IFC — Interface Clear: clears the interface loop.

IMP — Illegal Multiple Primary: sent to determine if there are too many devices in the loop.

LAD# — Listen Address: activates listener status of device specified in *device code*.

LADn — Listen Address: activates listener status of device at address n (0-30).

LLO — Local Lockout: disables LOCAL button on front panel of device. Device can be returned to local control only by a GTL or NRE command.

LPD — Loop Power Down: puts devices in power down state.

NOP — No Op command.

NRD — Not Ready For Data: controls interrupt of talker.

NRE — Not Remote Enable: returns devices to local control and removes remote enabled status.

PPD — Parallel Poll Disable: causes listen-addressed devices to no longer respond to PPEn.

PPEn — Parallel Poll Enable: enables listen-addressed devices to respond to a parallel poll where n (0-15) sets the state of response (refer to section 5).

PPU — Parallel Poll Unconfigure: disables all devices from responding to PPEn.

REN — Remote Enable: sets devices to remote enabled state. Devices go to remote mode when addressed to listen.

SADn — Secondary Address: enables talkers or listeners with secondary address.

SAI — Send Accessory ID: initiates talker to source accessory ID.

SDA — Send Data: initiates talker to source data.

SDC — Selected Device Clear: clears the active listeners.

SDI — Send Device ID: initiates talker to source device ID.

SST — Send Status: initiates talker to source status byte(s).

SZ= — The SZ= command sets the maximum input size for an ENTIO$ instruction. The default value is 256 (or the value set with IOSIZE). If DA- is not specified in the *command list*, the syntax is: SZ=XXXXX. XXXXX is a decimal number (range 1 to 32767) representing the number of bytes to read. The ENTIO$ instruction terminates when size is exceeded. If DA- is specified, the syntax is: SZ=XXXXXXXXX where XXXXXXXXX is a number in the range 0 to 999999999. If SZ=0 is specified, there is no size limit on the number of bytes to be read. (SZ=0 cannot be specified unless DA- is also specified.)

TAD# — Talker Address: activates talker status of device specified in *device code*.

TADn — Talker Address: activates talker status of device at address n (0-30).

TCT — Take Control: passes control to next controller in the loop.

TL+ — A TL+ command in a SENDIO or ENTIO$ *command list* inhibits the automatic UNT and UNL feature. Devices addressed as talkers and/or listeners will remain active after the SENDIO or ENTIO$ operation is completed.

TR! A TR! command in the *command list* of an ENTIO$ instruction establishes the End Byte message as a logical end-of-record.

TR* A TR* command in the *command list* of an ENTIO$ instruction establishes the current EOL sequence (defined with the ENDLINE statement) as a logical end-of-record.

TR: Any ASCII character can be specified as a logical end-of-record by including TR:XX in an ENTIO$ *command list*, where XX is the hexadecimal representation of the ASCII character number (00 will be ignored).

TR[Any desired character string (up to six characters) may be specified as a logical end-of-record by including TR[*string*] in an ENTIO$ *command list*. Note that the string is delimited with brackets rather than quotation marks, and that the] character cannot be included in the string. If the string contains quotation marks, they must not be the same form (single or double) that is used to delimit the *command list* itself.

UNL Unlisten: deactivates all listeners in the loop.

UNT Untalk: deactivates the talker.

ZES Zero Extended Secondary: assigns secondary addresses to devices with multiple address capability.

Extended HP-IL Command Capability

Extended HP-IL command capability allows the programmer to send commands for which no mnemonics exist. The capability can be used with both SENDIO and ENTIO$. This ensures that when new HP-IL devices and functions are introduced, SENDIO and ENTIO$ will continue to be usable.

> **Note:** By using extended command capability you can include any HP-IL command in a SENDIO or ENTIO$ *command list*. However, you should be careful when you are including a command that is not in the "Summary of HP-IL Commands" in this appendix. Certain unlisted commands may cause problems.

Recall that HP-IL messages consist of 11 bits: a three-bit prefix that identifies the type of message, followed by eight bits of message content. Eight possible prefixes exist, each with its own special meaning. Extended command capability provides an easy way for the programmer to construct HP-IL messages.

Eight identifiers are supplied, one for each type of HP-IL message. The types of messages and corresponding identifiers are listed below:

HP-IL Message Type	Identifier
Command	CD
Ready	RD
Data	DA
End	EN
Identify	ID
Data w/service request	DS
End w/service request	ES
Identify w/service request	IS

To send a message, simply specify "`XX:`*hex value*" in the *command list*, where `XX` is one of the eight identifiers listed above, and *hex value* is the content of the message in hexadecimal. To send an UNL command using extended HP-IL command capability, you would code:

```
SENDIO '','CD:3F',''
```

This would send a message with a three-bit prefix identifying the message as a command, and then a binary "00111111", which is the code for UNL.

Appendix D

Support Functions and Editing Keys

The HP-75 I/O ROM provides several support functions in addition to the I/O functions and statements that are covered in sections 1 through 5 of this manual. These support functions are covered in this appendix under the subheadings "I/O Support Functions," "Advanced Programming Support Functions," and "File Manipulation Functions." This appendix also covers some additional HP-75 editing keys provided by the ROM (refer to "Additional Editing Keys") and a facility for running an autostart program when the HP-75 comes on (refer to "Running an Autostart Program").

> **Note:** The syntax representations in this appendix follow the same conventions that are used elsewhere in this manual. Refer to the subheading "Syntax Guidelines" in section 1.

I/O Support Functions

The following functions are used, in conjunction with the primary I/O functions and statements described in sections 1 through 5, to facilitate I/O operations.

ASNLOOP$ — assign loop and return string:

```
ASNLOOP$
```

Assigns device codes to devices in the loop according to the same rules as `ASSIGN LOOP` (see appendix B), but returns a string. Each character in the string corresponds (in order) to a device in the loop, and represents the first byte of the Accessory ID response of that device.

DISPLAY$ — list current display devices:

```
DISPLAY$
```

Returns a string listing the device codes of the currently assigned display devices (in order of ascending address).

ENABLE SRQ — reenable `ON SRQ` after an `ON SRQ` execution:

```
ENABLE SRQ
```

Resets the active state for an `ON SRQ` statement. Programs that include `ON SRQ` processing of HP-IL SRQ (Service Request) messages must execute `ENABLE SRQ` at the end of the processing to allow another SRQ message to be processed (refer to `ON SRQ`).

ENDLINE$ — return current endline string:

```
ENDLINE$
```

Returns the current EOL sequence (established with the `ENDLINE` statement) as a string.

ESC-I/R ON/OFF — turn modified [I/R] on or off:

```
        ON
ESC-I/R
        OFF
```

This feature defaults to the `ON` state and sends escape sequences to control the cursor of the current `DISPLAY IS` device. When you press the [I/R] key, **ESC Q** is sent to change the cursor on the external display to the insert mode; **ESC R** is sent to return the cursor to replace mode. Type `ESC-I/R OFF` to suppress the output of **ESC Q** and **ESC R**. For some external display devices, you will need to turn this feature off to avoid getting a false echo on the display in the insert mode.

IOSIZE? — return current `IOSIZE` setting:

```
IOSIZE?
```

Returns the current `IOSIZE` setting as a number. The value returned represents the number of bytes that the `ENTER` buffer will hold — except that a zero value indicates that `IOSIZE` is set to its default value (256 bytes).

KEYBOARD$ — return the device code of the current keyboard device:

```
KEYBOARD$
```

Returns the device code of the HP-IL device currently assigned as the keyboard. The null string is returned if no device is assigned.

KEYBOARD IS — assign device for keyboard entry:

```
KEYBOARD IS ':device code'
```

device code — the device code of an HP-IL device to be assigned as the keyboard (may be the device code of an interface to which a keyboard or terminal is connected).

`KEYBOARD IS` can be used to assign an external device as the keyboard. You can assign any keyboard device capable of sending ASCII characters as data bytes. If the keyboard device is not HP-IL equiped, you can connect it to the loop through an appropriate interface. The HP-75 keyboard is not disabled, so you may enter characters from the external keyboard, from the HP-75 keyboard, or both.

All 256 decimal keycodes may be sent from the external keyboard if it is capable of generating them. Refer to the manual for your keyboard device to determine which keys generate which keycodes. The standard ASCII characters (decimal codes 0 through 127) can be transmitted from the external keyboard by simply pressing the appropriate keys. For these characters, the external keyboard uses the same keycodes as the HP-75. For other characters, you will have to determine which key on the external keyboard generates the keycode for the desired HP-75 key. For example, key number 132 on the HP-75 is the [↑] key. If the [↑] key on your external keyboard generates keycode 132, it will **map** directly to the HP-75 [↑] key. However, suppose the **roll-up** key on your external keyboard generates keycode 132. In this case, **roll-up** on the external keyboard maps to [↑] on the HP-75 keyboard.

Most keyboard devices use escape codes to represent editing keys such as the cursor keys, **roll-up**, **roll-down**, etc. The HP-75 can interpret escape codes by means of a `TEXT` file named `KEYMAP`. The `KEYMAP` file contains one line for each key to be mapped. Each line consists of a line number that corresponds to the desired HP-75 keycode and a character that is used to generate it (comments may be appended if desired). The following `KEYMAP` file is given as an example:

```
132 A
133 B
134 D
135 C
```

When an **ESC** character is received from the external keyboard, the next character received is "looked-up" in the `KEYMAP` file. If the character is found, the corresponding line number is used as a keycode. Suppose that your `KEYBOARD IS` device sends **ESC-A** when you press its [↑] key. The HP-75 looks up `A` in the `KEYMAP` file and finds it in line 132. The keycode 132 is generated from the `KEYMAP` file, executing [↑] on the HP-75.

You may also send escape codes from the external keyboard by pressing [ESC] followed by the desired character. If you type [ESC] `A` on the external keyboard, keycode 132 ([↑]) is generated by the HP-75. If you press [ESC] `B`, keycode 133 ([↓]) is generated, and so forth. If you press [ESC] twice on the external keyboard, **ESC** is generated by the HP-75.

Note: The `KEY$` function does not work for an external keyboard defined with `KEYBOARD IS`. The [ATTN] key will not stop a program if `KEYBOARD IS` is active unless the program receives it as part of an input statement. `OFF IO` will disable `KEYBOARD IS` until a `RESTORE IO` is executed. `KEYBOARD IS` will also be disabled if an error occurs while a key is being transmitted. If [LOCK] is pressed, only the HP-75 keyboard, not the external keyboard, will be affected. The computer will not timeout when `KEYBOARD IS` is active.

You may use `DISPLAY IS` to define an external display device as well as `KEYBOARD IS` to define an external keyboard device. If you are connecting a terminal to your HP-75, you may execute `DISPLAY IS` and `KEYBOARD IS` to the same device code (the device code of the terminal or its interface). The terminal will act as a display when characters are sent to it, and as a keyboard when a character is expected by the HP-75. If you are using an external display, you should also refer to "ESC-I/R ON OFF" in this appendix.

LISTIO$ — list HP-IL device codes in string:

```
LISTIO$
```

Returns a string listing the device codes of all HP-IL devices in the loop in order of ascending address. Device codes are preceded by colons and separated by commas, for example: `:M1,:P1`.

OFF SRQ — turn off HP-IL service request response:

```
OFF SRQ
```

Clears the `ON SRQ` statement. This should be done before a program stops, and definitely before the file is edited, purged, or renamed. Failure to do so may cause problems.

ON SRQ — respond to HP-IL SRQ messages:

`ON SRQ` *statement* [`@` *statement*] ...

statement — any statement valid after a `THEN`.

Similar to `ON ERROR` and `ON TIMER`. On receipt of an SRQ (Service Request) message, the program branches to the `ON SRQ` statement (after the entire current line has been executed). Once the `ON SRQ` statement is done, execution returns to the line after the one where the SRQ message was received. `ON SRQ` will not interrupt itself, and must be reenabled with an `ENABLE SRQ` statement before it will again branch. `OFF SRQ` permanently cancels an `ON SRQ` and should be done as part of the end-of-program cleanup routine.

PRINTER$ — list current printer devices:

```
PRINTER$
```

Returns a string listing the device codes of the currently assigned printer devices (in order of ascending address). For example: `:P1,:P2`.

REASSIGN — change device code of an HP-IL device:

`REASSIGN ':`*dev1*`' TO ':`*dev2*`'`

dev1 — old device code.
dev2 — new device code.

Change the device code of the specified device to new device code.

RIO — read data from an HP-IL register:

```
RIO(register number)
```

register number — an HP-IL register number (0 through 7).

Reads data from the specified HP-IL register. STANDBY must be set to ON for RIO to function properly.

WIO — write data to an HP-IL register:

```
WIO register number, data
```

register number — an HP-IL register number (0 through 7).
data — byte of data to be written (MOD 256 is performed).

Writes *data* byte to specified HP-IL register. STANDBY must be in the ON state for proper operation.

Advanced Programming Support Functions

The functions that follow are useful not only in I/O programming, but in advanced programming applications in general.

> **Note:** Functions that manipulate ASCII strings will accept any ASCII character in an input string. Upper and lower case letters have different ASCII decimal codes and are interpreted as different ASCII characters. Functions that manipulate hexadecimal strings will accept the characters 0 through 9, A through F, and a through f in an input string (upper and lower case letters are equivalent in a hexadecimal string).

AAND$ — AND of two strings:

```
AAND$('string 1','string 2')
```

string 1 and *string 2* — ASCII character strings.

A bit-by-bit logical AND is performed on the bit patterns of the corresponding characters of the two strings (the strings are left justified). The output string consists of ASCII characters that represent the resulting bit patterns. The length of the resulting string is equal to the shorter input string.

ADJUST — set adjust factor for clock:

```
ADJUST 'factor'
```

factor — a string that starts with a + or − and contains exactly 14 hexadecimal characters that represent the adjust factor.

Sets the clock adjust factor to the specified value. Specify + to make the clock run faster or − to make the clock run slower. The string must meet the size and format requirements, and the minimum absolute value that may be entered is `100H`. A smaller value (except 0) will cause an error. A zero value will negate the clock adjustment. The value specifes the number of 2^{-14} second intervals between 1/4 second adjustments (+/−) to the system clock. The proper sequence follows:

1. Set the time.
2. Execute `EXACT` twice to set the flags.
3. Execute `ADJUST` to set the factor.

ADJUST$ — show current clock adjust factor:

```
ADJUST$
```

Returns a string that starts with + or − and contains 14 hexadecimal digits representing the current adjust factor. + means the clock is slow (adjusting to a faster rate). − means the clock is fast (adjusting to a slower rate). A zero value means no adjustment is being made (clock running on time).

AOR$ — OR two strings:

```
AOR$('string 1','string 2')
```

string 1 and *string 2* — ASCII character strings.

A bit-by-bit logical OR is performed on the bit patterns of the corresponding characters of the two strings. Trailing characters of the longer string are ORed with `CHR$(0)`. The output string consists of ASCII characters that represent the resulting bit patterns.

AROT$ — rotate a string left or right by bit count:

```
AROT$('string',count)
```

string — ASCII character string to be rotated.
count — number of bits to rotate (to right if +, to left if −).

Rotates an ASCII string on a bit level, considering the string to be a binary number with a length that is a multiple of eight bits. Rotates the bits of the given string by the number of bits specified in the bit *count*. Bits rotated off one end are added on at the other end. Returns an ASCII character string that represents the rotated bit pattern. The resulting string will have the same length as the input string.

ASC$ — convert hexadecimal string to ASCII:

```
ASC$('hex string')
```

hex string — string of hexadecimal characters.

Converts hexadecimal characters to ASCII decimal codes, then returns the string of ASCII characters. Note that two hexadecimal characters specify one ASCII character. If the input string does not have an even number of hexadecimal digits, a leading zero is added.

ASCII$ — return string of ASCII characters in specified range:

```
ASCII$('start','end')
```

start — starting ASCII character. The null string specifies `CHR$(0)`.
end — ending ASCII character. The null string specifies `CHR$(255)`.

Returns a string of ASCII characters in the specified range (inclusive). If *start* is greater than *end*, the string is reversed.

ASHF$ — shift a string left or right by bit count:

```
ASHF$('string',count,bit)
```

string — string of ASCII characters to be shifted.
count — number of bits to shift (to right if +, to left if−).
bit — value to shift into the bit pattern (1 or 0).

Operates on an ASCII string at a bit level, considering the string to be a binary number with a length that is a multiple of eight bits. Shifts the bit pattern left or right by the bit count, shifting in 0's or 1's as specified by the *bit* parameter. If *count* is +, the bit pattern is shifted right, and leading 0's or 1's are shifted into the pattern. If *count* is −, the bit pattern is shifted left, and trailing 0's or 1's are shifted into the pattern. Returns an ASCII character string that represents the shifted bit pattern. The resulting string will have the same length as the original string. An example should clarify this:

```
ASHF$('W',1,0)
```

The *string* is the ASCII character `W` (decimal code 87). The bit pattern for `W` is "01010111". The *count* is 1, a positive number, so the bit pattern is shifted to the right one space. The *bit* value is "0", so 0's are shifted in to replace the leading characters. The resulting bit pattern is "00101011" (note that bits shifted past the end are lost). The corresponding decimal code is 43, and the returned string is the character +.

AXOR$ — exclusive OR of two strings:

```
AXOR$('string 1','string 2')
```

string 1 and *string 2* — ASCII character strings.

Performs a bit-by-bit logical EXOR on the bit patterns of the corresponding characters of the two strings. Each trailing character of the longer string is EXORed with `CHR$(255)`. The output string consists of ASCII characters that represent the resulting bit patterns.

BINAND — bit-by-bit logical AND of two integers:

```
BINAND(integer,integer)
```

integer — range: −32768 to +32767

Returns the 16-bit logical AND of two integers. Each bit of the result is calculated using the corresponding bit of each argument.

BINCMP — binary complement of integer:

```
BINCMP(integer)
```

integer — range: −32768 to +32767

Returns the 16-bit binary complement of an integer. Each bit of the result is the inverse of the corresponding bit in the argument. If the argument has less than 16 bits, leading zeros are assumed.

BINEOR — bit-by-bit exclusive OR of two integers:

```
BINEOR(integer,integer)
```

integer — range: −32768 to +32767

Returns the 16-bit binary exclusive OR of two integers. Each bit of the result is calculated using the corresponding bit of each argument.

BINIOR — bit-by-bit inclusive OR of two integers:

```
BINIOR(integer,integer)
```

integer — range: −32768 to +32767

Returns the 16-bit binary inclusive OR of two integers. Each bit of the result is calculated using the corresponding bit of each argument.

BIT — test bit in integer:

```
BIT(integer, position)
```

integer — range: −32768 to +32767
position — bit position to be tested (0 to 15). Bit number zero is the rightmost bit.

Returns value of specified bit in an integer argument. Result is "1" if bit is set, "0" if bit is clear.

BREAK — find next position of character in list:

```
BREAK('list', 'target', start)
```

list — string of characters to be accepted in search.
target — string to be scanned.
start — position in target string to scan from.

The *target* string is scanned from the specified starting position until a character from the *list* string is found. Returns the position number of that character. If no listed character is found, returns 0.

BTD — convert binary string to decimal number:

```
BTD('string')
```

string — string to be converted (represents binary number) range "0" to "1111111111111111".

Returns decimal value of binary representation contained in the string argument.

BUF$ — return contents of specified buffer:

```
BUF$('buffer')
```

buffer — I (input buffer) or L (LCD buffer).

The entire contents of the specified buffer are returned. The returned string is 96 characters long.

CALL — call basic program with parameters:

```
CALL 'filename[:device code]'[;](parameters)
```

filename — name of program. If a string variable is used to name the file, a semicolon must precede the parameters list. Otherwise the semicolon is optional.
device code — device code of device where program is located.
parameters — list of actual parameters to pass.

A mainframe extension that allows the passing of variables to and from the subprogram named in a CALL statement. This statement calls a basic program and passes the variables to it. The results are passed back through the same variables. The variables may be passed in two forms:

- Passed by reference: Provides bidirectional access to the values of the variables. Values of variables may be updated by the subprogram, and such updates are reflected immediately in the main program. For example: A, X, D$(,), and G() are all passed by reference.
- Passed by value: Provides unidirectional access to the values of the variables. The values of the variables in the calling program remain static during the execution of the subprogram. All expressions and subscripted variables are passed by value. For example: X*Y/Z, A$[1,5], C(2,1), and (X) are all passed by value.

An example of a CALL statement (with parameters) would be:

```
CALL 'Aprog' (A,A5$[1,5],G$(1,1),(X))
```

COPY ':BCRD' — recover bad card with missing tracks:

```
COPY 'filename:BCRD[/password]' TO 'filename'
```

filename — a valid filename for a BASIC or TEXT file.
password — the password of a private file on the card.

COPY ':BCRD' works just like COPY ':CARD' unless you press [ATTN] or [SHIFT] [ATTN] before all of the tracks of the card have been read. The *filename* parameter is required for COPY ':BCRD', and must match the name on the card (use CAT CARD to determine the proper name). When the copy process is allowed to go to normal completion, the result will be a normal copy. If there are errors, the partial file is purged, just as with COPY ':CARD'. However, if the copy is aborted with the [ATTN] key, the file copied up to that point is manipulated into a valid file and retained. The new file will contain as many lines of the original file as could be recovered. This process only works for BASIC and TEXT files.

Note: If you are using a KEYBOARD IS device, you cannot use the external keyboard to abort COPY ':BCRD'. You must press the [ATTN] key on the HP-75 keyboard.

COUNT? — show current length of DISP or PRINT output:

```
COUNT?('flag')
```

flag — D (DISP), or P (PRINT).

Returns the number of characters in the DISP or PRINT buffer (since the last time **carriage-return** was sent).

> **Note:** This function will not operate correctly for the DISP buffer if WIDTH is set to **INF**; for the PRINT buffer if PWIDTH is set to **INF**.

DEFKEY$ — return current key definition:

```
DEFKEY$('character')
```

character — character representing key wanted (may be specified with the CHR$ function).

Returns the key definition string for the specified key as stored in the keys file. If the key was defined with a trailing semicolon, the first character will be a semicolon. Otherwise the first character will be blank.

DELAY? — return current delay setting:

```
DELAY?
```

Returns the current delay setting. The returned value may not be exact due to some internal round-off error. For example: DELAY .6 @ DISP DELAY? returns .599975585938.

DO ERROR — cause given error:

```
DO ERROR [error#]
```

error# — number of error to cause.

Causes the specified error condition to occur. If the *error#* field is left blank, the last error is caused. Program execution is stopped, ERRN is set to the specified error number, and the error message is displayed. ROM errors will not display error messages, but ERROR: error# will be displayed. Refer to appendix E for I/O ROM error definitions.

DTB$ — convert decimal number to binary string:

```
DTB$(number)
```

number — number to convert (−32768 to 32767).

Rounds decimal number to the nearest integer and returns the binary representation as a string.

DTH$ — convert decimal number to hexadecimal string:

```
DTH$(number)
```

number — number to convert (−32768 to 32767).

Rounds decimal number to the nearest integer and returns the hexadecimal representation as a string.

DTO$ — convert decimal number to octal string:

```
DTO$(number)
```

number — number to convert (−32768 to 32767).

Rounds decimal number to the nearest integer and returns the octal representation as a string.

ESC$ — return string of escape-character sequences:

```
ESC$('string')
```

string — string to be escaped.

Returns string with ESC added in front of each character.

EXIT — leave a FOR-NEXT loop early:

```
EXIT index variable name
```

index variable name — the name of the `FOR` variable to be exited.

Causes program execution to branch to the statement following the `NEXT` that corresponds to the *index variable name.* For example: `EXIT X` would cause a branch to the statement following `NEXT X`. If `EXIT` is included in a multiple-statement line, statements that precede the `EXIT` will be executed, but the `EXIT` will cause an immediate branch, skipping the statements that follow it in the line. If `NEXT` is in a multiple-statement line, execution will continue with the statement after the `NEXT` **in that line.**

FILL$ — fill a string:

```
FILL$('left','middle','right',size)
```

left — left fill string.
middle — string to fill around.
right — right fill string.
size — size of string to be returned.

Places the *middle* string in a string of the specified *size*, and fills in on the left and right sides with the *left* and *right* strings, respectively. Each fill string is duplicated (if necessary) to fill the space from the left or right margin to the *middle* string. Odd pieces of the fill string will bracket the *middle* string since the fill is from the edges in, both sides. If both *left* and *right* strings are specifed, the *middle* string will be centered (odd space to the right). If the *left* string is null, the *middle* string will be left justified. If the *right* string is null, the *middle* string will be right justifed. If both strings are null, the *middle* string will be right and left justifed (spaces will be expanded to fill the size). If the *middle* string is longer than the size, then the *middle* string is returned truncated to that size.

FIND — find specified occurrence of substring in string, with wild card:

```
FIND('subject','target','[wild]',occur)
```

subject — substring to find (with wild cards).
target — string to scan for occurrence of subject substring.
wild — character to use as wild card in subject substring.
occur — an integer specifying the desired occurrence of the subject substring.

Finds the specified occurrence of the *subject* substring in the *target* string. The *wild* character (if specified) will match any character, and overlapping occurrences are counted. If the pattern is not found, the returned value is zero, otherwise it is the position of the first character of the match. For example, in `HHHH` the second occurrence of `HHH` is at position 2 and there is no third occurrence. This match could also be made with the *subject* string `H--`, where `-` is the declared *wild* character.

FLAG$ — set specifed bit to specified value in given string:

```
FLAG$('flag string',bit#,value)
```

flag string — string being used as an array of flag bits.
bit# — number of bit to set (negative numbers default to zero).
value — 0 or 1. Set the bit to the specified value.

This will set the specifed bit to the specified value and return the new string. If the bit is outside the current string length, an error will result. The flag string may be initialized with `ASC$`, for example: `F$=ASC$('00FFA')`. Bit number zero is at the extreme right.

FLAG? — test specified bit in string:

```
FLAG?('flag string', bit#)
```

flag string — string being used as an array of flag bits.
bit# — number of the bit to be tested, (negative numbers default to zero).

Returns 0 if bit is clear, 1 if bit is set. Bit number zero is at the extreme right.

FOR — `FOR` allowed after a `THEN` or `ELSE`:

The I/O ROM provides a modified `FOR` that works just like the mainframe `FOR`, except that it is allowed after a `THEN` or an `ELSE` in a multiple-statement line. `FOR` may be used in multiple-statement lines as shown in the following two examples:

```
30 IF F>=2 THEN FOR X=1 TO 5 @ F=2*F @ DISP F @ NEXT X
70 IF F=2 THEN GOTO 90 ELSE FOR X=1 TO 10 @ F=2*F/PI @ DISP F @ NEXT X
```

The I/O ROM is required only while such a statement is being written into a program. Once the program has been written, it can be run even if the ROM has been removed.

GOSUBX — `GOSUB` to a variable as a line number:

```
GOSUBX numeric expression
```

numeric expression — numeric expression to be evaluated and used as line number. Expression is rounded to an integer (MOD 10000). Negative numbers default to zero.

Performs a `GOSUB` to the line number derived from the numeric expression, or the line after that if that line does not exist.

GOTOX — `GOTO` to a variable as a line number:

```
GOTOX numeric expression
```

numeric expression — numeric expression to be evaluated and used as line number. Expression is rounded to an integer (MOD 10000). Negative numbers default to zero.

Performs a `GOTO` to the line number derived from the numeric expression, or the line after that if that line does not exist.

HAND$ — AND of two hexadecimal strings:

```
HAND$('string 1','string 2')
```

string 1 and *string 2* — two hexadecimal strings.

A bit-by-bit logical AND is performed on the bit patterns of the corresponding characters of the two strings (the strings are left justified). The output string consists of hexadecimal characters that represent the resulting bit patterns, and is equal in length to the shorter input string. If an input string does not have an even number of hexadecimal digits, a leading 0 is added (before left justification).

HEX$ — convert ASCII string to hexadecimal:

```
HEX$('ASCII string')
```

ASCII string — string of ASCII characters.

Returns string of hexadecimal characters that represent the bit pattern specified by the ASCII string.

HOR$ — OR two hexadecimal strings:

```
HOR$('string 1','string 2')
```

string 1 and *string 2* — hexadecimal character strings.

A bit-by-bit logical OR is performed on the bit patterns of the corresponding characters of the two strings. Trailing characters of the longer string are ORed (in pairs) with "00". The output string consists of hexadecimal characters that represent the resulting bit patterns. If an input string does not have an even number of hexadecimal digits, a leading zero is added to it before the OR is performed.

HROT$ —rotate a hexadecimal string left or right by bit count:

```
HROT$('string',count)
```

string — hexadecimal character string to be rotated.
count — number of bits to rotate (to right if +, to left if −).

Rotates a hexadecimal string on a bit level, considering the string to be a binary number with a length that is a multiple of eight bits. (If the input string does not contain an even number of hexadecimal digits, a leading zero will be added.) Rotates the bits of the given string by the number of bits specified in the bit count. Bits rotated off one end are added on at the other end. Returns hexadecimal character string that represents the rotated bit pattern.

HSHF$ — shift a hexadecimal string left or right by bit count:

```
HSHF$('string',count,bit)
```

string — string of hexadecimal characters to be shifted.
count — number of bits to shift (to right if +, to left if−).
bit — value to shift into the bit pattern (1 or 0).

Operates on a hexadecimal string at a bit level, considering the string to be a binary number with a length that is a multiple of eight bits (if the input string does not have an even number of hexadecimal digits, a leading zero will be added). Shifts the bit pattern left or right by the bit count, shifting in 0's or 1's as specified by the *bit* parameter. If *count* is +, the bit pattern is shifted right, and leading 0's or 1's are shifted into the pattern. If *count* is −, the bit pattern is shifted left, and trailing 0's or 1's are shifted into the pattern. Returns a hexadecimal character string that represents the shifted bit pattern. An example should clarify this:

```
HSHF$('A5B',-3,1)
```

First, a leading zero is added to make an even number of hexadecimal digits. The string becomes `0A5B`. The bit pattern for this string is "0000 1010 0101 1011" The *count* is −3, so the bit pattern is to be shifted three spaces left, with 1's shifted in on the right. The shifted bit pattern is "0101 0010 1101 1111". The hexadecimal string that represents the shifted pattern is `52DF`, and this string is returned by `HSHF$`.

HTD — convert hexadecimal string to decimal number:

```
HTD('string')
```

string — hexadecimal string to convert, range "0" to "FFFF". Limited to the characters "0" through "9", "A" through "F", or "a" through "f".

Returns the decimal numeric value of a base 16 representation contained in the string argument.

HXOR$ — EXOR two hexadecimal strings:

```
HXOR$('string 1','string 2')
```

string 1 and *string 2* — hexadecimal character strings.

A bit-by-bit logical EX0R is performed on the bit patterns of the corresponding characters of the two strings. Trailing characters of the longer string are EXORed (in pairs) with "FF". The output string consists of hexadecimal characters that represent the resulting bit patterns. If an input string does not have an even number of hexadecimal digits, a leading zero is added to it before the EXOR is performed.

INSTALL — load private file from tape (created by MCOPY):

```
INSTALL 'filename:device code'
```

filename — filename of desired file.
device code — device code of desired tape drive.

Copies a private file (created by MCOPY) from tape to RAM. This is the only way to retrieve a private MCOPY tape file (refer to MCOPY).

LCD ON/OFF — turn LCD on/off:

```
    ON
LCD
    OFF
```

LCD ON specifies normal LCD operation. LCD OFF prevents anything further from being displayed on the LCD. LCD OFF remains in effect until LCD ON is executed or the program stops.

LEFT$ — return left portion of string:

```
LEFT$('string',count)
```

string — input string (left part to be returned).
count — number of characters to be returned.

Returns the number of characters specified, starting from the left end of the string. If *count* is greater than the length of the string, the right end is padded with blanks.

LTRIM$ — left trim a string:

```
LTRIM$('trim','target')
```

trim — list of characters to trim.
target — string to be trimmed.

Trims the listed characters off the left edge of the string until a character is encountered that is not in the trim list.

LWRC$ — convert string to lowercase:

```
LWRC$('string')
```

string — string to be converted.

The characters "A" through "Z" are converted to lowercase. Other characters are not changed.

MAP$ — map "from" characters into "to" characters in target string:

```
MAP$('from','to','target')
```

from — list of characters to find.
to — list of characters to replace the *from* characters.
target — string to operate on.

Scans target string, searching for any *from* characters. Each *from* character found is replaced with the corresponding character from the *to* list. All other characters are passed through unchanged. For example: `MAP$('bac','de','abcfde')` will return the string `edfde`. `MAP$` maps `a` into `e` and `b` into `d`. The `c` goes to null, and `fde` is passed through. Note that `MAP$` differentiates between upper and lower case characters. For example: `MAP$('Aa','bc','Aardvark')` returns the string `bcrdvcrk`.

MARGIN? — return current right margin setting:

```
MARGIN?
```

Returns the current right margin setting as a decimal number.

MCOPY — duplicate tape onto multiple tapes:

```
          N                      :slave[,:slave]...
MCOPY '[     ]:master' TO '                        '
          P                      ALL
```

master — device code of source tape drive (`N`=normal, `P`=private).
slave — device code of a destination tape drive (ALL will find all of the drives).

Copies the entire contents of the master tape onto all of the destination tapes. Tapes are first initialized unless the colon before *master* is replaced with a period. The resulting tapes will be made private if you specify a `P` in the `MCOPY` statement (only BASIC and LEX files will be private). The files of the `MCOPY` tape can be read into memory with the `INSTALL` command (see `INSTALL`).

> **Note:** The slave tapes will be exact copies of the master tapes. You cannot use `MCOPY` to append data to an existing tape. You should only specify a period before *master* if you have already initialized the destination tapes.

MID$ — return middle portion of string:

```
MID$('string',start,count)
```

string — string of which to return middle portion.
start — starting position.
count — number of characters to return.

Returns specified number of characters from the given string, starting from the *start* position. If the count passes the end of the string, blanks are appended to the end.

NEXT — `NEXT` allowed after a `THEN` or `ELSE`:

The I/O ROM provides a `NEXT` that works just like the mainframe `NEXT`, except that it may be used after a `THEN` or `ELSE` in a multiple-statement line. For more details, refer to `FOR`.

NSCR$ — remove underscoring:

`NSCR$('`*string*`')`

string — string to be modified.

Removes the underscore bit from all characters in the string and returns the string without the underscoring.

OTD — convert octal string to decimal number:

`OTD('`*octal*`')`

octal — string to be converted, range "0" to "177777".

Returns the decimal numeric value of the octal representation contained in the string argument.

PWIDTH? — return current `PWIDTH` setting:

`PWIDTH?`

Returns the current `PWIDTH` setting as a number. Returns 9.99999999999E499 if the setting is `INF`.

REPL$ — replace substring in target string with another:

`REPL$('`*from*`','`*to*`','`*target*`','`[*wild*]`',`*occur*`)`

from — old substring to replace.
to — new substring.
target — string to scan.
wild — character to use as a wild card in the from substring.
occur — an integer specifying the occurrence of the from substring to replace.

Scans the target string for the specified occurrence of the *from* substring. The *wild* character (if specified) will match any character, and overlapping occurrences are counted. If a match (with or without a wild character) is found, the specified occurrence of the *from* substring will be replaced with the *to* substring (or deleted if the *to* substring is null). If the *from* substring is null, the *to* substring will be inserted in front of the *occur* character in the *target* string. If no match is found, the *target* string is returned unchanged. For example: `REPL$('a--','b','aaaef','-',3)` will return the string `aab`. The first, second, and third occurrences of `a--` are `aaa`, `aae`, and `aef`, respectively. The third occurrence, `aef`, is replaced with `b`.

REV$ — reverse string:

```
REV$('string')
```

string — string to be reversed.

Returns reversed string, (`ABCD` becomes `DCBA`).

RIGHT$ — return right portion of string:

```
RIGHT$('string',count)
```

string — string of which right portion is to be returned.
count — number of characters to return.

Returns the specified number of characters at the right end of the string. If the count is greater than the string length, blanks are added on at the left end.

ROT$ — rotate string by character count:

```
ROT$('string',count)
```

string — string to be rotated.
count — number of spaces to rotate (to right if +, to left if −).

String is rotated right or left by specified count. Characters rotated off one end are added on at the other end. Returns rotated string. For example: `ROT$('ABCD',-1)` returns the string `BCDA`.

RPT$ — repeat string.

```
RPT$('pattern',count)
```

pattern — pattern to be repeated.
count — number of times to repeat the pattern.

Concatenates *pattern* the number of times specified by *count* and returns the resulting string. `RPT$('AB',3)` returns the string `ABABAB`.

RTRIM$ — trim trailing characters:

```
RTRIM$('trim','string')
```

trim — list of characters to trim.
string — string to be trimmed.

Trims trailing characters listed in the *trim* list. All listed characters to the right of the last non-listed character are trimmed. For example: `RTRIM$(', .','abc,de,..,')` returns the string `abc,de`.

SHELL — automatic run of programs by name:

```
      ON
SHELL
      OFF
```

Turns SHELL mode on or off. If SHELL mode is on, CALL '*filename*' is automatically executed for any line that is a valid filename for a BASIC file. For example, if there is a BASIC file named APROG in memory, typing APROG [RTN] will cause CALL 'APROG' to be executed. SHELL mode also can be used to execute a CALL with parameters (refer to CALL). For example, typing BPROG(A,X) [RTN] will cause CALL 'BPROG'(A,X) to be executed. Note that BPROG(A,X) must be typed with **no** embedded blanks.

SKEY$ — wait for significant key:

```
SKEY$
```

SKEY$, like KEY$, returns the character associated with any pressed key or keystroke combination, allowing "live" keyboard branching. However, SKEY$ does not return a character until a key is pressed (KEY$ will return the null string if no key is depressed while it is being executed). This allows a running program to "wait" for a pressed key.

There are some keys that do not cause SKEY$ to return a character. You may press [SHIFT] [FET] to fetch an error message if an error occurs before the SKEY$ statement. Also, the [←] and [→] keys (and their variations) are not returned, but scroll the LCD.

SPAN — find position of first character not in list:

SPAN('*list*','*target*',*start*)

list — list of characters to pass over.
target — string to be scanned.
start — starting position in target string.

Scans *target* string and returns the position number of the first character found that is not in the *list* string. The scan starts at the specified *start* position, and continues to the end of the string. If no unlisted character is found, zero is returned. The function is inclusive. If the starting character is not listed, the *start* position is returned.

STATUS — set status of system flags:

```
STATUS 'flagset'
```

flagset — 12 character string. Characters indicate settings for flags:

1. A = ALARM ON, a = ALARM OFF
2. L = AUTOLOOP ON, l = AUTOLOOP OFF
3. I = ESC-I/R ON, i = ESC-I/R OFF
4. S = SHELL ON, s = SHELL OFF
5. B = BEEP ON, b = BEEP OFF
6. D = DEFAULT ON, d = DEFAULT OFF
7. S = STANDBY ON, s = STANDBY OFF
8. T = TIMEOUT ON, t = TIMEOUT OFF
9. V = VERIFY ON, v = VERIFY OFF
10. D = DEGREES, R = RADIANS
11. T = TRACE FLOW/VARS, F = TRACE FLOW,
    V = TRACE VARS, t = TRACE OFF
12. M = MDY mode, D = DMY mode
13. A = AM/PM mode, * = 24 hour mode

Any flag may be left in its present state by including a period (.) as a place holder in the string. Strings shorter than 13 characters do not change trailing flags. For example: STATUS 'A...bD' sets ALARM ON, leaves AUTOLOOP, ESC-I/R, and SHELL in their present state, sets BEEP OFF, sets DEFAULT ON, and leaves the trailing flags in their present state.

STATUS$ — show current system flag settings:

```
STATUS$
```

Returns flag string representing system flag settings as set with STATUS. The format is the same as for STATUS (see above).

STRING ARRAYS — dimensioning and referencing:

The I/O ROM provides the capability to declare string arrays. String arrays may be one or two dimensional, and consist of string elements of specified length. The syntax of the `DIM` (dimension) statement is:

`DIM A$(`*col*`,`*row*`)`[*size*]

col — column upper bound.
row — row upper bound.
size — size of element (all elements have the same size).

Dimensioning a string array is similar to dimensioning a numeric array. The column and row upper bounds are specified in the `DIM` statement, but the actual number of elements is affected by `OPTION BASE` just as for numeric arrays. The following `DIM` statement would dimension a one-dimensional string array with six elements, each a string 10 characters long (assuming the default of `OPTION BASE 0`):

```
10 DIM A$(5)[10]
```

You can reference a dimensioned string array as follows:

`B$ = A$(`*col*`,`*row*`)`[*start*`,` [*stop*]]

col — column specifier.
row — row specifier.
start — start position in element.
stop — stop position in element.

If you do not specify a *start* and *stop* position, the entire element is copied. For example, `B$ = A$(1,5)` copies the element `A$(1,5)` into `B$`. If *start* and/or *stop* are specified, only the specified portion of the element is copied. For example, `B$ = A$(1,5)2,4` copies characters two through four of the element `A$(1,5)` into `B$`.

SUB — header for subprogram:

`SUB` *name*`(`*formal parameters*`)`

name — name of subprogram.
formal parameters — list of parameters to be passed.

Each subprogram must have a `SUB` statement as the first line in the file (only one subprogram may be in a file). `SUB` defines the beginning of the subprogram and the parameters expected by the subprogram. Parameters within the subprogram must match the passed parameters in type. Formal parameters must be used, for example: `X`, `A1(,)`, `C$`, and `F1$(,)`. The *name* field must match the *filename* of the subprogram. The `SUB` statement is used in conjunction with `CALL`.

SUB$ — return middle portion of string:

```
SUB$('string', left, right)
```

string — string to process.
left — left position.
right — right position.

Returns the portion of the string bounded by the *left* and *right* positions (inclusive). If *left* is negative, blanks are added in front. If *right* is larger than the string, blanks are added at the end.

**TCAT$** — CAT$ of a tape drive:

```
TCAT$(':device code', file#)
```

device code — device code assigned to tape drive.
file# — number of desired file.

Returns catalog entry for the specified file as a string (like CAT$). If file does not exist on tape, returns null string.

TEMPLATE$ — return template string with protected fields:

```
TEMPLATE$('protect templ', 'trail')
```

protect templ — protected template string up to 96 characters long.
trail — trailing field flag (P = protected, U = unprotected).

Returns a protected template string with unprotected fields that the user may change. Specify protected fields with underlined characters (use [CTL] [I/R]). The underlining will not appear in the returned string. Use characters without underlining to specify unprotected fields. The trailing field may be protected, or left unprotected, by specifying P or U for *trail*. For example:

```
TEMPLATE$('Time_=_hh:mm__Temp_=_dd_F','P')
```

returns the string Time = hh:mm Temp = dd F. You can change the fields hh, mm, and dd, but all other characters are protected. The trailing field is also protected because P is specified. You can tab right and left from field to field with [TAB] and [SHIFT] [TAB]. The [CLR] key restores the original template. When input is terminated with [RTN], the entire 96 character string (with user changes) is returned. Termination with any other terminator (such as [ATTN]) causes the null string to be returned.

TIMEOUT ON/OFF — set timeout mode:

```
         ON
TIMEOUT
         OFF
```

ON — allow timeout after five minutes.
OFF — prevent timeout after five minutes.

STANDBY ON/OFF will affect this setting. If TIMEOUT ON is done after a STANDBY ON, the HP-75 will stay fully on for five minutes, then turn itself off. If TIMEOUT OFF is done after a STANDBY OFF, the HP-75 will go into the partial power down state almost immediately, and will stay in this state indefinitely. Normally you would want to execute STANDBY OFF first if you are using TIMEOUT ON/OFF.

TIMER? — return current timer interval setting:

TIMER?(*timer number*)

timer number — number of timer to be checked.

Returns the value of the specifed timer's interval. Zero is returned if the timer is not declared.

TOBASE$ — convert number to specified base, return as string:

TOBASE$(*number*, *base*)

number — decimal number (floating point format) to be converted.
base — positive integer (range: 2 through 36).

Converts decimal number to the specified base (2 through 36). Returns result as a string. Maximum string length is 256 characters. Issues warning if the string is too long.

TODEC — convert string from specified base to decimal number:

TODEC('*string*', *base*)

string — string representing number to convert. Valid characters are: 0-9, A-Z, and a-z (characters must be valid for the specified base).
base — positive integer (range: 2 through 36).

Returns decimal number in floating point format equivalent to the string representation in the specified base.

USCR$ — underscore string:

```
USCR$('string')
```

string — string to be underscored.

Returns specified string, but with underscored characters.

USERMSG — send message to display and error buffer:

```
USERMSG 'message'[,error number]
```

message — message to be displayed (maximum of 32 characters).
error number — error number to be reported with message.

The specified message is sent to the display and error buffer. The message may be recalled with [SHIFT] [FET] (until the next terminator key is pressed). If *error number* is non-zero and positive, the error annunciator will be turned on, BEEP will sound, and you may recover the number with ERRN. If *error number* is zero or negative, the message will be displayed, but the error annunciator, BEEP, and ERRN will remain unchanged.

VERIFY ON/OFF — set verify mode for card reader:

```
        ON
VERIFY
        OFF
```

ON — turn on verify mode for card reader.
OFF — turn off verify mode for card reader.

WEND? — show current window end:

```
WEND?
```

Returns the current window end column as a number.

WIDTH? — return current WIDTH setting:

```
WIDTH?
```

Returns the current WIDTH setting as a number. Returns 9.99999999999E499 if the setting was INF.

WINDOW — set the LCD window start, end:

WINDOW [*start*[, *end*]]

start — start column: 1 through 32 (defaults to 1).
end — end column: 1 through 32 (defaults to 32).

Sets the start and end columns of the LCD window. The window setting remains until reset. When used in a program, WINDOW may be used to set up a field within which data may be displayed. Anything that is outside the window, and that is sent to the display by a DISP or PRINT statement before the WINDOW statement is executed, will remain "frozen" until the display is cleared by a CR/LF. To avoid clearing the display, append a semicolon (;) to all DISP and PRINT statements, and set WIDTH and PWIDTH to INF. The following program exemplifies the use of WINDOW:

```
10 DISP '*****     *****';
20 WINDOW 6,10
30 DISP '12345';
40 END
```

The program displays *****12345***** when it is run. You may scroll 12345 with the [←] and [→] keys. Type WINDOW [RTN] to return the display to normal.

WKEY$ — wait for key, return any key pressed:

WKEY$

Works like KEY$ except that it will not execute until a key is pressed. Unlike SKEY$, it returns a character for **any** key that is pressed (including [SHIFT] [FET], [←], and [→]).

WSIZE? — show current window size:

WSIZE?

Returns a number representing the number of columns in the current window.

WSTART? — show current window start:

WSTART?

Returns number of the starting column of the current window.

File Manipulation Functions

The following functions provide enhanced file manipulation capabilities.

ADVANCE# — advance data item pointer in a file:

```
ADVANCE# file number; count, return variable
```

file number — number of data file (assigned with ASSIGN#).
count — number of items to skip.
return variable — variable to contain the number of items not skipped.

Moves data item pointer forward in the file specified by *file number.* Skips the number of data items specified by *count*. If the end-of-file marker is encountered before *count* items are skipped, the number of items not skipped (*count* less the number skipped) is returned as the value of *return variable*.

CAT# — return file number of nth ASSIGN# file:

```
CAT#(n)
```

n — 0 to 9999 (negative numbers default to zero).

Returns the file number of the nth ASSIGN# file. Returns zero if the nth file does not exist. If file numbers 1, 5, and 8 have been assigned, CAT#(1) returns 1, CAT#(2) returns 5, and CAT#(3) returns 8. If n = 0 is specified, the next **available** ASSIGN# file number is returned. In the above example, CAT#(0) would return 2.

CLEAR ASSIGN# — clear all ASSIGN# assignments.

```
CLEAR ASSIGN#
```

All ASSIGN# assignments are cleared, recovering space in memory.

DELETE# — delete data items.

```
DELETE# file number, count
```

file number — specifies ASSIGN# file to delete data from.
count — count of items from current position.

Delete specified number of data items from specified ASSIGN# file. Number of items is specified by *count*, beginning at the current position.

FILE$ — show name of specified ASSIGN# file:

FILE$(*file number*)

file number — number of ASSIGN# file (0 specifies the current run file if any reads have been done, a negative number specifies the current edit file).

Returns the name of the ASSIGN# file specified by *file number*. Returns the null string if the *file number* does not exist. Returns underlined name if the file has been assigned, but does not exist.

INDEX# — return current data pointer position in file:

INDEX#(*file number*)

file number — number of ASSIGN# file (0 specifies the current run file if any reads have been done).

This returns the current data pointer position in the specifed file, in terms of the number of **items** from the beginning of the file.

INSERT# — insert an item at the current data pointer:

INSERT# *file number*; *value*

file number — the number of the desired ASSIGN# file.
value — the value to be inserted into the file.

Inserts item into the file **in front of** the item at the current data pointer position. You can use ADVANCE# to position the pointer at the end of the line (after the last item), then insert an item at the end of the line.

ITEM# — return pointer position in current line:

ITEM#(*file number*)

file number — number of ASSIGN# file (0 specifies the current run file if any reads have been done).

Returns the pointer position in the current line, (the number of items from the beginning of the line). Returns an error if the file has been purged.

LASTLN? — return line number of last line in specified file:

LASTLN?('[*filename*]')

filename — name of file to be checked.

Returns the line number of the last line in the specified file. If you specifiy the null string for *filename*, the line number of the last line in the current file will be returned.

LINE# — return current line number in specified `ASSIGN#` file:

```
LINE#(file number)
```

file number — number of `ASSIGN#` file (0 specifies the current run file if any reads have been done, a negative number specifies the current edit file).

Returns current line number in the file specified by *file number*. If the file is not assigned, `INF` is returned. If the file has been assigned, but does not exist, a negative line number is returned.

LINELEN# — return the number of items in a line:

```
LINELEN#(file number, line number)
```

file number — number of `ASSIGN#` file.
line number — number of line in `ASSIGN#` file.

Returns the number of items on the specifed line, in the specified file. Text files return the character count of the line.

PRINT# ... USING — `PRINT#` to a TEXT file with `USING` format:

```
PRINT# file number[, line number] USING image list / line number ; expression[, expression]...
```

file number — `ASSIGN#` file number (must be a TEXT file).
line number — line number to print to.
image list or *line number* — a valid list of image specifiers or the line number of a statement containing the image list.
expression — item to print (a numeric or string expression).

`PRINT#` ... `USING` works just like `PRINT` ... `USING`, except that it "prints" to an `ASSIGN#` file.

REPLACE# — replace a data item in a file:

```
REPLACE# file number; value
```

file number — `ASSIGN#` file number.
value — value to replace old value.

Replaces item currently pointed to in the specified `ASSIGN#` file with the new item specified by *value*.

SEARCH# — search for value in data file:

```
SEARCH# file number[, start[, end]]; value
```

file number — ASSIGN# file number.
start — start line number for search.
end — end line number for search.
value — value to search for.

Moves item pointer in specified ASSIGN# file to the first occurrence of the specified value. If *start* is not specified, search starts at the current location. If *end* is not specified, search continues to the end of the file. The pointer does not move and an error is issued if the value is not found.

SEEK# — position item pointer at a given location:

```
SEEK# file number, [line number, ]item number
```

file number — ASSIGN# file number.
line number — line to position pointer in (optional).
item number — item number (in line if line number is specified; otherwise, in file).

Positions item pointer in the specified ASSIGN# file to the specified position. If *line number* is specified, positions pointer to *item number* in the specified line. If *line number* is not specified, *item number* is an absolute item number, and the pointer is placed at that item, counting from the beginning of the file.

Additional Editing Keys

The HP-75 I/O ROM provides several additional editing keys. Some of these keys are redefinitions of existing keys or key sequences, while others are entirely new. These editing keys cannot be reassigned to other keys or key sequences, and the key sequences that execute these keys cannot be redefined with DEF KEY.

[CTL] [CLR] — clear display devices:

Press [CTL] [CLR] to clear all current display devices without affecting the contents of the input buffer. Sends **ESC H** and **ESC J** to the current display devices.

[CTL] [DEL] — delete to beginning of line:

Press [CTL] [DEL] to delete all characters from the beginning of the current edit line to the position just left of the cursor. If there is a line number adjacent the prompt, the beginning of the line is defined as just after the line number. Otherwise, the line begins just after the prompt. The remaining characters are justified left.

[CTL] [I/R] — literalize and underscore next key:

Works like [SHIFT] [I/R], but with the addition of underscoring.

[CTL] [SHIFT] [→] — find next occurrence of character on line:

Press the [CTL], [SHIFT], and [→] keys (holding all three down), release all of them, then press a character key. The cursor will move to the next (right) occurrence of the specified character on the current edit line. The cursor does not move if no occurrence of the character is found.

[CTL] [SHIFT] [←] — find previous occurrence of character on line:

Works like the previous function, except that the cursor moves to the left instead of to the right.

[TAB] — tab left or right in non-protected field:

[TAB] enables you to tab from field to field. Press [TAB] to move right, [SHIFT] [TAB] to move left. Stops on the first character of the next or previous field (delimited by a space, semicolon, comma, or period). For example, in the string `abc def;ghi,jkl.mno` the tab points are `a`, `d`, `g`, `j`, and `m`.

Running an Autostart Program

The HP-75 I/O ROM enables the HP-75 to automatically run a program named `AUTOST` when the computer is turned on (or turns itself on). This facility operates through the definition of key number 159. If a program named `AUTOST` is present when the power is turned on and key number 159 has not been defined, the function executes `DEF KEY CHR$(159), "&RUN 'AUTOST'‡"`, then runs the `AUTOST` file. If key number 159 has been defined, its current definition will be executed when you turn on the power. You can turn the feature off by executing `DEF KEY CHR$(159),''` (establishing a null definition). To turn the feature back on, execute `DEF KEY CHR$(159), "&RUN 'AUTOST'‡"`. If no `AUTOST` program exists and key number 159 has not been defined, the feature remains inactive.

Note: Type [SHIFT] [←] to produce &. Type [SHIFT] [DEL] to produce ‡.

The content of the `AUTOST` program depends on your application. Simply write a program named `AUTOST` that causes the HP-75 to do whatever you want it to do when it is turned on. The program will run the next time the computer is turned on (unless key 159 is defined to do something else). You may also define key 159 to run any desired program or function. For example, if you execute `DEF KEY CHR$(159), 'CATALL'`, `CATALL` will be executed each time the computer is turned on.

Appendix E

Errors and Warnings

The HP-75 I/O ROM displays the following error messages when the listed error conditions occur. Other error messages and warnings are listed in the *HP-75 Owner's Manual.*

Note: Errors 28, 42, 47, 52, 68, 82, 85, 88, 89, and 91 are HP-75 mainframe error messages. These error messages have their usual meanings and may also be used by the HP-75 I/O ROM to indicate the error conditions listed in the following table. Errors 120 through 129 are specific to the I/O ROM.

Number	Message and Condition
28	record overflow IOSIZE is exceeded by the record being entered.
42	string too long Device code of more than two characters entered in a REASSIGN statement.
47	no matching FOR No NEXT can be found to match the index variable of the EXIT statement.
52	invalid IMAGE Invalid field in an ENTER or OUTPUT image.
68	wrong file type BCRD used on a file of a type other than BASIC or TEXT.
82	string expected ENTER image and variable type do not match (image is a string).
85	expr too big Reported on key entry if KEYBOARD IS has no room left for entering a key.
88	bad statement An unrecognized mnemonic is used in a SEND statement.
89	bad parameter An I/O ROM statement or function detects an invalid parameter (form or content).
91	missing parameter A parameter has been left out for a SEND mnemonic that requires one.
120	number expected ENTER image and variable type do not match (image is numeric).
121	bad digit A function that processes base dependent strings (HEX$, HAND$, etc.) encounters an invalid digit for the current base.
122	bad template Reported when TEMPLATE$ is given a template with no unprotected field.

Number	Message and Condition
125	`data not found` A file manipulation function cannot find the data requested.
126	`type mismatch` `CALL` and `SUB` parameters do not match in type.
127	`bad param value` `CALL` value does not match `SUB` parameter type.
128	`invalid subname` `SUB` name does not match filename.
129	`bad param type` `CALL` parameter is not of valid type. Numbers must be `REAL` (`INTEGER` and `SHORT` are not allowed).

Keyword Index

Keyword	Page	Description
DELAY?	99	Return current delay setting.
DELETE#	116	Delete data items.
DEVADDR	45,62	Return HP-IL address of specified device.
DEVAID$	48,63	Return Accessory ID as a string.
DEVID$	48,64	Return Device ID as a string.
DEVNAME$	45,65	Return device code of specified device.
DIM	111	Dimension string arrays.
DISPLAY$	89	List current display devices.
DO ERROR	99	Cause given error.
DTB$	100	Convert decimal number to binary string.
DTH$	100	Convert decimal number to hexadecimal string.
DTO$	100	Convert decimal number to octal string.
ENABLE SRQ	89	Reenable ON SRQ after an ON SRQ execution.
ENDLINE$	90	Return current endline string.
ENTER	14,22,66	Input bytes from specified device; build number or string; place result in BASIC variable.
ENTIO$	32,68	Send HP-IL commands to specified devices; return data as a character string.
ESC$	100	Return string of escape-character sequences.
ESC-I/R ON OFF	90	Turn modified I/R on or off.
EXIT	100	Leave a FOR-NEXT loop early.
FILE$	117	Show name of specified ASSIGN# file.
FILL$	101	Fill a string.
FIND	101	Find specified occurrence of substring in string, with wild card.
FLAG$	101	Set specifed bit to specified value in given string.
FLAG?	102	Test specified bit in string.
FOR	102	FOR allowed after a THEN or ELSE.
GOSUBX	102	GOSUB to a variable as a line number.
GOTOX	102	GOTO to a variable as a line number.
HAND$	103	AND of two hexadecimal strings.
HEX$	103	Convert ASCII string to hexadecimal.
HOR$	103	OR two hexadecimal strings.
HROT$	103	Rotate a hexadecimal string left or right by bit count.
HSHF$	104	Shift a hexadecimal string left or right by bit count.
HTD	104	Convert hexadecimal string to decimal number.
HXOR$	104	EXOR two hexadecimal strings.
IMAGE	17,69	Specify format of ENTER or OUTPUT statement.

Keyword	Page	Description
INDEX#	117	Return current data pointer position in file.
INSERT#	117	Insert an item at the current data pointer.
INSTALL	105	Load private file from tape (created by MCOPY).
IOSIZE	28,71	Set enter buffer size.
IOSIZE?	90	Return current IOSIZE setting.
ITEM#	117	Return pointer position in current line.
KEYBOARD$	90	Return device code of current keyboard device.
KEYBOARD IS	90	Assign device for keyboard entry.
LASTLN?	117	Return line number of last line in specified file.
LCD ON/OFF	105	Turn LCD on/off.
LEFT$	105	Return left portion of string.
LINE#	118	Return current line number in specified ASSIGN# file.
LINELEN#	118	Return the number of items in a line.
LISTIO$	92	List HP-IL device codes in string.
LOCAL	46,72	Return HP-IL devices to local control.
LOCAL LOCKOUT	47,73	Lock out local control of HP-IL devices.
LTRIM$	105	Left trim a string.
LWRC$	105	Convert string to lowercase.
MAP$	106	Map "from" characters into "to" characters in target string.
MARGIN?	106	Return current right margin setting.
MCOPY	106	Duplicate tape onto multiple tapes.
MID$	106	Return middle portion of string.
NEXT	107	NEXT allowed after a THEN or ELSE.
NSCR$	107	Remove underscoring.
OFF SRQ	92	Turn off HP-IL service request response.
ON SRQ	92	Respond to HP-IL SRQ messages.
OTD	107	Convert octal string to decimal number.
OUTPUT	13,17,74	Output bytes (string or numeric) to specified devices.
PPOLL	50,76	Return result of parallel poll.
PRINT# ... USING	118	PRINT# to a TEXT file with USING format.
PRINTER$	92	List current printer devices.
PWIDTH?	107	Return current PWIDTH setting.
REASSIGN	92	Change device code of an HP-IL device.
REMOTE	46,77	Set specified devices to remote mode.
REPL$	107	Replace substring in target string with another.
REPLACE#	118	Replace a data item in a file.

HEWLETT PACKARD

Portable Computer Division
1000 N.E. Circle Blvd., Corvallis, OR 97330, U.S.A.

European Headquarters
150, Route Du Nant-D'Avril
P.O. Box, CH-1217 Meyrin 2
Geneva - Switzerland

HP-United Kingdom
(Pinewood)
GB-Nine Mile Ride, Wokingham
Berkshire RG11 3LL

00075-90243 English

Printed in U.S.A. 1/84