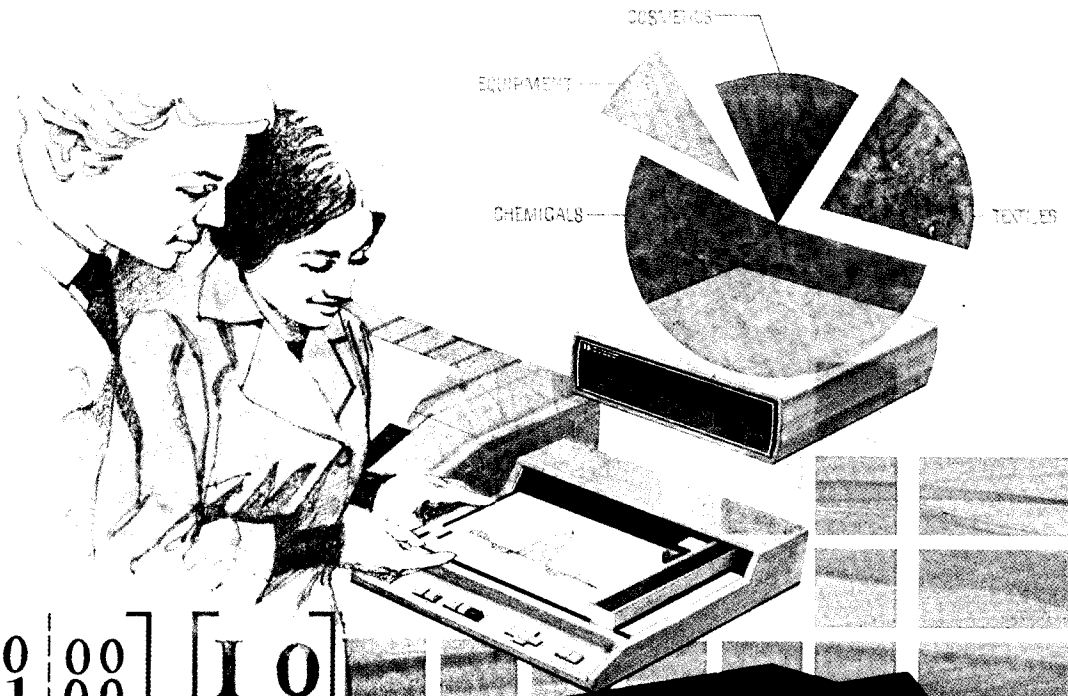
HEWLETT-PACKARD

I/O ROM

OWNER'S MANUAL

SERIES 80



I/O ROM
Owner's Manual

Series 80

January 1983

00087-90121

Printed in U.S.A.

Contents

Section 1

An Introduction to I/O

Introduction

The power of your HP Series 80 Personal Computer is greatly extended by its ability to monitor and control external events. I/O, which stands for Input/Output, refers to this process of interaction between the external, or *peripheral,* device and the computer. There are several interfaces available so that practically any device designed for computer control may be attached and thus monitored and controlled.

When the computer is monitoring an event it is said to be the *listener* and it receives a message from the talker or *source.* This data message is referred to as *Input.* When the computer is controlling, on the other hand, it is the *source* and the peripheral device that it is talking to is called the *destination.* In this case, the message is considered *Output.*

The process of transferring data can become rather involved, depending on the type of peripheral and the application. Fortunately, a thorough understanding of interfacing techniques is not necessary to be able to utilize the I/O ROM with your computer; hence, for maximum effectiveness, this manual is arranged into three parts:

1. getting started with I/O—how to make use of simple I/O features quickly and easily (sections 1-3);
2. undertaking more complex I/O operations—understanding the full capability of I/O ROM interface commands (sections 4-10);
3. appendices—references and useful information in condensed form.

Sections 1 through 3 are adequate to explain operation of the printer interface. Also, a number of examples are presented in the HP 82949A Printer Interface Owner's Manual. If you intend to use one of the additional interfaces it is recommended that you become familiar with the additional flexibility of the I/O ROM as presented in sections 4 through 10. The owner's manuals for the Serial, HP-IB, GPIO, BCD, and other interfaces will then provide detailed instructions for each particular interface.

This manual covers both the I/O ROM for the HP-85 and HP-83 (part number 00085-15003) and the I/O ROM for use with the HP-87 and HP-86 (part number 00087-15003).

Installing the I/O ROM

Use of the I/O ROM requires an HP 82936A ROM Drawer. The ROM Drawer is a plug-in module that contains six rectangular slots for individual plug-in ROMs. Any HP Series 80 ROM will fit in any of the six positions in the ROM Drawer.

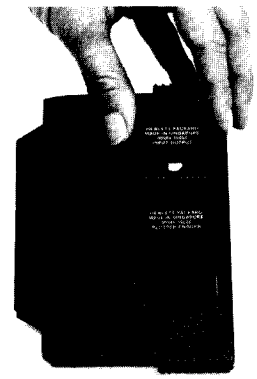
The HP Series 80 Personal Computer has four ports located in the back. These ports are used to hold extra memory, option ROMs, and interfaces. The I/O ROM should be installed in a ROM Drawer, which is then installed in one of the computer ports.

CAUTION

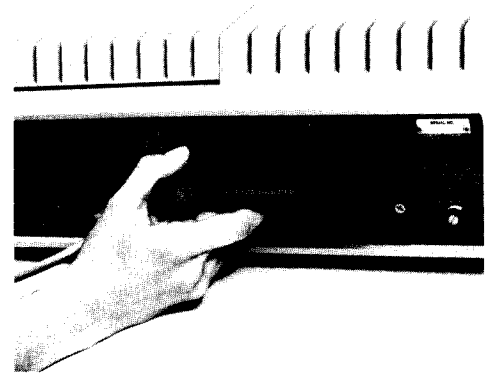
Do not remove or install the ROM Drawer or any plug-in module while the computer power is switched on. Failure to switch off the computer power may cause damage to the module or the computer.

The installation procedure is as follows:

1. If the ROM Drawer is already plugged into the computer, TURN OFF power to the computer and remove the ROM Drawer.
2. Remove the plastic cover from an empty ROM socket by inserting the eraser end of a pencil into the circular hole on the underside of the cover.
3. Align the ROM so that its chamfered end matches the chamfered end of the socket. Press the ROM lightly into the socket until it is even with the top of the ROM Drawer.



4. With the ROM labels facing up, press the ROM Drawer firmly into one of the ports in the back of the computer. The port and module are keyed so that the ROM Drawer cannot be installed upside down.



Removing the I/O ROM

The procedure to remove a ROM is as follows:

1. TURN OFF power to the computer.
2. Remove the proper ROM drawer.
3. Turn the drawer over and remove the ROM by using a pen, pencil, or small screwdriver to push gently through the hole underneath the ROM.

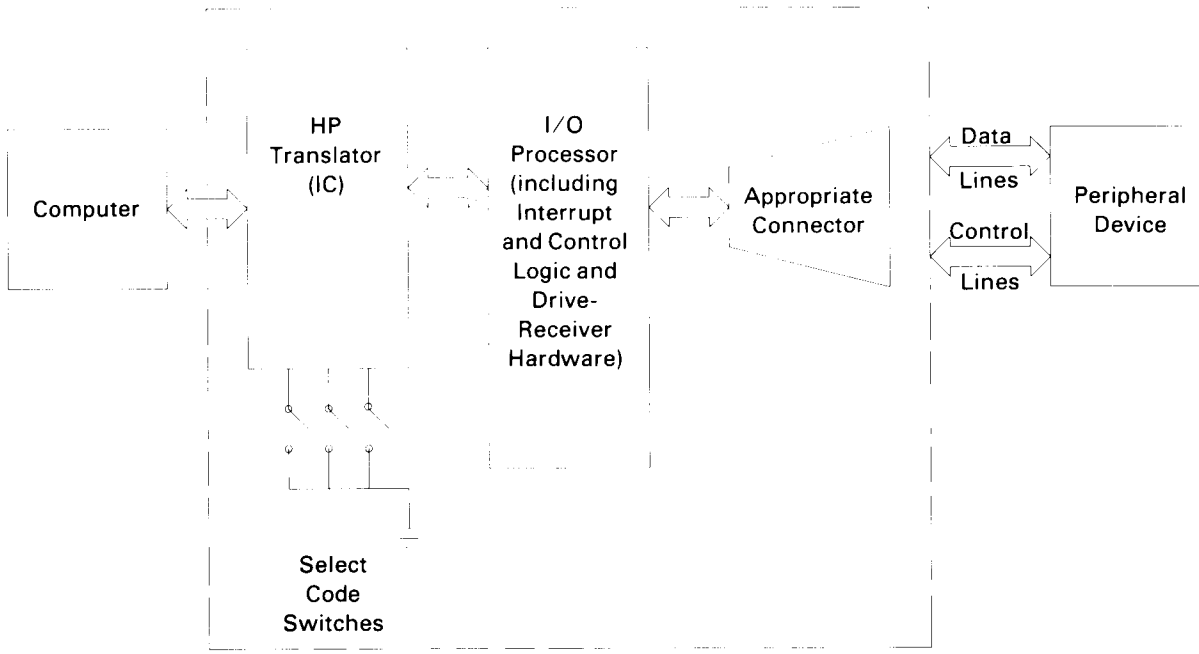
The Job of an Interface

An interface is the hardware link that is needed to allow efficient communication with peripheral devices. The job of an interface is to provide compatibility in four major areas. These are:

- Mechanical Compatibility
- Electrical Compatibility
- Data Compatibility
- Timing Compatibility

The following diagram shows the basic hardware components of an interface in its position between the computer and the peripheral device.

Interface Functional Diagram



Mechanical and Electrical Compatibility

Mechanical compatibility simply means that the plugs and connectors must fit together. The HP 829XX Series plug-in modules are designed to be mechanically compatible with your computer. Certain interfaces, like HP-IB, are always mechanically compatible with their peripheral devices. Other interfaces, like the 16-bit parallel interface (GPIO), are supplied without peripheral connectors. In these cases, it becomes your responsibility to install a mechanically compatible connector. If you need to do this, study the owner's manual supplied with your interface. Electrical compatibility means that the interface must change the voltage and current levels used by the computer to those used by the peripheral device. The *Translator IC* used in each interface and the *Drivers* and *Receivers* insure that all HP interfaces are compatible with the computer, and usually compatible with the peripherals. If you have questions about electrical compatibility with your peripheral device, study the interface owner's manual and the peripheral device documentation.

Data Compatibility

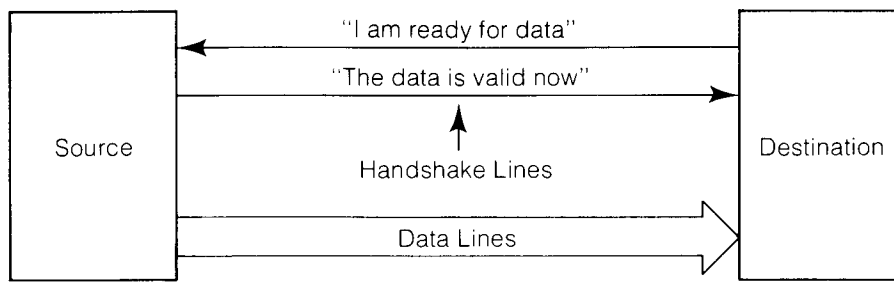
Mechanical and electrical compatibility alone do not guarantee that the computer and peripheral device will be able to communicate. Another requirement is that both devices must understand the data being sent by the other. Just as two humans who do not speak the same language need a translator, messages between the computer and the peripheral device may require some form of translation. The computer, with its versatile programming capability, usually performs this function. However, this job is sometimes given to the interface. The BCD interface is one example of giving the translation process to the interface. To handle those cases where the computer performs the data translation or conversion, the I/O ROM provides a wide variety of special functions and conversion capabilities. These capabilities are covered in subsequent sections of this manual.

Timing Compatibility

Computers and their peripheral devices have such a wide range of operating speeds that an orderly mechanism is required for successful transfer of data. This timing mechanism is referred to as *handshake*. Although there are different varieties of handshake, the basic sequence can be summarized as follows:

1. The receiver signals that it is ready for an item of data, then waits for a signal from the sender that the data is available.
2. The sender outputs an item of data and signals the receiver when the data is available.
3. When this "data available" signal is recognized, the receiver inputs the data and signals that it is busy with this input operation.
4. The sender waits until the receiver is ready before it makes a new item of data available. When the receiver is ready, this process repeats.

The following simplified diagram further illustrates the general concept of handshaking.



The *I/O Processor* is a component of each interface, which is itself a microcomputer. It is supplied with a small set of programs limited to data conversion and handshake management routines. Through the use of various commands available with the I/O ROM, the I/O Processor can be reconfigured to perform several different types of handshake operations.

Choosing the Source or Destination

To send someone a message through the mail, you must specify their address before the post office will even attempt delivery. So it is when you want to communicate with peripheral devices. The device with which you want to communicate must be specified within your program. This selection process is called *addressing*. The HP Series 80 Personal Computer addresses its peripheral devices through the use of a *device selector* in the I/O statements. Choosing the proper device selector depends upon the interface used and the way it is used. The following two discussions detail the two types of device selectors.

Using Interface Select Codes

If you are not using an HP-IB interface, and you have only one device connected to the interface, the device selector can be simply the *select code* of the interface. An interface select code is analogous to a house number in a mailing address. It is a number between 3 and 10 (inclusive) that identifies the interface. Each type of interface is set to a different select code at the factory. The following table summarizes these factory settings.

HP Part Number	Name	Select Code Setting
82937A	HP-IB	7
82938A	HP-IL	9
82939A	Serial	10
82940A	GPIO (Parallel)	4
82941A	BCD	3
82949A	Printer	8
82950A	Modem	10

Note: For electrical reasons, you must never plug in two interfaces that are set to the same select code.

Serious electrical conflicts can result if the interfaces and select codes do not correspond uniquely. In other words, if there are two interfaces present with the same select code, neither one of them will work. As you can see from the preceding table, the factory settings prevent this from happening unless you are using two interfaces of the same type. If you need to use two interfaces that came with the same factory setting, you must change the select code on one of them. The procedure for changing a select code is covered in the owner's manual for your interface. Follow its instructions carefully.

Using a Primary Address

If you are using an HP-IB interface, or if you have more than one device connected to an interface, the device selector is a 3-digit or 4-digit number formed from the interface select code and a *primary address*. This method of addressing is like mailing a letter to someone in an apartment building. Giving the street address will get the letter to the right building, but you still need to specify an apartment number to get the letter to the final destination. When a primary address is used, it is analogous to the apartment number. It identifies a specific device to be selected from a group of devices serviced by one interface. Some examples:

- A device selector of **721** specifies device **21** on interface **7**.
- A device selector of **301** specifies device **1** on interface **3**.
- A device selector of **1002** specifies device **2** on interface **10**.

Printing to Peripheral Devices

One of the simplest ways to direct the computer's output to a peripheral device is the PRINTER IS statement. The I/O ROM provides the capability of printing to external devices by using statements such as PRINTER IS 4 or PRINTER IS 720. Any valid device selector can be used with the PRINTER IS statement. The same holds true for the CRT IS statement.

The PRINTER IS device is the destination for the output from all PRINT and PLIST statements. The CRT IS device is the destination for the output from all DISP, LIST, and CAT statements, as well as all "Error" and "Warning" messages. When programs are listed to a peripheral device, each program line is output as a single string. For the HP-85 this means that there is no indenting or 32-character wraparound as occurs when listings are done on the internal printer or CRT. PRINT and DISP both print out with the default line length set by the computer. To extend the line length or utilize more elaborate formatting you will need to explore the next two chapters. If, on the other hand, your only I/O requirement is to direct program listings or the output from PRINT statements to an external device, you need not read any further. Simply connect the desired interface and include the appropriate PRINTER IS statement in your program.

Section 2

Simple I/O Operations

Introduction

Section 1 talked about performing output operations with PRINTER IS and PRINT statements. Although this simple technique is very handy, it falls short of the mark in many circumstances. The most obvious shortcoming is that there is no corresponding KEYBOARD IS statement to allow input from external devices. Even when output is the only desired operation, it can be very inconvenient re-specifying the PRINTER IS device all the time when a program communicates with multiple peripheral devices.

The principal tools for using interfaces to move data in 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 code values is included in appendix D.

When special formatting is desired or when binary code is handled directly without using ASCII representation, 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 can be used anywhere that a simple PRINT statement is proper. The OUTPUT statement contains the device selector(s) of the destination device(s) and a list of the items to be output. The primary difference between OUTPUT and PRINT is that PRINT statements do not contain a device selector. Here are some examples of properly written OUTPUT statements:

```
OUTPUT 1 ; "Hello"
OUTPUT 3 ; X
OUTPUT S1 ; A$,B$
OUTPUT 703,725 ; X;Y;Z
OUTPUT 1000 ; A(1);B(3),N$[2,7]
```

Notice that a semicolon is used to separate the device selector from the output list, and commas or semicolons can be used to separate items within the output list. Items in the output list may be numeric variables, numeric constants, string variables, or string constants. A Carriage Return/Line Feed (End of Line sequence) is output after the last item in the output list.

The difference between using a comma and a semicolon to separate items in the output list is the spacing, or *field* of the items. The simple OUTPUT statement uses the same field as the PRINT statement. The semicolon calls for a compact field, while the comma produces free field. These fields are summarized in the following table.

	Numeric Data	String Data
Compact Field ; (*semicolon*)	Digits of the number are output, preceded by a space (if plus) or a minus sign (if minus), and followed by one space.	Characters of the string are output with no leading or trailing spaces.
Free Field , (*comma*)	Digits of the number (with leading space or minus sign) are output left-justified in a field of 11, 21, or 32 characters. Trailing spaces are output as necessary to fill the unused portion of the field.	Characters of the string are output with no leading spaces and no more than 20 trailing spaces.

The actual field width is determined by the same process used when items are output with the PRINT statement. Therefore on the HP-85, the computer pretends that it is displaying items on the CRT and sets a field width that would cause items to start in column 1 or 22 of the 32-column display. On the HP-87 the items will start in columns 1, 22, 43, or 64 of the 80-column field. If this is an undesirable format, you may need to separate items in the output list with semicolons or use formatted output as explained in the following section.

Using Simple ENTER Statements

A simple ENTER statement can be used anywhere that an INPUT statement is proper. The ENTER statement contains the device selector of the source device and a list of items to be entered. Remember that INPUT statements always use the keyboard as the source and contain no device selector, while ENTER statements always use a peripheral device as the source and contain the device selector of that device. Here are some examples of properly formed ENTER statements:

```
ENTER 3 ; X
ENTER S1 ; A$,B$,C$
ENER 703 ; X,Y,Z
ENTER 1000 ; A(1),B(3),N$
```

Notice that a semicolon is used to separate the device selector from the enter list and 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 3 ; X,Y
```

Now assume that when this statement is executed, the following character sequence is received through the interface at select code 3:

T	U	E	S	D	A	Y		D	E	C		1	1	,		1	9	7	9	cr	lf

The computer ignores all leading 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 non-numeric character signals the end of that number. Therefore, the comma after the 11 causes the computer to place the value eleven into X and start looking for the next value. The space in front of "1979" is ignored and the computer reads the "1979". The carriage-return character causes the computer to place the value nineteen hundred seventy nine into variable Y. Finally, the computer keeps reading until it finds a line feed character. This terminates the ENTER statement, so the computer goes on to the next program line with X=11 and Y=1979.

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

1. Ignores leading non-numeric characters.
2. Ignores all spaces—leading, trailing, or imbedded.
3. Uses numeric characters to build a numeric value.
4. Terminates the building of a value when a trailing non-numeric character is encountered.
5. Inputs characters until a line feed character is encountered.

The discussion so far has referred to numeric and non-numeric characters without being specific. The digits 0 thru 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 12.5×10^{-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. A line feed character is received or,
3. A carriage return/line feed sequence is received.

Assume that the computer is executing the statement:

```
ENTER 4 ; A$,B$,C$
```

The following character sequence is received:

H	E	L	L	O	lf	cr	lf	T	H	E	R	E	cr	lf

The letters "HELLO" are placed into A$ when the first line-feed is encountered. Note that the line-feed itself is not placed into A$; it acts only as a terminator for the entry into A$. Then the entry into B$ begins. However, a carriage return/line feed sequence is read immediately. This terminates the entry into B$. Since neither the carriage return nor the line feed is placed into B$, B$ becomes the null string. Next, the entry into C$ begins. The characters "THERE" are placed into C$, terminated by the carriage return/line feed following those characters. With the enter list now satisfied and a line feed detected at the end of the data, the computer will go on to the next program line.

Note that carriage return characters are only ignored when they are immediately followed by a line feed character. If a carriage return is received and not followed by a line feed, the carriage return is placed 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$[3]
ENTER 4 ; X$
```

The following characters are sent to the computer:

B	O	Y	C	O	T	T	cr	lf

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

Section 3

Formatted I/O Operations

Introduction

Although free-field format works well for some I/O situations, there are times when more control over format is necessary. Perhaps the data is some binary pattern which has nothing to do with ASCII, or a line-feed terminator is not wanted or expected, or a column of numbers with the decimal points in line is desired, or numbers with only two exponent digits instead of three are required. There is a wide variety of 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 computer uses an output image when some form of the OUTPUT USING statement is encountered. There are three forms of this statement:

1. 10 IMAGE <*output image*>
   20 OUTPUT *ds* USING 10 ; <*output list*>
2. OUTPUT *ds* USING <*output image*> ; <*output list*>
3. 10 FMAT: IMAGE <*output image*>
   20 OUTPUT *ds* USING FMAT; <*output list*>

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

```
10 TOT: IMAGE "Total =",ZZ.D
20 IMAGE 5A,2X,17A
:
60 OUTPUT 4 USING TOT ; C1,C2,C3
70 OUTPUT 701 USING 20 ; A$,B$
80 OUTPUT 9 USING "#,B" ; X
90 OUTPUT S3 USING "MDDD.DD" ; T(1),T(2)
100 OUTPUT 710,711 USING I$ ; N$,A
```

In the general forms, the *ds* stands for "device selector". Device selectors are explained in Section 1. The symbol <*output image*> represents a proper list of image specifiers. The image specifier list may be a literal enclosed in quotes or the name of a string variable which contains the specifier list. The specifiers within the list must be separated by commas. The list of items to be output is shown by <*output list*>. It does not matter whether you use commas or semicolons to separate items within the list. All spacing is controlled by the image specifiers, so a semicolon has the same effect as a comma.

Numeric Images

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

Digit Characters

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	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	Same as "D", except leading zeros are output.
*	Same as "D", except leading zeros are replaced by asterisks.
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.
e	Same as "E", except only two exponent digits are output.
K	Causes the number to be output in compact format. No leading or trailing spaces are output.

Sign Character

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	Causes the output of a leading plus or minus sign to indicate the sign of the number.
M	Causes the output of a leading space for a positive number or a minus sign for a negative number.

Punctuation Characters

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	Causes a European radix point to be output (a comma).
C	Usually placed between groups of three digits. Causes a comma to be output to separate the groups of digits (American convention).
P	Same as "C", except a period is used to separate the groups of digits (European convention).

The following examples show some of the ways of combining these specifiers and the resulting output when numbers are sent to a typical printer. Additional examples may be found in the "Printer and Display Formatting" section of the Series 80 owner's manuals and programming guides.

Example Statement		Printed Output
`OUTPUT 701 USING "ZZZZ.DD" ; 30.336`		`0030.34`
`OUTPUT 701 USING "4Z.2D" ; 30.336`		`0030.34`
`OUTPUT 701 USING "4Z.2D" ; -30.336`		`-030.34`
`OUTPUT 701 USING "3DC3DC3D" ; 1E6`		`1,000,000`
`OUTPUT 701 USING "3DC3DC3D" ; 1.2345E4`		`12,345`
`OUTPUT 701 USING "3DC3DC3D" ; 1.2E9`	(Overflow Error)	`, ,`
`OUTPUT 701 USING "S2.DDD" ; .5`		`+0.500`
`OUTPUT 701 USING "MZ.DDD" ; .5`		`0.500`
`OUTPUT 701 USING "MD.DDD" ; .5`		`.500`
`OUTPUT 701 USING "Z.DDE" ; .00456`		`4.56E-003`
`OUTPUT 701 USING "Z.DDe" ; .00456`		`4.56E-03`
`OUTPUT 701 USING "Z.DDe" ; -.00456`		`-.45E-02`
`OUTPUT 701 USING "MZ.DDe" ; -.00456`		`-4.56E-03`

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)"`, means something different. The image `"3D"` means "output one numeric quantity in a three-digit field". The image `"3(D)"` means "output three numeric quantities, putting each one in a 1-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, exactly what is output is difficult to predict and will probably bear little or no resemblance to the number that caused the overflow.

String Images

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 `PRINT` image specifiers that may already be familiar to you.

Image Specifier	Meaning
`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"`	A *literal* is a string constant formed by placing text in quotes, or by using the `CHR$` function, 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 quote marks themselves are not output. Literal images are commonly used for labeling other output. Literal images cannot be placed directly into `OUTPUT` statements. An `IMAGE` statement must be used if literal images are desired.
`X`	Causes the output of one space.
`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 can be found in the "Printer and Display Formatting" section of each of the HP Series 80 owner's manuals.

Example Statements **Printed Output**

```
OUTPUT 701 USING "5A,A" ; "X","Y"                    X    Y
OUTPUT 701 USING "K,3X,K" ; "UNCLE","SAM"          UNCLE   SAM
OUTPUT 701 USING "K,3X,K" ; 98.6,99.9              98.6   99.9

10 IMAGE "TOTAL = ",3D,X,K
20 T=125 @ A$="CARS"
30 OUTPUT 701 USING 10 ; T,A$                  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 can be combined in the same image statement. If literal (string constant) images are desired, they must be placed in an IMAGE statement.

Binary Images

These image specifiers are not available without the I/O ROM, so they may not already be familiar to you. These images are used to cause information to be output as one or two binary bytes, rather than as a character representation. Sections 10 and 11 of this manual explain the details of binary (base 2) representation. If you are unfamiliar with binary numbers, it is suggested that you read section 9 before trying to use the binary image specifiers.

The items to be output using these images must be numbers in the proper range. If a value to be output is not an integer, it will be rounded to the nearest integer before being sent as a binary value.

Image Specifier	Meaning
B	Outputs a value as a single 8-bit byte. The value must be in the range of 0 thru 255. If the value to be output is out of range, the value modulo (256) is output.
W	Outputs a value as two 8-bit bytes comprising a 16-bit word. The most significant byte of the word is output first, followed by the least significant byte. The value to be output must be in the range of −32,768 to +32,767. Negative numbers are output in 16-bit 2's complement form. If the value to be output is out of range and positive, 32,767 is output. If the value is out of range and negative, −32,768 is output.

Example Statement **Interface Output, Bit Pattern**

```
OUTPUT 7 USING "B" ; 127                          01111111
OUTPUT 7 USING "B" ; 3                            00000011
OUTPUT 7 USING "W" ; 3                   00000000 00000011
OUTPUT 7 USING "W" ; -1                  11111111 11111111
```

Note that specifying a binary image does not automatically suppress the end-of-line sequence after the last byte is output. Therefore, in the examples just given, the bit pattern shown is output followed by a carriage return/line feed.

BCD interface users should note that "B" and "W" specifiers do not work with the HP 82941A BCD Interface. See the *BCD Interface Owner's Manual* for specific formatting examples.

End-of-Line Sequence Images

These image specifiers control the output of end-of-line sequences. An end-of-line sequence is one or more characters that is normally output after the last item in an output list, and/or a signal on an interface wire concurrent with the last byte output. Exactly which sequence or signal is used depends upon the programming of the interface responsible for the output. See your interface owner's manual for more details. If your program does not change the end-of-line sequence in the interface, the default is a 2-character sequence; a carriage return followed by a line feed. The following images do not alter the end-of-line sequence. They simply control whether or not it is output.

Image Specifier	Meaning
/	Causes the output of an end-of-line sequence. Often used for skipping lines in a printout.
#	Suppresses the output of the final end-of-line sequence. This specifier is frequently used with binary image specifiers to prevent the destination device from interpreting the end-of-line characters as binary data.

The "/" may be placed anywhere in the image list and may have a number before it to indicate how many EOL (end-of-line) sequences are desired. The "#" must be the first item in an image list and can only be specified once. Note also that the "#" only suppresses the EOL sequence that would ordinarily occur after the last item in the output list. It does not suppress any imbedded EOL sequences caused by the "/" specifier.

A typical use of the "#" image is to output one byte, and only one byte. The following statement does this:

```
OUTPUT 6 USING "#,B" ; X
```

This statement outputs the binary representation of X with no carriage return, line feed, or any other potentially unwanted bit patterns.

A typical use of the "/" image is shown by the statement:

```
OUTPUT 701 USING "K,4/,K" ; A$,B$
```

If the destination is a printer, A$ is printed, followed by three blank lines, then B$ is printed. If A$="HI" and B$="JOE", the character sequence output looks like this:

H	I	cr	lf	cr	lf	cr	lf	cr	lf	J	O	E	cr	lf

The "#" image specifier is unique to the I/O ROM. Its power is shown in the following example where repetitions of the PRINT statement cause printout on the same line.

```
10 FOR I=1 TO 3
20 PRINT USING "#,6A" ; "DOOBY"
30 NEXT I
40 PRINT USING "2A" ; "DO"
50 END
DOOBY DOOBY DOOBY DO
```

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 condition(s) constitutes the end point of an entry to a variable and the end point of the ENTER statement itself.

This discussion deals with data formatting images first, then presents the terminator images. The HP Series 80 Personal Computer uses an ENTER image when some form of the ENTER USING statement is encountered. There are three forms of this statement:

1. 10 IMAGE *<enter image>*
   20 ENTER *ds* USING 10 ; *<enter list>*
2. ENTER *ds* USING *<enter image>* ; *<enter list>*
3. 10 LBL: IMAGE *<enter image>**
   20 ENTER *ds* USING LBL ; *<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 FMT: IMAGE 5D,2X,3De
:
60 ENTER 4 USING 10 ; A$,B$,X
70 ENTER 711 USING FMT ; I,J
80 ENTER 9 USING "#,B" ; A(1),A(2)
90 ENTER 52 USING "%,8A,/,K" ; Q$,R$
100 ENTER 712 USING I$ ; N$,A
```

The general forms use the same type symbols which were used to represent the OUTPUT statement. These are *ds* for "device selector", *<enter image>* for the list of image specifiers, and *<enter list>* for the list of variables to be entered. As with simple ENTER statements, the enter list must contain either string or numeric variables.

Data Images

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

1. Use characters to build a numeric variable.
2. Place characters into a string variable.
3. Input bytes as binary values.
4. Skip over a number of characters.

Numeric Image Specifiers

These specifiers are used to control the input of numeric characters, including digits, sign, exponent, and punctuation.

---

*Note that labels are not allowed on the HP-85 or HP-83.

Image Specifier	Meaning
D Z * . S M	These specifiers all do the same thing. They tell the computer to accept one character to be used in building a numeric quantity. 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.
E	Tells the computer to accept five characters to be used for building a number. The five characters do not have to be exponent information, but they can be.
e	Same as "E", except the computer accepts four characters to be used in building a number.
C	This specifier also tells the computer to accept one character to be used in building a numeric quantity. 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 end the entry of a numeric quantity.
K	Tells the computer to enter a string or numeric variable using free field format (explained in Section 2).
R P	These specifiers are used with the OUTPUT statement to provide a European radix point and digit separator. However, these images are NOT permitted for an ENTER statement. If you need to enter numbers in European format, you can use the CONVERT statement (covered later in this section) to change the number into American format.

String Image Specifiers

These specifiers are used to enter characters into string variables.

Image Specifier	Meaning
A	Tells the computer to enter one string character.
K	Tells the computer to enter a string or numeric 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	cr	lf

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

```
ENTER 720 USING "4D,5A" ; X,Y$
ENTER 720 USING "Z.DD,5A" ; X,Y$
ENTER 720 USING "e,K" ; X,Y$
```

Notice that any numeric image that accepts four characters will properly enter the "1234". String data can be entered with an "**n**A" image if **n** (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	cr	lf

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

```
ENTER 720 USING "C4D,K" ; X,Y$
ENTER 720 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.

Binary Image Specifiers

These specifiers are used to enter data that is received in binary format.

Image Specifier	Meaning
B	Tells the computer to enter one byte of binary data and enter its equivalent decimal value into a numeric variable.
W	Tells the computer to enter two bytes of binary data to be used in building a 16-bit, 2's complement binary word. The equivalent decimal value of the resulting word is entered into a numeric variable. The first byte entered is used as the most significant byte of the word.

Skipping Unwanted Characters

These specifiers can be used with incoming numeric or string data to skip over any characters not wanted for the input.

Image Specifier	Meaning
X	Tells the computer to skip over one character.
/	Tells the computer to skip to a line feed. Thus, after the variable has been satisfied, the computer "throws away" incoming characters until a line feed is received.

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 720 USING "8X,K" ; A$
```

The "/" specifier is used to demand a line-feed field terminator 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	lf	B	Y	E	cr	lf

Using the statement:

```
ENTER 720 USING "3D,K" ; X,Y$
```

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

```
ENTER 720 USING "3D,/,K" ; X,Y$
```

is used, then X gets the value 123 and Y$ becomes "BYE". The "/" specifier caused the computer to skip all characters after X was entered until it saw the line feed. Then the entry into Y$ began with the first character after the line feed. Without the "/" specifier, the entry into Y$ began as soon as the "3D" field was exhausted.

Eliminating the Line-Feed Requirement

The ENTER statement must "see" a line-feed character at the end of the incoming data before the program can go on to the next statement. If there is no line-feed character at the end of the data, the computer will be "hung up" waiting for one. If your incoming data does not have a line feed at the end, you can get the ENTER statement working properly by using an image specifier.

Image Specifier	Meaning
#	Eliminates the requirement for a line feed to terminate the ENTER statement. When this specifier is present, the ENTER statement terminates as soon as the last variable in the statement has been satisfied.

When the "#" specifier is used for this purpose, it must be listed as the first specifier in the image list. For example:

```
ENTER 3 USING "#,K" ; A$
ENTER 720 USING "#,4D,6D" ; X,Y
```

The first example statement shows an entry into a string variable using free-field format with the line-feed requirement removed. This statement terminates when the string is full. The second example shows a formatted entry into numeric variables with the line-feed requirement removed. This statement terminates after inputting ten characters.

Advanced Use of Terminator Images

The ENTER image specifiers discussed in the preceding sections are sufficient to handle the great majority of requirements. However, there are some special situations that demand an even greater amount of flexibility. Most of these special cases involve the EOI (End or Identify) line on the HP-IB. The following discussion is probably of no concern to most programmers. If you are one of those who must consider the EOI line, or if you have an unusual problem with line feeds, then read carefully. This is the most complex part of this section.

Field and Statement Terminators

The purpose of an ENTER statement is to read a *record.* To the programmer, a record is a logical grouping of data items. To the computer, a record is an incoming stream of data ended with a record terminator. Since the ENTER statement is ended when the record terminator is read, this manual refers to the record terminator as a statement terminator. If there is a requirement for a statement terminator in effect, the ENTER statement does not end until that terminator is received. (The action is slightly different when using buffers. These are covered in section 5.) If no terminator image is specified, the default statement terminator is a line feed character. To allow a carriage return/line feed sequence as a statement terminator, the I/O ROM ignores a carriage return if it is immediately followed by a line feed.

An incoming record often contains multiple *fields.* A field is the group of characters used to determine the input to a variable in the ENTER list. For example, an ENTER statement used to input a list of names and ages might look like this:

```
ENTER 720 ; N$,A
```

This statement reads a record containing a name and an age. This record has two fields. The first is a string field (the name), and the second is a numeric field (the age). A properly specified ENTER statement places the string field in N$ and the numeric field in A.

It is not generally necessary to specify any terminator images to get the ENTER statement to perform properly. The system has built-in field terminators and a default statement terminator which are sufficient for most common applicants. These normal terminators are:

- A string field ends when the string is full (check your DIM statements), the character count from an image field is exhausted, or a line feed is received.
- A numeric field ends when any non-numeric character (except a space) is encountered or the character count from an image field is exhausted.
- The ENTER statement ends upon receipt of a line feed character or a carriage return/line feed sequence. This can be the same line feed that satisfied the last field in the ENTER list.

Given these normal terminating conditions, the ENTER statement mentioned previously properly separates the name field and the age field in two cases. One case is when there is a line feed separating the string field from the numeric field. The other case is when the string field is always of fixed length and N$ is dimensioned to that length.

Terminator Images

If the normal terminating conditions are not ideally matched to your application, the use of terminator images can help solve the problem. The following image specifiers apply to both field and statement terminating conditions. Field terminators are the conditions that end the entry of data into a variable. Statement terminators are the conditions that end the ENTER statement after the last variable is satisfied.

Image Specifier	As a Field Terminator	As a Statement Terminator
#	Eliminates line feed as a terminating condition during free-field string entry. Line feeds entered are placed into the string.	Suppresses the requirement for a line-feed terminator. Statement ends when last field is satisfied.
%	Allows EOI as an additional terminating condition.	Allows EOI or line-feed as terminating conditions.
#% (or %#)	Allows EOI as an additional terminating condition, and also eliminates line feed as a terminator during free-field string entry.	Specifies that an EOI must be received to terminate the statement, and line feed is not a terminator.

Whether an image specifier controls statement terminator or field terminators depends upon where it is placed in the image. Consider the following example statement:

```
ENTER 720 USING "%,%K" ; X
```

When the terminator image is specified by itself as the first item in the image list (like the first %), it specifies the statement terminator. When the terminator image is combined with another specifier (like the %K), it specifies a field terminator. The "#", "%", and "#%" images all follow this convention.

Because the built-in field terminators are always in effect, these special terminator images only alter the system's action in a few cases. Let's look at each of these meaningful field terminating combinations individually.

Entering Line-Feeds Into a String

The image "#K" causes the computer to place all incoming characters (including line feeds) into a string until it is full. If there is a line feed forthcoming after the string is filled, this image is all that is necessary. If you wish the statement to end as soon as the string is filled (without waiting for a final line feed), the image "#,#K" should be used.

Using EOI to Terminate a String Entry

The image "%K" allows the computer to terminate a free-field string entry with the EOI signal. However, a device which uses EOI as its end-of-line indicator may not output any other end-of-line characters, like line feed. If this is the case, the proper image is "%,%K". This allows the EOI signal to also terminate the ENTER statement. If you wish to enter line feed characters into the string and also wish to terminate with EOI, the image "#%K" can be used. This may need to be expanded to "%,#%K" if no line feed is expected to terminate the statement. The further expansion "#%,#%K" not only *allows* EOI to terminate the ENTER statement, but *requires* it as the only method of terminating the statement. Fixed-field entries can be checked for an expected EOI. For example, the image "%7A" inputs seven characters into a string and expects to have an EOI signal with the seventh character. Keep in mind that there are many valid combinations of these image specifiers. The combinations shown here are only some of the more common ones.

Using EOI to Terminate a Numeric Entry

The image "%K" allows the computer to terminate a free field numeric entry with the EOI signal. As mentioned in the preceding paragraph, the image "%,%K" may be necessary if the EOI signal is to terminate the ENTER statement also. Fixed field entries can be checked for an expected EOI. For

example, the image "%7D" inputs seven characters to build a number and expects to have an EOI signal with the seventh character. Binary fields work in a similar manner. The image "%W" inputs two bytes to make a 16-bit integer and expects an EOI signal with the second byte.

There's Always an Exception

Not all terminator problems can be solved with terminator images. Consider again 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 simple comma. If the ages came first, this would not be a problem since the comma would end the entry to the numeric variable. But since the string data is entered first in this example, the task is a bit trickier. You might be able to use a CONVERT statement (explained at the end of this section) to change the comma into a line feed and terminate the string that way. If the application does not permit the blanket conversion of commas to line feeds, then the entire record would have to be input into a temporary string variable. Once the record is entered, the POS function and string subscripts could be used 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.

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 statement 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 exactly 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 a programmer determine the exact nature of incoming data when he or she can't get it into the computer to study it? If the only tools available were the string and numeric image specifiers, this might be a significant problem. Fortunately, there is a way to inspect a totally unknown character sequence. Any sequence of bytes, including potential terminators, can be entered with the "#,B" image. The values that are printed or displayed are the decimal equivalents of the binary value for each byte. Admittedly, this is not the most convenient form of data to work with. However, you can use an ASCII table or the CHR$ function to determine the exact character sequence which is being received. Then, knowing the exact nature of the incoming data, the job of choosing image specifiers will be much simpler. The following example program shows a typical use of this technique.

```
100 ! Program to inspect incoming data
110 S1=3 ! Interface select code
120 ! Estabish a terminating condition
130 SET TIMEOUT S1;3000
140 ON TIMEOUT S1 GOTO 230
150 !
```

```
160 I=1 ! Initialize counter
170 ! Input 1 byte; display analysis
180 ENTER 51 USING "#,B" ; X
190 DISP "BYTE";I;TAB(11);"VALUE =";X;TAB(24);"CHAR = ";
    CHR$(X)
200 I=I+1 ! Count bytes
210 GOTO 180
220 !
230 DISP "DATA INPUT HAS STOPPED"
240 RESET 51! Stop I/O operation
250 END
```

Converting I/O Data

The final type of formatting involves changing the data characters that are entered or output. An example cited earlier was incoming numbers in European format (with periods separating digit groups and a comma for the radix point). There is no image available to accept this type of data directly. The periods and commas need to be changed to other characters to give the computer what it wants. The tool for performing this kind of operation is the CONVERT statement. Its general form is:

CONVERT *<direction>* *<select code>* *<access method>* ; *<string>*

The parameters are defined as follows:

<direction> Indicates whether the conversion is to take place during an ENTER (choose IN) or an OUTPUT (choose OUT).

<select code> Indicates which interface will use the conversion. Note that the CONVERT operation applies to all devices on a particular interface. A device selector is not allowed. The parameter must be an interface select code, range 3 thru 10.

<access method> This specifies the method of accessing the conversion table. The conversion table is a string variable, and there are two access methods. If PAIRS is specified, the string is treated as a list of character points. The second character of a pair is substituted for the first character whenever the incoming or outgoing character matches a first character. This method is a good choice when only a few characters need to be converted.

If INDEX is specified, the string is treated as a sequential look-up table. The numeric value of each incoming or outgoing character is used as an index into that table. The first element in the string corresponds to the character with a value of 0. If the value of the character to be converted is too large for the number of characters in the string, no conversion is performed. This method is a good choice when a large number of characters need to be converted.

<string> This represents the actual conversion table. It must be a string variable. A literal (string constant) is not allowed.

The use of the CONVERT statement should become more clear with a few examples. First, the European number format problem. This is a conversion for incoming data. One effective conversion is to replace a comma with a decimal point and replace a period with a space. The statements for doing this with an interface at select code 7 are:

```
A$=",. "
CONVERT IN 7 PAIRS ; A$
```

The conversion has this effect:

12.345,6 is converted to 12 345.6

Since the free field format ignores spaces within a number and recognizes a decimal point, you do not even need an ENTER image to recognize the converted data. It is important to note that this CONVERT statement changes all periods to spaces and all commas to periods, whether they are part of a European number or simply part of a block of text. Since this could have some undesired effects, it is necessary to be able to turn off the conversion when it is no longer desired. The statement which cancels the conversion in this example is:

```
CONVERT IN 7
```

Giving only the direction and interface select code, without specifying PAIRS or INDEX or any other parameters cancels a previously selected conversion.

Control characters, such as carriage return or line feed, can also be converted. The following example shows the statements used to convert a carriage return to a line feed. This conversion is needed when entering data from a device which gives only a carriage return, without a line feed, as a delimiter.

```
A$=CHR$(13)&CHR$(10)
CONVERT IN 7 PAIRS ; A$
```

Another conversion example is the output of Extended Binary Coded Decimal Information Code (EBCDIC) instead of ASCII code. EBCDIC is another form of character representation used on certain types of computers. Since all the ASCII symbols have corresponding EBCDIC symbols, it is reasonable to choose the INDEX conversion mode using a string with 128 characters. In the following example, it is more important to understand the general process being used than to understand what the actual EBCDIC values are. The decimal equivalents of 128 EBCDIC characters are read from data statements and converted to string characters by the CHR$ function. The resulting look-up table is included in a CONVERT statement for interface select code 7. The INDEX specifier tells the computer to use the outgoing ASCII character as an index to find the equivalent EBCDIC character. For example, an ASCII right brace (decimal value 125) will convert to a CHR$(155), which is an EBCDIC right brace.

```
10 DIM A$[128]
20 A$=""
30 DATA 0,1,2,3,55,45,46,47,22,
   5,37,11,12,13,14,15,16,17,18
   ,19,53,61,50,38,24,25,63,36
40 DATA 28,29,30,31,64,90,127,1
   23,91,108,80,125,77,93,92,78
   ,107,96,75,97,240,241,242
```

```
 50 DATA 243,244,245,246,247,248
    ,249,122,94,76,126,110,111,1
    24,193,194,195,196,197,198
 60 DATA 199,200,201,209,210,211
    ,212,213,214,215,216,217,226
    ,227,228,229,230,231,232,233
 70 DATA 173,21,189,95,109,20,12
    9,130,131,132,133,134,135,13
    6,137,145,146,147,148,149
 80 DATA 150,151,152,153,162,163
    ,164,165,166,167,168,169,139
    ,79,155,74,7
 90 FOR I=1 TO 128
100 READ X
110 A$=A$&CHR$(X)
120 NEXT I
130 CONVERT OUT 7 INDEX ; A$
```

Section 4

Error Handling

Run-time errors on the HP Series 80 Personal Computer can be trapped by using the ON ERROR statement. You may already be familiar with the ERRN and ERRL functions which provide essential error information in a program. These functions can be used with the I/O ROM. However, all option ROMs share the error numbers starting at 101. So some other tools are necessary to identify the source of an error when more than one ROM is installed.

The I/O ROM provides two error functions in addition to the standard diagnostic capabilities of the HP Series 80 Personal Computer. These functions give the programmer the extra information necessary to isolate error conditions in a program.

ERROM — Provides a number which identifies the option ROM which generated the most recent error. If the most recent error was caused by the I/O ROM, the ERROM function returns a value of 192. Note that this function is only updated by option ROM errors. Therefore, ERROM "remembers" the last option ROM error, even if the most recent error in the system was not caused by an option ROM.

ERRSC — Provides the select code of the interface which generated the most recent interface-dependent error. Note that this function is only updated by interface-dependent errors. Therefore, ERRSC "remembers" the last interface error, even if the most recent error in the system was not caused by an interface.

Because other option ROMs share similar error numbers to those of the I/O ROM, and because these functions are not updated by every system error, it is important to interrogate the various error functions in the proper order. If you are looking for I/O errors in an error recovery routine, check first for ERRN>100. If there is a ROM error, check ERROM to find which ROM. Having determined that the I/O ROM generated the error, check ERRN for an interface error before looking at ERRSC. Error numbers 101 and 112 will not occur during a running program. All other errors below 123 are interface-dependent errors. Therefore, a simple test for ERRN<123 will tell if there was an interface error.

The following simple program segment shows the recommended order of function checks used to isolate I/O errors. This segment only displays an error message. An actual error recovery routine would also include statements to take whatever corrective action is appropriate in your specific situation.

```
 10 ON ERROR GOSUB 100
 20 !
 30 !
100 ! Test for non-I/O errors
110 IF ERRN<101 THEN GOTO 350
120 IF ERROM<>192 THEN GOTO 350
130 !
140 ! Test for interface errors
150 IF ERRN<123 THEN GOTO 260
160 !
170 !
```

```
180 ! Process ROM errors
190 DISP "I/O ERROR";ERRN;"at li
    ne";ERRL
200 !
210 ! Recovery routine goes here
220 !
230 RETURN
240 !
250 !
260 ! Process interface errors
270 DISP "I/O ERROR";ERRN;"at li
    ne";ERRL
280 DISP "Problem on select code
    ";ERRSC
290 !
300 ! Recovery routine goes here
310 !
320 RETURN
330 !
340 !
350 ! Process non-I/O error here
360 RETURN
```

There is a complete listing of all I/O errors, their meaning, and some debugging hints in appendix C.

Section 5

Specialized Transfers

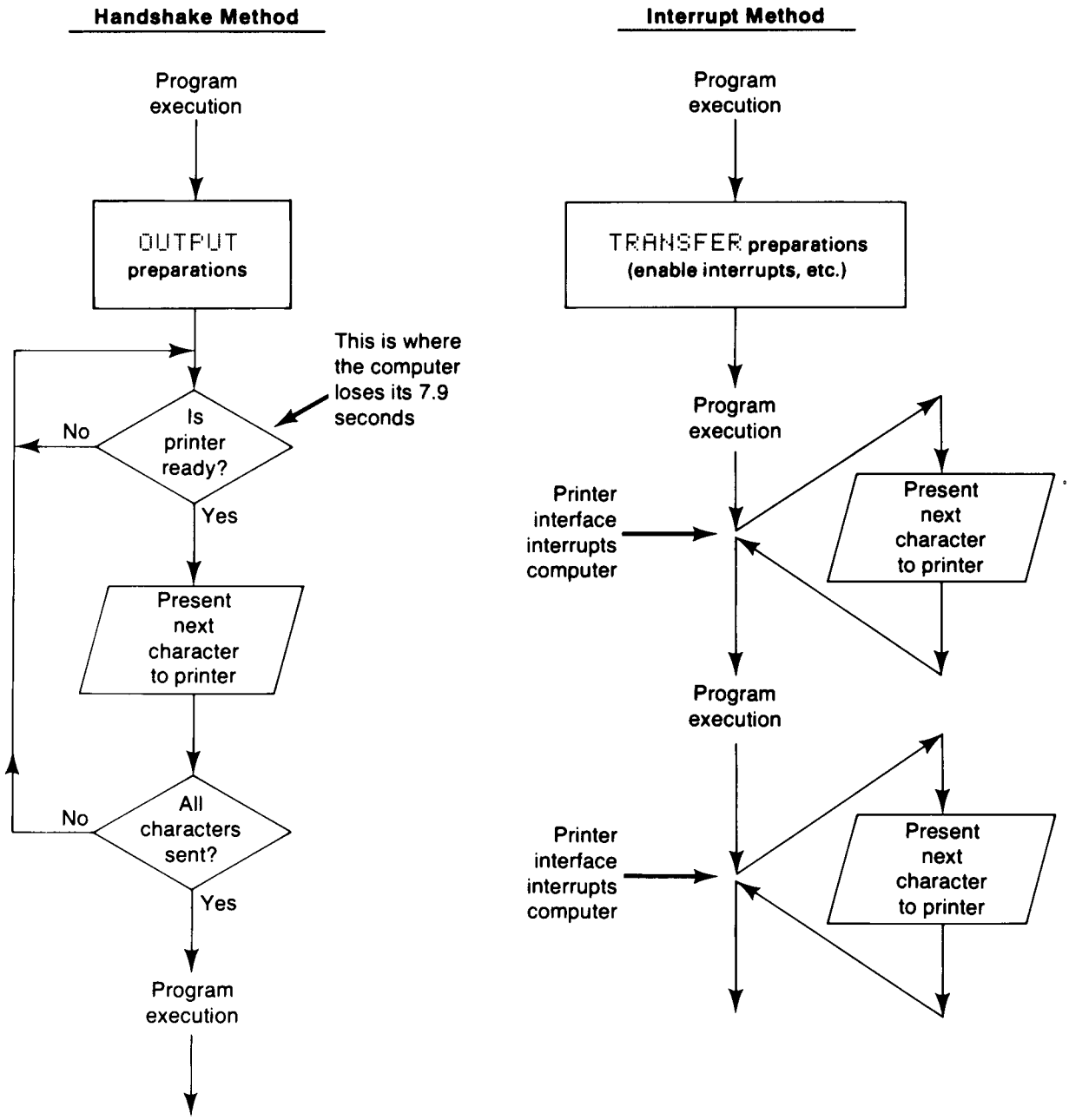
Introduction

This section deals with data transfers as they are implemented by the TRANSFER statement. The basic purpose of the TRANSFER statement is to provide a flexible tool for moving data into and out of the computer. The key word here is flexibility.

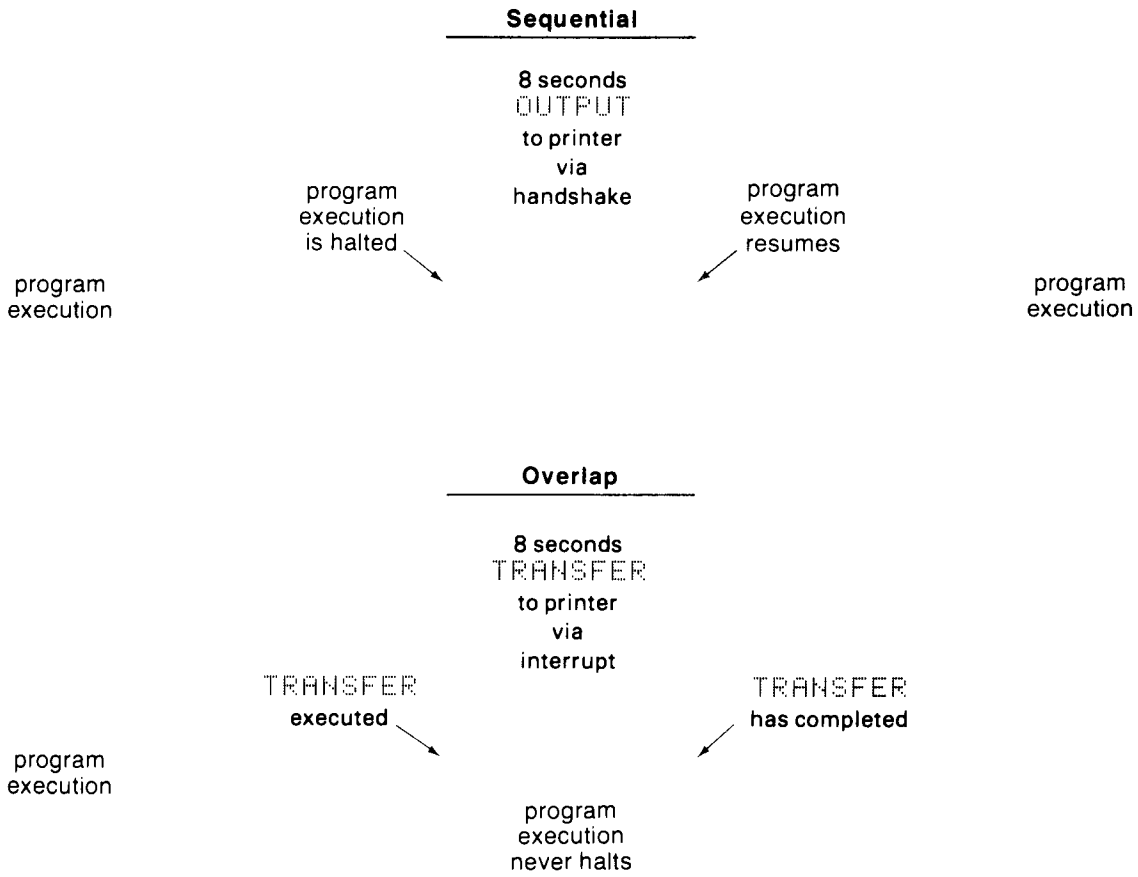
This flexibility allows you to better match the computer's speed to that of the peripheral it is communicating with. Take the case of a very slow device, such as a 10 character-per-second printer. It takes such a printer 8 seconds to print an 80 character line, but our computer could send those same 80 characters in less than .1 second. If the computer is forced to wait on the printer, then the computer is losing 7.9 seconds of computation time out of every 8 seconds! The computer's power can obviously be increased by gaining back that 7.9 seconds. Let's see how.

The following diagrams constrast the default handshake method used by OUTPUT and ENTER with the *interrupt* method of TRANSFER. When the computer executes the OUTPUT statement (for example), it is forced to handshake *each* character of the data list until all the data has been sent. Only then is the computer free to execute the next program statement, about 8 seconds later. On the other hand, the interrupt TRANSFER statement sets up some special pointers to the data and enables the printer interface to interrupt the computer. Then the computer is free to execute the next program line, about 10 *milliseconds* later! (Enabling an interrupt is like hanging up a telephone receiver: the telephone is now able to "interrupt" you by ringing whenever someone calls.)

The computer continues program execution until the printer is ready for another character. The printer interface interrupts the computer from whatever it was doing, and the computer then fetches the next character, updates its pointers, check to see if all data has been sent, then continues on with what it was doing. If all the data has now been sent, the computer disables further interrupts from the interface (like taking the telephone receiver off the hook—no more rings) before continuing on with the program.



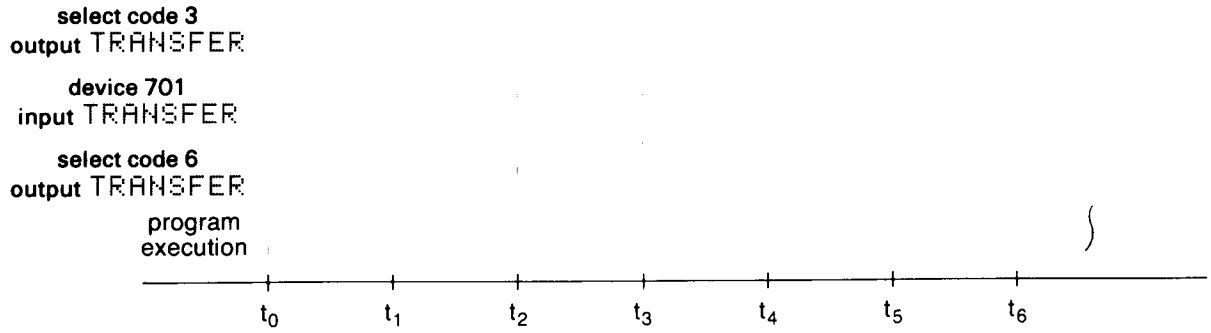
If the two methods are looked at in a broader sense, it is possible to see the real difference: the handshake method is a linear, or sequential operation, while the interrupt method is a parallel, or overlap operation. Consider the following diagram:



The sequential method effectively pauses the program for the duration of the OUTPUT operation, while the overlap method continues both with program execution and transfer operation.

An interesting possibility brought about by this overlap is that of multiple, simultaneous I/O operations. Suppose that the next program statement after the TRANSFER statement is *another* TRANSFER to a different device (a large-screen CRT monitor, for instance). Then *three* things are happening at once: the program is being executed, the printer is printing, and the external CRT monitor is displaying new characters.

The following diagram illustrates some of the power of overlapped I/O operations made possible with TRANSFER. (Please note that this is for interrupt transfer only, as explained later.)

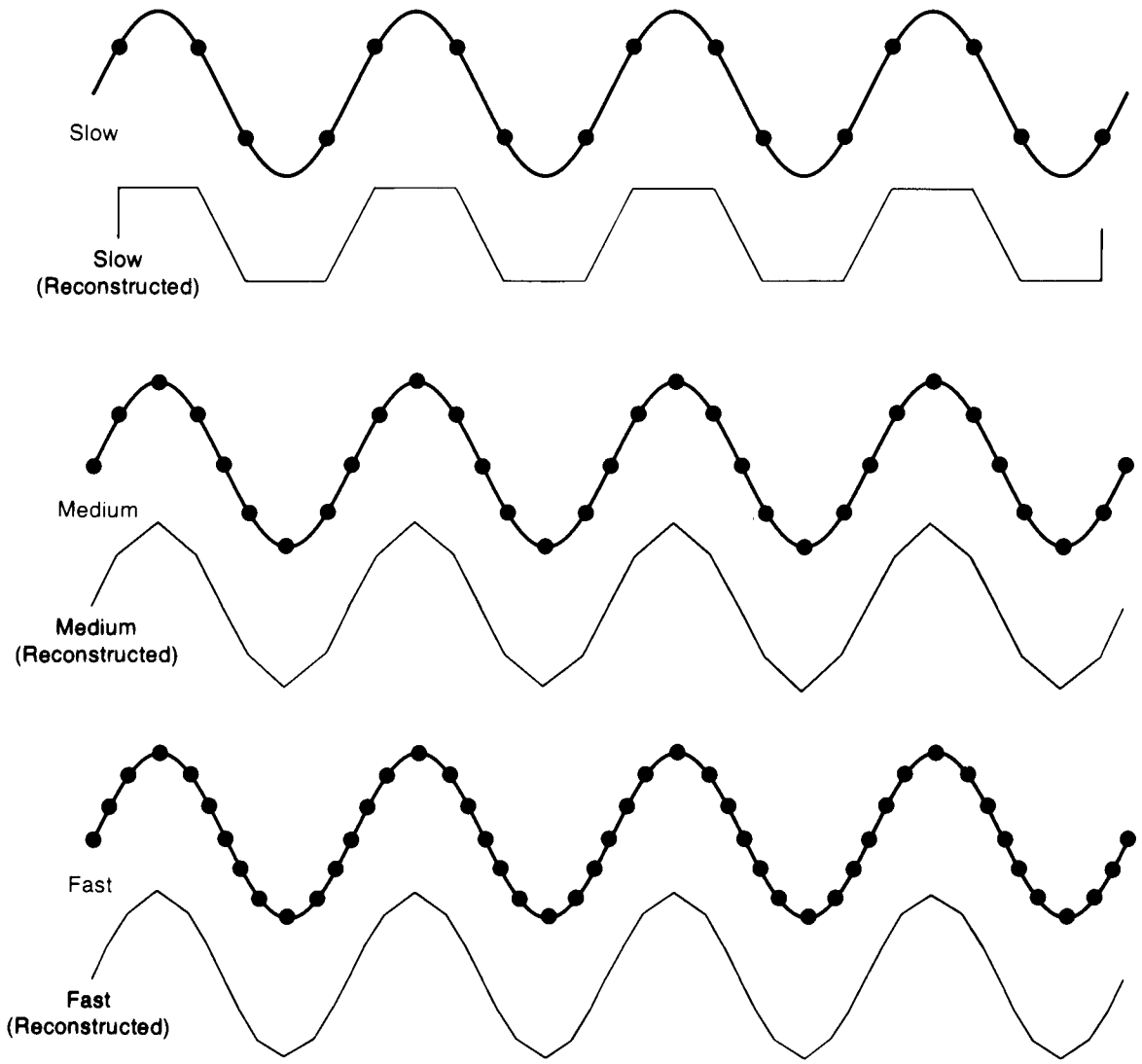


Program execution begins at time t_0, and some time later an output transfer to select code 6 is initiated (time t_1). At time t_2 an input transfer from HP-IB device 701 is started, and at time t_3 another output transfer is started, this time to select code 3. By now, three transfers and program execution are all going on at once. At time t_4, the input transfer terminates, and the two output transfers finish at t_5 and t_6. This overlap capability demonstrates some of the flexibility of system design made possible with the transfer statement.

There is another side to the transfer statements flexibility: speed. Certain operations are ineffective or impossible at slow or medium speeds, and require instead a high-speed transfer. Take the case where it is desired to analyze a signal's waveform by using an HP 3437A Voltmeter. This voltmeter is capable of producing a 3½ digit voltage reading up to 3600 times per second, which is a transfer rate of about 25,000 characters per second! The computer is not quite that fast, however; its transfer rate is closer to 20,000 characters per second.

The other type of TRANSFER statement is the *fast-handshake transfer.* This is an entirely different type of operation from the interrupt transfer. A fast handshake transfer represents the fastest possible data movement operation.

The following diagram illustrates the effect of sampling a signal at slow, medium, and high rates of speed (sample points are represented by dots):



The slow sample rate provides at best an inaccurate picture of the signal, while the high sample rate comes much closer to approximating the actual shape of the signal.

So where does TRANSFER fit into this picture? Consider the ENTER and OUTPUT statements with their extensive formatting and conversion capabilities as being like a Rolls-Royce automobile with electric windows, television, liquor cabinet, automatic transmission, and other accessories.

A fast-handshake transfer is then comparable to a Formula I race car, with no windows, manual transmission, one seat, a spine-jolting ride, and that is capable of speeds over 300 kilometers per hour. Its main objective is performance.

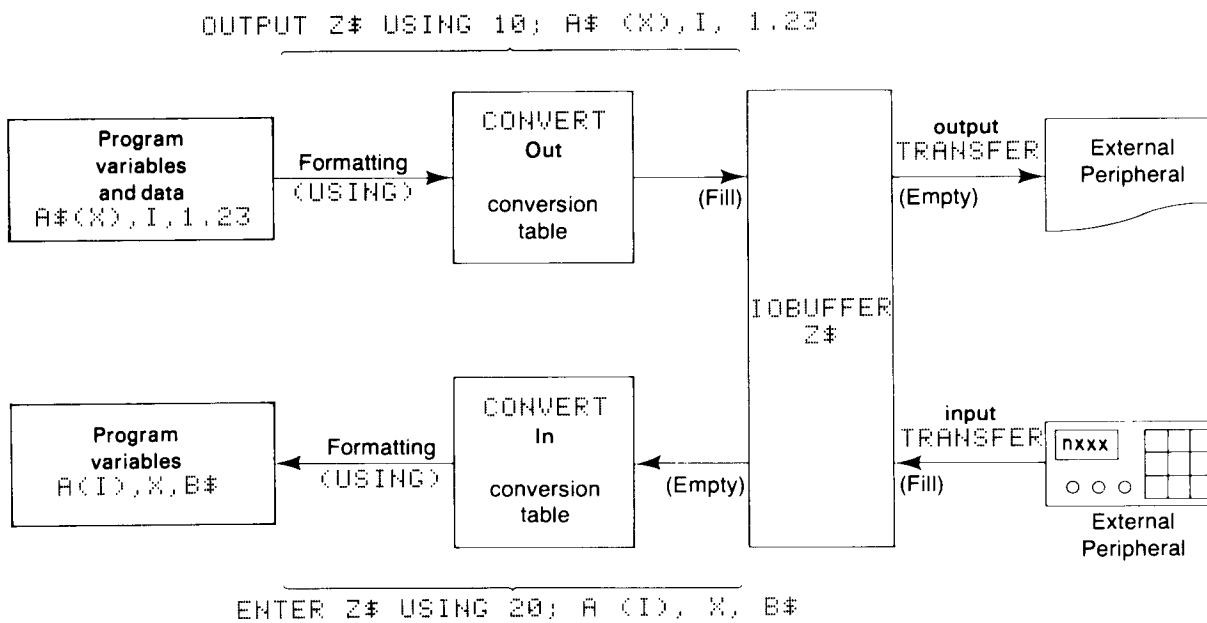
The fast-handshake transfer delivers the highest data transfer rate possible. When a fast-handshake transfer operation is in progress, **all** other activities stop. Even the computer's (RESET) key is disabled and will have no effect until the transfer completes.

As with the Formula I race car, you pay a price for performance.

Using Buffers

A buffer is a section of read/write memory set aside for the purpose of temporary data storage. It is used to either input data, output data, or both by means of the TRANSFER statement. There is no formatting or data conversion done by TRANSFER, so what is in the buffer is what is sent to the peripheral device. The same holds true for data being input by TRANSFER; it is placed in the buffer exactly as it is received from the device.

To illustrate how output, enter, conversion, formatting, I/O buffers, and transfers work together consider the following diagram.



The OUTPUT statement takes data from program variables and does any necessary formatting while placing that data into its ASCII representation. Then, if an output conversion is in effect, the ASCII characters are converted accordingly and placed into the I/O buffer at the position specified by the *fill pointer*. The I/O buffer is *full* when the fill pointer is at the end of the buffer (the string Z$ in this case). You should be aware that default OUTPUT and TRANSFER operations to a buffer place a carriage return and a line feed at the end of the data in the buffer. This means that the buffer should be dimensioned to a length greater by two on the HP-86 and HP-87, when using the default OUTPUT and TRANSFER operations.

The output transfer takes characters from the I/O buffer at the position specified by the empty pointer. These characters are sent to the specified interface and its associated peripheral. This is done either by interrupt or fast-handshake as specified by the programmer. When the transfer completes, the interface's end-of-line character sequence is sent.

The input transfer accepts characters from the specified interface (and its associated peripheral) and places them into the I/O buffer at the position specified by the fill pointer. Again, this is done either by interrupt or fast handshake as specified by the programmer.

The ENTER statement takes characters from the I/O buffer at the position specified by the empty pointer. If an input conversion is in effect, these characters are converted accordingly, formatted as necessary, and changed into the proper internal representation for the program variables. If you are entering data from an active buffer, errors can be avoided by using the form ENTER 2# USING "#,#K";A$.

This form of ENTER removes the requirement for statement and variable terminators, which may not be in the buffer yet.

The Pointers

On the HP-85, when a string variable is first designated as an I/O buffer (by executing an IOBUFFER statement), its dimensioned length is effectively reduced by eight characters. This is to provide room for four "pointers." There is a fill pointer, an empty pointer, an active-out select code, and an active-in select code.* On the HP-86 and HP-87 the buffer length is equal to the dimensioned length.

The *fill pointer* first equals zero (0). This pointer always contains the same value as that returned by the LEN function for the string variable. Placing a character into the buffer goes as follows: 1) increment the fill pointer, 2) store the character. This operation is *automatically* handled by the OUTPUT (fill pointer equals LEN function) statement and also by any string variable assignment operations such as Z$=Z$&A$. You do not normally need to assign values to the fill pointer.

The *empty pointer* first equals one (1). Taking a character from the buffer goes as follows: 1) read the character, 2) increment the empty pointer. This operation is performed automatically by the ENTER statement.

A buffer is *full* on the HP-83/85 when the fill pointer equals the string's dimensioned length minus 8. A string with a dimensioned length of 8 would not be a very useful buffer, as it would be full and empty at the same time, without any data being placed in it at all! Any OUTPUT operation to a full buffer will result in an error. On the HP-87 the length declared is the same as the usable length.

A buffer is *empty* when the empty pointer equals the fill pointer plus one. This has no relationship to the dimensioned length of the string. When a buffer is emptied, the full pointer is reset back to 0 and the empty pointer is reset back to 1. This is exactly the same effect as executing the IOBUFFER statement, except that the IOBUFFER statement also initializes (destroys) conversion table pointers.

---

*On the HP-87, additional memory is allocated for the pointers (additional to the length that the buffer is declared to be). Because of this, on the HP-87, the number of I/O buffers is limited to ten. Declaring more than 10 buffers will result in an error and abortion of the program at that point.

Buffer Activity

When a TRANSFER statement is executed, the specified buffer is then an *active* buffer. The buffer may be *active-out*, *active-in*, or both, depending upon the direction(s) of the transfer(s). The buffer is assigned an *active-in select code* when an input TRANSFER statement is executed. An *active-out select code* is assigned when an output TRANSFER statement is executed. For example when the following interrupt transfer to select code 6 is executed—

```
TRANSFER Z$ TO 6 INTR
```

the active-out select code equals 6.

A buffer is made *inactive* when the transfer completes. This is a direction-specific inactive state; that is, buffer may be active-out but inactive-in, or vice-versa. When an input transfer completes, the buffer's active-in select code is set to zero (0). Similarly, when an output transfer completes, the buffer's active-out select code is set to zero (0).

Buffer Status and Control

The four buffer pointers can be checked by means of the STATUS statement. The four I/O buffer status registers are as follows:

Status Register	Default Value	Register Function	Statement used to read register value of buffer Z$
SR0	1	Buffer empty pointer	STATUS Z$,0,T0
SR1	0	Buffer fill pointer	STATUS Z$,1;T1
SR2	0	Active-in select code	STATUS Z$,2;T2
SR3	0	Active-out select code	STATUS Z$,3;T3

These registers may be read at any time on an active or inactive buffer, but attempting to read the status of a non-buffer string variable (that is, if no IOBUFFER statement has been executed for that string variable) results in an ERROR.

An example of using buffer status registers to control program flow follows. In the example, a string variable is dimensioned to 88 characters (to allow for 80 characters of data after becoming a buffer on the HP-85) and declared as an I/O buffer. Data is output to the buffer, a transfer out to an HP-IB printer is initiated, then the buffer's active-out select code is checked to determeine when the transfer has completed. The program ends when the transfer completes. The buffer may be dimensioned to 80 characters on the HP-86 or HP-87.

```
 10 DIM Z$[88]
 20 IOBUFFER Z$ ! Usable size of Z$ is 80 (88 on the HP-86/87)
 30 FOR I=0 TO 1 STEP .1 ! EOL is suppressed
 40 OUTPUT Z$ USING "#,D.DDD,X" ; SIN(I)
 50 NEXT I
 60 ! Show values of buffer registers
 70 STATUS Z$,0 ; T0,T1,T2,T3
 80 PRINT "Registers before TRANSFER ",T0,T1,T2,T3
 90 TRANSFER Z$ TO 701 INTR
100 STATUS Z$,0 ; T0,T1,T2,T3
110 DISP T0,T1,T2,T3
120 IF T3<>0 THEN GOTO 100
130 PRINT "Transfer completed"
140 END
```

The variable T0 shows how many characters have been taken from the buffer. T1 shows how many characters total are in the buffer, and T3 indicates select code activity.

For another example, assume device X is to send three numeric values followed by a carriage return, line feed. The following program displays the buffer registers until the transfer completes. An ENTER is then executed to take the values out of the buffer, which are then printed.

```
 10 ! Our device X is at HP-IB address 721
 20 DIM Z$[88]
 30 IOBUFFER Z$
 40 TRANSFER 721 TO Z$ INTR ; DELIM 10
 50 ! Note that 10 is the decimal value for line-feed
 60 STATUS Z$,0 ; T0,T1,T2,T3
 70 DISP T0,T1,T2,T3
 80 IF T2<>0 THEN GOTO 60
 90 ENTER Z$ ; X,Y,Z
100 DISP X,Y,Z
110 END
```

If device X is a very slow device, you could "watch" characters come into the buffer by means of variable T1. The variable T0 is not altered until the ENTER is executed, and T2 reflects the TRANSFER statement activity. When the TRANSFER is initiated, T2=7; when the TRANSFER completes, T2=0.

The buffer empty pointer and the buffer fill pointer can be assigned new values by using the CONTROL statement. This gives you the capability of sending the same data over and over again without having to re-compute the data, for example. The following table shows these registers and how they are accessed:

Control Register	Default Value	Register Function	Statement used to write register value of buffer Z$
CR0	1	Buffer empty pointer	CONTROL Z$,0;V0
CR1	0	Buffer fill pointer	CONTROL Z$,1;V1

These buffer registers may be written to at any time, but attempting to write to a control register of a non-buffer string variable (before an IOBUFFER statement has been executed for the string variable) results in an error.

In the following example, 360 values are computed for SIN(X) to represent one complete cycle of a sine wave. These values are output to the buffer Z$ and subsequently sent to device X, a digital-to-analog converter, by a fast-handshake transfer.

When the transfer completes (T2 = 0), the buffer empty pointer is automatically reset to 1, the fill pointer is set equal to 360, and the transfer restarted. This continues indefinitely until the (PAUSE) key is pressed to stop the program. The effect of this program is to produce a near-continuous sine wave from device X. (Details of device X are purposefully left out here to avoid confusing the issue.)

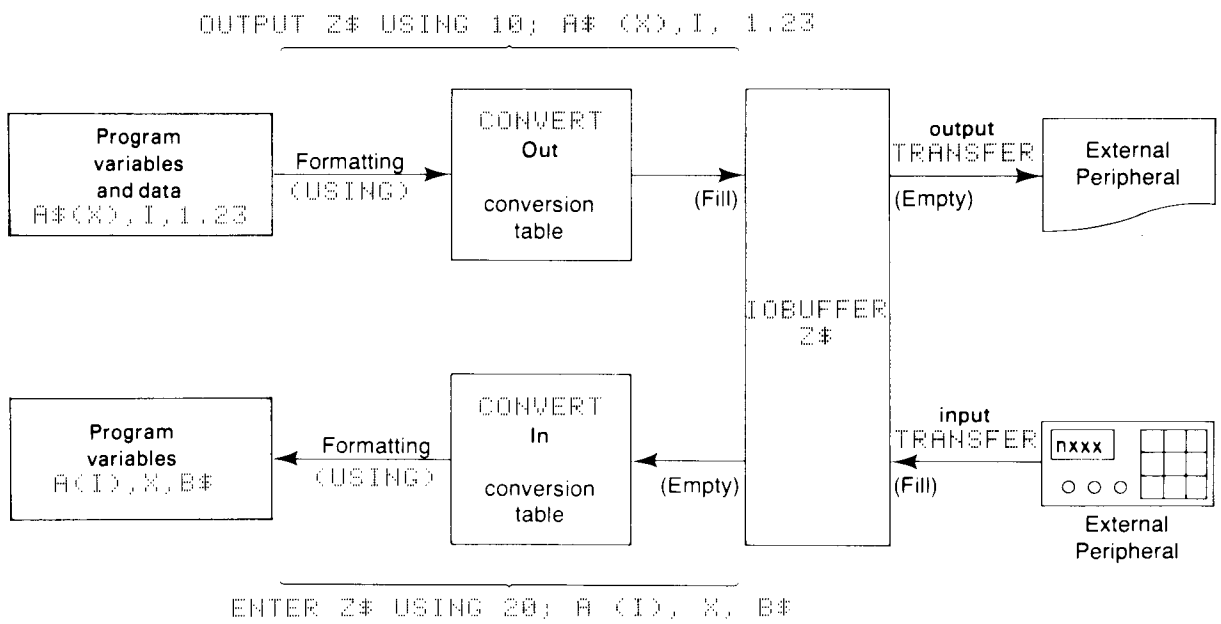
```
 10 ! Make room for 360 eight-bit values plus pointers
 20 DIM Z$[368]! 360 values are adequate for the HP-86/87.
 30 IOBUFFER Z$
 40 DEG
 50 FOR I=0 TO 359
 60 OUTPUT Z$ USING "#,B" ; SIN(I)*255
 70 DISP I;@ NEXT I
 80 TRANSFER Z$ TO 5 FHS
 90 CONTROL Z$,1 ; 360
100 GOTO 80
110 END
```

Lines 10-40 merely set up the buffer and place the computer in degrees mode. Lines 50-70 fill the buffer with the 8-bit value for each of 360 degrees (one complete sine wave), and line 80 starts the fast-handshake transfer. Line 90 is executed when the transfer is complete. The buffer fill pointer is set back to 360 (so it looks full again), and the transfer is restarted.

By exercising control over the buffer empty and fill pointer, it is possible to retransmit data, transmit any portion of the data in the buffer, to write data into any section of the buffer, read data out of any section of the buffer, etc. These operations may not be ones that you need to use in your application, but the flexibility they provide you could make feasible certain I/O operations not possible through any other means.

Data TRANSFERS

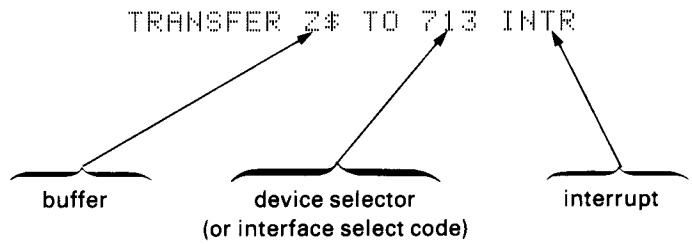
The TRANSFER statement has been mentioned several times up to this point. In combination with the IOBUFFER, it provides you with unmatched flexibility in tailoring and optimizing a program to exchange data with one or more peripheral devices. The diagram below shows the relationship of the transfer with the I/O buffer, conversion tables, and program variables.



The transfer itself is the easiest section of the entire picture to understand. Simply stated, an output transfer takes characters, or bytes, out of the buffer from the position specified by the buffer empty pointer and sends them to the external peripheral. Conversely, the input transfer takes characters from the external peripheral and places them in the buffer at the position specified by the buffer fill pointer.

Output TRANSFER

For an output transfer, you merely specify whether an interrupt or a fast-handshake transfer is to be performed. For example, to specify an interrupt output transfer, the following statement could be used:



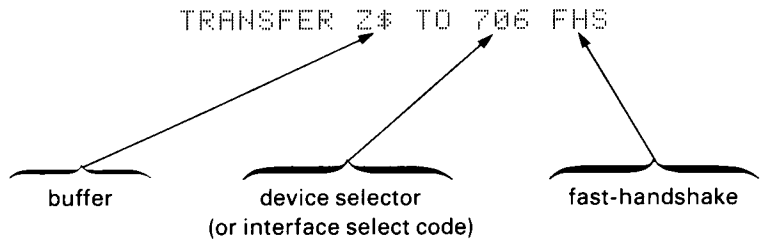
The operation of an interrupt output transfer is as follows:

- When the statement is executed, the interface involved (in this case, device 13 on an HP-IB Interface at select code 7) is automatically enabled to *interrupt** the computer when ready to accept a new character.
- Thereafter, each time the interface interrupts, the computer temporarily suspends program execution long enough to move the next character from the buffer to the interface. (It is then the interface's responsibility to see that the new character is properly sent to the peripheral.)

*This is a hardware-level interrupt, and is distinct from the software-level end-of-line branch discussed in Section 6.

- When the buffer is finally emptied (described in the Using Buffers section), the computer disables (or turns off) further interrupts from the interface. The transfer is now essentially complete, although the interface may still be sending out the specified end-of-line character sequence (normally a carriage return/line feed: see the appropriate interface owner's manual).
- When the transfer is terminated, a check is made for a user-defined end-of-transfer branch. If one is defined, then the branch is taken upon completion of the current program line. This is explained more thoroughly in the section End-of-Line Branching.

A fast-handshake output transfer is specified as—



The operation of a fast handshake output transfer is significantly different:

- The computer suspends program execution and dedicates itself to the task of moving all the characters in the buffer to the interface. The computer *totally* devotes itself to this task until the buffer is emptied. No other interrupts are allowed, not even a (RESET)! This means that once a fast-handshake transfer begins the computer will see it through to completion, and all other operations are ignored until that time.
- When the buffer is empty, the transfer is essentially finished. (The interface may still be in the process of sending the end-of-line character sequence.) If a user-defined end-of-transfer branch is specified, the branch is taken upon completion of the current program line.

Input TRANSFER

The input transfer is essentially the same as the output transfer, except that data is being moved from the external peripheral to the computer. However, in addition to specifying whether an interrupt or fast-handshake transfer is to be performed, you may also specify the terminating condition(s) for the transfer. The terminating conditions for *both* interrupt and fast-handshake transfers are:

- COUNT—the number of characters or bytes of data to be input before the transfer is considered complete. Use COUNT when the number of characters or bytes being sent from the peripheral device is known.
- EOI—an interface-specific terminating condition. In the case of the HP-IB Interface, some devices set the EOI control line when sending the last byte of data indicating end-of-data. This terminates the transfer if EOI is specified.
- *Default termination*—a full buffer termination. An input transfer terminates with the first occurrence of any specified terminating condition or a full buffer. (Obviously, if there is no room left in the I/O buffer, the transfer cannot continue.) If no other terminating condition has been specified, an input transfer will terminate when the I/O buffer becomes full.

Additional terminating conditions for interrupt transfers *only* are:

- DELIM—the numeric value of the character or byte of data that indicates all data has been sent. Use DELIM when the number of characters or bytes being sent from the peripheral device is either not known or varies for some reason. The peripheral device should not send the DELIM character as part of the data being sent, or else the transfer will be terminated prematurely!
- *Interface termination*—an interface specific termination. Certain interfaces allow you to specify additional terminating conditions by writing to the interface control registers. These conditions may be either receipt of user-specified characters or line status, so consult the appropriate interface programming section to determine which, if any, conditions your interface supports.

Programming with TRANSFERS

Every transfer operation requires an I/O buffer, which in turn requires a dimensioned string variable of adequate size. *After* executing the IOBUFFER statement, any CONVERT statement(s) for that buffer may be executed. This is because the IOBUFFER statement initializes all the buffer pointers, including those to any conversion tables.

If there is any necessary interface initialization to do, it may be done anytime before executing the transfer. If your program uses end-of-line branching (ON EOT: see the End-of-Line Branching section), the appropriate statement(s) should be executed before the TRANSFER statement.

An input transfer may be executed at any point after the setup sequence described above. An output transfer, however, requires data in the buffer before anything can be sent. Therefore, an output or a string assignment operation needs to be performed before attempting to execute the output transfer.

It has been said that either an output or a string assignment operation needs to be performed before executing an output transfer so that there is data available in the buffer. This is not entirely true. Nor is it necessary to have all the data in the buffer before initiating an output interrupt transfer. Also, it is not necessary to wait for an input interrupt transfer to complete before reading data from the buffer. (Fast-handshake transfers are totally sequential operations and do not apply to the following discussion.)

If the external device is sufficiently slower than the computational speed of the computer, it is possible to initiate an output transfer with the buffer only partially filled. Then, as more data becomes available to send to the device, the new data can be output (for example) to the buffer while the transfer is still in progress. This process can go on indefinitely until either: 1) all data has been sent or, 2) the buffer is filled (then the user's program must wait for more buffer space) or, 3) the buffer is emptied (then the transfer terminates and it must be restarted).

Case 1 above is straightforward and needs no explanation. Case 2 is determined by the buffer status, and the program can wait for more buffer space by monitoring buffer status (with the STATUS statement). The occurrence of case 3 indicates that the peripheral device is actually faster than the computer in this instance. (This may be due to excessive computation being performed or some other circumstance causing a delay in the program.) An end-of-line branch or a status check on the buffer indicates a transfer completion, and the decision to continue the transfer can be made at that time. The following example program illustrates how these conditions might be dealt with:

```
10 ! This program shows how you might check for cases 1,
20 ! 2,and 3. The buffer size is determined empirically
30 ! to optimize memory usage with transfer rate.
```

```
 40 DEG @ DIM B$[160]
 50 DEF FNA(X) = INT(SIN(X)*35+36.5)
 51 CONTROL 7,16 ; 0 ! This turns off CR/LF from the
    interface by specifying zero EOL sequence characters
 60 IOBUFFER B$
 70 !
 80 FOR X=0 TO 359 STEP 10 ! The For-Next loop handles
    case 1
 90 STATUS B$,0 ; S1,S2,S3,S4
100 ! Test for buffer empty before putting in data (case
    2)
101 Z=FNA(X)
110 IF S1+Z>145 THEN GOTO 90
115 I$=VAL$(Z)&"X,A"
120 OUTPUT B$ USING I$ ; "*"
130 ! Test the active-out select code before starting the
    transfer
140 IF S4=0 THEN TRANSFER B$ TO 720 INTR ! (case3)
150 NEXT X
160 !
170 GOTO 80
180 END
```

Section 6

End-of-Line Branching

Some Background on Interrupts

If you've never heard the term "interrupt" with regards to a computer before, then a simple way to think of it is like a telephone ringing while you are working. It is a means of diverting your attention from whatever it is you are doing. "Servicing" an interrupt is similar to the act of going over to answer the ringing telephone. When the telephone business is completed, you typically resume the "interrupted" activity where you left off.

If you have a switch that can disconnect the telephone bell (so it can't ring), then you can disable or enable that interrupt by the setting of the switch. The computer has essentially the same facility.

In addition, the computer has two types of interrupts to deal with: low-level, or hardware interrupts; and high-level, or software interrupts. In general, the hardware interrupt is used by the computer for its own purposes, and is transparent to the user. Hardware interrupts make possible such things as the INTR type of TRANSFER, where program execution and I/O are concurrent operations. However, external (and internal) events can also trigger hardware interrupts that require specialized service routines to deal with them. Because there is such a wide variety of interrupt causes possible, it is necessary to make provisions for the programmer to custom tailor service routines for those interrupts specific to his system. This provision is the software interrupt, or end-of-line branch.

Imagine a "ghost" IF *<event>* THEN GOSUB *<line #>* statement appended to the end of each program line, like the ones below:

```
50 PRINT "Result=";X @IF <event=true> THEN GOSUB <routine>
60 X=X+Y @IF <event=true> THEN GOSUB <routine>
```

In effect, at the end of each program line, a check is made for the *event*, which may be an error, a hardware interrupt, a select code timeout, a transfer completion, a special function key-press, or a timer timeout. (End-of-line service for three of the above events is already provided for in the HP Series 80 mainframe: errors, special function keys, and timers.) When one of the events occurs, a special portion of the program can be executed to deal with that event.

The programmer also has the ability to disable or enable the end-of-line service facility for each event, as will be shown in the next section on programming for end-of-line branches.

End-of-Line Branch Programming

Introduction

This section shows you how to program service routines to deal with end-of-line branches for I/O related events. At the end of this section is a sub-section entitled "Interactions and Permutations." The discussion there deals with problems and questions you may encounter such as GOSUB vs. GOTO.

Interface Interrupts

Each interface (HP-IB, Serial, BCD, etc.) has a control register that allows you to specify an interrupting condition. This is control register CR1, the Interrupt Mask register. When you set a bit in that register, you *enable* the interface to *interrupt* the computer when the event corresponding to that bit occurs. The interrupt event is determined by referring to the programming section for the particular interface. For example, the HP-IB Interface provides for an interrupt on SRQ, or Service Request. The SRQ interrupt is enabled by setting bit 3 (decimal value 8) of control register CR1. To enable the SRQ interrupt for the HP-IB Interface at select code 7, execute

```
ENABLE INTR 7;8
```

This is equivalent to

```
CONTROL 7,1;8
```

and both statements set bit 3 of control register CR1 on select code 7.

Suppose (for simplicity) that only one device is connected to the HP-IB Interface, and that device asserts SRQ only to indicate it is ready to send a numeric value to the computer. The statement we'll use to read this value is

```
ENTER 701;V1
```

This is the extent of our example service routine, which will take the form of a subroutine in the program.

Once the computer has read the new value from the device, it must again enable the SRQ interrupt so that another value can be read when the device is again ready. The service routine begins to take shape:

```
1000 ! SRQ Service Routine for select code 7
1010 STATUS 7,1 ; A ! Always read interrupt cause register
1020 ENTER 701 ; V1
1030 ! Re-enable interrupt and return on same line
1040 ENABLE INTR 7;8 @ RETURN
```

Note that the ENABLE and the RETURN are on the same program line. There are some occasions where this is a necessary practice, and this is discussed under "Interactions and Permutations."

It is necessary to specify *where* the service routine for an event is, and also whether the routine is a subroutine (GOSUB) or just a program segment (GOTO). This is accomplished by the ON INTR statement. Note that the computer must know where to go and what to do before an interrupt occurs, or else it will be forced to ignore it. Therefore, the ON INTR statement should be executed before the ENABLE INTR statement, as shown in the example below.

```
10 ON INTR 7 GOSUB 1000
20 OUTPUT 722 ;"F1R7T2T3D1"
30 V1=X=0
40 LOCAL 722
50 ENABLE INTR 7;8
```

```
   60 ! The following lines are a dummy program
   70 X=X+1 @ DISP X,V1
   80 WAIT 100 @ GOTO 70
      :
 1000 ! SRQ Service Routine for select code 7
 1010 STATUS 7,1 ; A
 1020 ENTER 722 ; V1
 1030 ENABLE INTR 7;8 @ RETURN ! This must be done to reset
      interrupt
 1040 END
```

Line 10 specifies the location (line 1000) and type (GOSUB, not GOTO) of the service routine for select code 7. Line 20 initializes the HP 3455A Digital Voltmeter to take readings whenever it is triggered. The LOCAL statement of line 40 allows the DVM to be triggered manually, and the ENABLE statement enables the interface for SRQ interrupt (bit 3 of the interrupt mask is set). Lines 70 and 80 merely display an incrementing counter and the last reading taken from the DVM.

Each time an SRQ is received, program execution of lines 70 and 80 is suspended, and the subroutine at line 1000 is executed. The ENTER is performed, which takes a new value for V1. The SRQ interrupt condition is again enabled in line 1030, and the RETURN allows the program display loop to resume. However, the value for V1 that is now displayed is the new value just read by the service routine.

What would happen if the ENABLE INTR on line 1030 were removed?

Simple. The first SRQ would be serviced as described above, and the program loop then resumed. However, no more SRQ interrupts would be detected, even if the HP-IB device were frantically sending the SRQ message to the interface! The interface *ignores* the event—regardless of its importance, until an ENABLE INTR statement is performed to again enable the interface to be able to interrupt the computer. Also, the ENABLE INTR statement sets the status register to the current state.

To disable or cancel the end-of-line branch condition set up by ON INTR, simply execute an OFF INTR statement for the appropriate select code. No more branches will be taken until another ON INTR statement is executed.

It is good programming practice to always read the interrupt cause register. Note, however, that the execution of a STATUS statement clears the status register so it cannot meaningfully be read again immediately. It may also indicate more than one interrupt.

Timeouts

It is possible for an external device to exhibit symptoms of intermittency or disconnection from time to time. This "falling asleep" is generally caused by an electronic malfunction or some other unusual condition, such as being switched off. It is not as important to focus here on the possible *causes* of device failure so much as the possible *effects*.

When a device fails, for whatever reason, the immediate effect on the computer is that it is no longer handshaking data to or from the computer. (Handshaking is a means of reliably transferring data between two devices. The sender makes data available, signals that data is ready, and the receiver accepts the data and signals that it has taken the data. This protocol is called handshake.) If an ENTER, SEND or OUTPUT operation is in progress, a loss of handshake means no data can be transferred and the operation cannot complete. If the operation does not complete, the program "hangs" until the operator becomes aware that something has gone wrong. This is unacceptable for most I/O systems, and especially those where unattended operation is frequent.

The SET TIMEOUT statement gives you the capability of establishing a maximum time period for the computer to wait on an interface to handshake data. If an interface exceeds the SET TIMEOUT limit specified, two alternative courses of action may be selected. One method simply aborts the I/O operation in progress and continues program execution at the next line. The other method executes a timeout service routine after the I/O operation is aborted.

The following example shows the simpler form of SET TIMEOUT, where no timeout service routine is specified:

```
10 SET TIMEOUT 7;10000
20 OUTPUT 706 ;"Simple test data"
30 DISP "Now at line 30"
40 END
```

If device 706 stops responding to handshake before the output operation is complete, then after 10 seconds line 30 will be executed. (Line 30 would also be executed if the output were successful, obviously.)

A more sophisticated method for dealing with a timeout condition is to execute a timeout service routine when an I/O operation is aborted due to a timeout. A separate service routine may be programmed for each select code to deal with the specific device or conditions for that select code. The ON TIMEOUT statement provides this capability, as shown below:

```
  10 SET TIMEOUT 7;2000
  20 SET TIMEOUT 5;4000
  30 ON TIMEOUT 7 GOSUB 1000
  40 ON TIMEOUT 5 GOSUB 1100
  50 ENTER 706 ; V1,X
  60 OUTPUT 5 ;V1,X
  70 GOTO 50
     :
1000 ! Timeout service for select code 7
1010 PRINT "HP-IB Timeout"
1020 RESET 7 ! Attempt to recover
1030 RETURN
1100 ! Timeout service for select code 5
1110 PRINT "Select code 5 failure"
1120 RESET 5
1130 RETURN
```

Obviously, in a dedicated system, more significant action could be taken than simply printing a message, aborting the operation with a RESET, and returning to the program. For example, a flag could be set by the timeout routine which would be checked before attempting an I/O operation to that select code. This flag might indicate printer out of paper or tape reader at end of tape, which would make I/O to such a device a useless endeavor. It might also be the only means available of determining when an operation has completed. The following example shows how a flag might be used:

```
10 SET TIMEOUT 6;1500
20 ON TIMEOUT 6 GOSUB 1000
21 ! Clear timeout flag (F1)
30 F1=0 @ DIM V(50)
40 FOR I=1 TO 50
50 IF F1<>0 THEN GOTO 80 ! Test timeout flag before
   ENTER
```

```
  60 ENTER 6 ; V(I)
  70 NEXT I
  80 PRINT I;" Values entered" @ STOP
     :
1000 DISP "Timeout : end of data"
1010 F1=1 ! Set timeout flag
1020 RETURN
1030 END
```

When a timeout occurs, the flag F1 is set equal to 1, and the IF statement of line 50 causes the ENTER statement of line 60 to be skipped. The loop is exited and the program continues. (In the example, only a PRINT and STOP are shown for simplicity.) In this case, the cause for the timeout is not device failure, but rather the end of data to be received.

As with ON INTR, there is a corresponding disable statement for timeout end-of-line branches. The OFF TIMEOUT statement can be used to cancel end-of-line service for a specified interface timeout. The SET TIMEOUT statement with a time limit of 0 can also be used, which sets an (almost) infinite timeout limit for the interface. For more information regarding syntax rules, refer to the alphabetical syntax listing section of this manual.

Transfer Terminations

There are two means of determining when a transfer operation has completed. Both are straightforward, but using an end-of-line branch service routine allows the greatest flexibility of program design.

As is mentioned in "Using Buffers", buffer status registers SR2 and SR3 indicate input and output activity, respectively. When the appropriate register becomes equal to 0, that transfer has terminated. However, this requires rechecking the status register until the transfer terminates, which inhibits program execution. If there is other work to be done, a fairly convoluted program logic will be the result of trying to monitor transfer status and also getting the other work done. There is a simpler way.

When an ON EOT (On End Of Transfer) statement is executed, an end-of-line branch is enabled for transfer termination for the specified interface. Then you can program the operations necessary to deal with the transfer termination into the ON EOT service routine. These operations might include placing new values into the buffer and re-initiating the transfer (for output) or taking values from the buffer and placing them into program variables (for input).

To illustrate how the ON EOT statement might appear in a program, consider the following example. An HP 3437A Voltmeter is programmed to take a reading every .9 seconds. An interrupt transfer is set up for 100 readings of seven characters each, plus a final carriage return/line feed and the eight extra bytes needed for pointers.

When the transfer completes, the service routine at line 1000 is executed which saves the readings and initiates a new transfer.

```
10 ! Transfer termination service routine example
20 ON EOT 7 GOSUB 1000
30 OUTPUT 724 ;"D.9SN100SE0SR3T1F1"
40 I=1 @ X=0 @ DIM D$[710],D(100)
50 IOBUFFER D$
60 ! This program assumes 10 data files are already
```

```
   70 ! created, and that the file names are in the next
   80 ! data statement.
   90 DATA data1,data2,data3,data4,data5,data6,data7,data8,
      data9,data10
  100 TRANSFER 724 TO D$ INTR
  110 X=X+1 ! Simple "work" loop (dummy)
  120 DISP X
  130 GOTO 110
      :
 1000 ! EOT service routine
 1010 ! Read next file name and open file on tape.
 1020 READ F$@ ASSIGN# 1 TO F$
 1030 ! Take values from buffer
 1040 FOR N=1 TO 100
 1050 ENTER D$ ; D(N)
 1060 NEXT N
 1070 ! Now initiate a new TRANSFER
 1080 TRANSFER 724 TO D$ INTR
 1090 ! Save data on tape. NOTE: this temporarily halts the
      transfer!
 1100 PRINT# 1 ; D() @ ASSIGN# 1 TO *
 1110 ! See if all 10 data files have been saved.
 1120 I=I+1 @ IF I>10 THEN GOTO 1140
 1130 RETURN
 1140 ! If done, then halt I/O and end program.
 1145 ! Note that ABORTIO causes an EOT branch, so disable
      it.
 1150 OFF EOT 7 @ ABORTIO 7
 1160 PRINT "All data saved"
 1170 END
```

The program logic associated with the transfer and the EOT service routine is quite simple (shaded program lines). The extra programming shown is included as one example of how the EOT service might be used for data-save operations to tape. The EOT service routine might just as well have sent the collected data to a remote central computer (with another TRANSFER, of course!).

End-of-transfer service can be cancelled by execution of an OFF EOT statement for the desired select code. For further details about these statements, refer to the alphabetical syntax listing in this manual.

Interactions and Permutations

This section is written so that you may be able to better predict operation of those programs that utilize end-of-line branching.

End-of-line service occurs in a specific order. That is, if more than one end-of-line branch is pending at the end of a program line, one of the branches will be taken before the other. The following table lists the types of end-of-line branches and the select codes, and gives the precedence order for combinations of branch type and select code.

Branch Precedence Table

Branch Type	Select Code 3	4	5	6	7	8	9	10
ON ERROR	1							
ON INTR	2	3	4	5	6	7	8	9
ON TIMEOUT	10	11	12	13	14	15	16	17
ON EOT	18	19	20	21	22	23	24	25
ON KEY	26							
ON TIMER	27							

For example, a pending ON INTR branch for select code 5 would be taken before a pending ON INTR branch for select code 9. *Any* pending ON EOT branch would be taken *after* any pending ON TIMEOUT branch, regardless of select codes.

You should take note that the term *precedence* is used here, not *priority*. This means that when the computer is executing one service routine, other service routines are not implicitly locked out. (In a priority system, any service routine having a lower priority than the routine currently being executed will not receive control until the current routine completes.) If two end-of-line branches are pending on the HP Series 80 computer, the one having precedence is executed first. However, after the first line of that service routine executes, the still-pending end-of-line branch to the second service routine is taken!

The only way the first routine can guarantee uninterrupted execution is to disable the other routine's end-of-line branch with its first line. An example will help show this:

```
 10 ON INTR 7 GOSUB 100
 20 ON INTR 9 GOSUB 200
 30 ENABLE INTR 7;8
 40 ENABLE INTR 9;64
    :
 70 ! Program body goes here
    :
100 OFF INTR 9 ! Now select code 9 cannot get EOL service.
    :
130 ! This routine would do necessary service functions
140 ! for select code 7.
    :
180 ! Now re-enable EOL service for select code 9
190 ON INTR 9 GOSUB 200 @ ENABLE INTR 7;8 @ RETURN
200 OFF INTR 7 ! Now select code 7 cannot get EOL service.
    :
240 ! This routine would do necessary service functions
250 ! for select code 9.
    :
280 ! Now re-enable EOL service for select code 7
290 ON INTR 7 GOSUB 100 @ ENABLE INTR 9;64 @ RETURN
300 END
```

Assume that while line 50 is executing, both interrupts occur. Both end-of-line branches are pending when line 50 completes. Select code 7's service routine has precedence, and so line 100 is executed. Line 100 disables select code 9's service routine, so the end-of-line branch that would have occurred is not taken. Line 190, the last line of the select code 7 service routine, re-enables the select code 9 service routine and when the RETURN is executed, line 200 is then executed. Note that the OFF INTR does not *eliminate* the pending end-of-line branch—it just defers the branch until an ON INTR is executed later.

Events such as interrupts, timeouts, transfer terminations and errors are "remembered" indefinitely, until a RESET or STOP occurs. Therefore, once an event has occurred and remains unserviced, execution of an ON INTR, ON TIMEOUT or ON EOT results in an immediate end-of-line branch.

The *type* of branch taken can affect program operation. The most predictable program execution occurs when GOSUB end-of-line branches are taken. Use GOTO when the program is simple and only one or two end-of-line branches are expected. One very simple example of this would be:

```
ON INTR 7 GOTO 9999          (99999 for the HP-87)
ENABLE INTR 7;128
```

These two statements direct program execution to line 9999 (presumably an END statement) when an Interface Clear on HP-IB occurs. This kind of "bail-out" provision can be useful when developing certain types of programs such as ones using keyboard masking (see section 7). Then, when something goes wrong you can halt the program by asserting IFC on the interface bus with another controller or a bus analyzer.

The following example is presented to show you how some of the ON-event branches interact. The listing and results are shown with a brief explanation, but you will learn more from it if you experiment on your own. Try setting timers, timeouts, and keys up and then watch it work (or not work). Press the special function key while the computer is waiting, then watch the display.

```
 10 ! This program serves to illustrate end-of-line
    branching.
 20 ! Substitute YOUR select codes and device numbers
 30 ! on lines 80 and 90 to make this run successfully.
 40 ! Remember that an ENTER and an input TRANSFER cannot
 50 ! occur simultaneously on the same select code.
 60 !
 70 !
 80 S1=10 ! This is the TRANSFER select code assignment.
 90 S2=705 ! This is the ENTER select code assignment.
100 I=0 @ DIM A$[80],B$[88]
110 IOBUFFER B$
120 !
130 !
140 ON KEY# 1 GOSUB 390 ! Key 1 clears the counter
150 ON KEY# 2 GOSUB 450 ! Key 2 terminates the TRANSFER
160 ON KEY# 3 GOSUB 510 ! Key 3 ends the program
170 ON KEY# 4 GOSUB 770 ! Key 4 starts another ENTER
180 !
190 !
200 ON TIMER# 1,1300 GOSUB 560
210 ON TIMER# 2,1400 GOSUB 610
220 !
230 !
```

```
240 SET TIMEOUT S2;1000 ! Don't forget to set the select
    code!
250 ON TIMEOUT S2 GOSUB 660
260 !
270 !
280 ON EOT S1 GOSUB 710 ! Don't forget to set the select
    code!
290 TRANSFER S1 TO B$ INTR
300 ENTER S2 ; A$
310 !
320 !
330 BEEP 100,20 @ WAIT 500 @ BEEP 20,20
340 DISP "Looping ";I
350 I=I+1 @ GOTO 330
360 !
370 !
380 !
390 ! Key 1 service clears the counter.
400 I=0
410 DISP "Serviced Key 1"
420 RETURN
430 !
440 !
450 ! Key 2 service terminates the active TRANSFER.
460 RESET S1
470 DISP "Terminated TRANSFER"
480 RETURN
490 !
500 !
510 ! Key 3 service ends the program.
520 DISP "End of program"
530 END
540 !
550 !
560 ! Timer 1 service.
570 DISP "Timer 1 serviced."
580 RETURN
590 !
600 !
610 ! Timer 2 service.
620 DISP "Timer 2 serviced."
630 RETURN
640 !
650 !
660 ! Select code S2 ENTER Timeout service
670 DISP "ENTER operation timed out"
680 RETURN
690 !
700 !
710 ! Select code S1 TRANSFER termination service.
720 DISP "End of transfer service"
730 TRANSFER S1 TO B$ INTR
740 RETURN
750 !
760 !
770 ! Key 4 service executes another ENTER operation
780 ENTER S2 ; A$
790 RETURN
800 END
```

Section 7

Keyboard Control

Introduction

In certain applications, the operation of a system could be adversely affected by a curious passerby pressing keys which could halt or alter program execution. In other applications the system operator must have access to certain keys, but not others.

Traditionally, the dedicated computer's keyboard was covered by a "mask" of plastic, metal or wood which covered all keys unnecessary to the operation of the system. Cut-outs were provided to allow access to those keys the operator used to control the system. Obviously, the extra cost of designing and manufacturing such a mask added to the cost of a dedicated system.

The I/O ROM provides the keyboard mask in software, which—short of turning the computer off—offers far less chance of user interference than does a mechanical mask. In addition, greater flexibility is possible with the software masks, as is discussed in the following section.

Key Mask Programming

There are three basic modes of operation for the HP Series 80 Personal Computer: halted (or idle), program execution, and keyboard input.

In the idle mode, the computer accepts commands and is available for program modification/entry. This is the mode that the computer normally is in when not executing a program (when first turned on, or after executing a STOP, PAUSE, or END type statement). The idle mode is generally used for program development or debugging, so a keyboard mask for the idle mode is not very useful and is not provided.

In the program execution mode, pressing any key other than the (TR/NORM) key or an ON KEY defined special function key will halt the program. A keyboard mask can be specified for four classes of keys while in the program execution mode. These classes are:

1. (RESET)
2. (PAUSE)
3. Special function keys and (KEY LABEL) key
4. Other keys (all remaining keys not in classes 1, 2, or 3)

Any or all four classes of keys can be masked out for the program execution mode.

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

1. (RESET)
2. (PAUSE)
3. Special function keys and (KEY LABEL) key
4. Other keys (all remaining keys not in classes 1, 2, or 3)

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

Bit Number	Decimal Value	Operating Mode	Keys Not Masked
7	128	↑	(RESET)
6	64	Program	(PAUSE)
5	32	Execution	Special function keys and (KEY LABEL)
4	16	↓	Other keys
3	8	↑	(RESET)
2	4	Keyboard	(PAUSE)
1	2	Input	Special function keys and (KEY LABEL)
0	1	↓	Other keys

The following example illustrates the use of the keyboard mask to disable all keys except the special function keys for program execution. The (RESET), (PAUSE), and special function keys are masked out for the keyboard input mode. This still allows the operator to respond to an INPUT statement, but program execution cannot be affected.

```
 10 ENABLE KBD 1+32 ! Select allowed keys
 20 REMOTE 706,710 @ LOCAL LOCKOUT 7
 30 ON KEY# 1 GOTO 1000
 40 ON KEY# 2 GOTO 2000
 50 ON KEY# 3 GOTO 3000
    :
100 PRINT "Select operating mode"
    :
1000 DISP "Enter date" @ INPUT M,D,Y
1010 ! and etc.
```

The operator can select the desired program section by pressing the appropriate special function key—but only the special function keys will respond to being pressed. When the operator presses special function key #1, the program branches to line 1000. When the INPUT statement of line 1000 is executed, the special function keys no longer respond but the numeric keys do, allowing the operator to set the time as requested. When the program continues on from the INPUT, once again all keys except special function keys are disabled.

The following example gives the operator a ten second time span in which he can pause or reset the computer if necessary:

```
10 ! Line 50 enables the operator to pause or reset the
20 ! program whenever an INPUT statement is executed.
30 ! Line 100 exists specifically to allow the operator
40 ! to halt the program if necessary.
50 ENABLE KBD 12
60 !
70 !
80 !
```

```
 90 BEEP @ DISP "You have 10 sec to PAUSE or RESET the
    computer"
100 ON TIMER# 1,10000 GOTO 110 @ INPUT X9$
110 OFF TIMER# 1 @ DISP "Computer beginning automatic
    operations"
120 !
130 ! Now the operator cannot halt the program!
140 !
150 !
```

Direct Interface Communication

Introduction

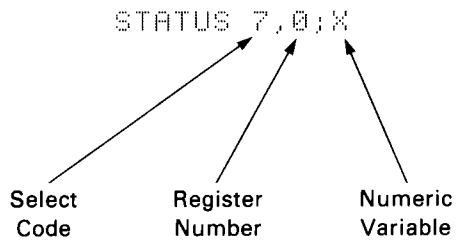
This section deals with the statements available to the programmer for tailoring the operations of an interface to the specific requirements of his system.

The status of interface operations can be monitored with the STATUS statement. This status may reflect the actual hi/low voltage level of external I/O lines or it may indicate the interface's internal state, depending upon which status register is being read.

The mode of operation of an interface can be directed by the CONTROL statement. The interface generally provides automatic control of external I/O lines acording to the mode selected, and also may provide for manual override of certain of the I/O lines for custom sequences.

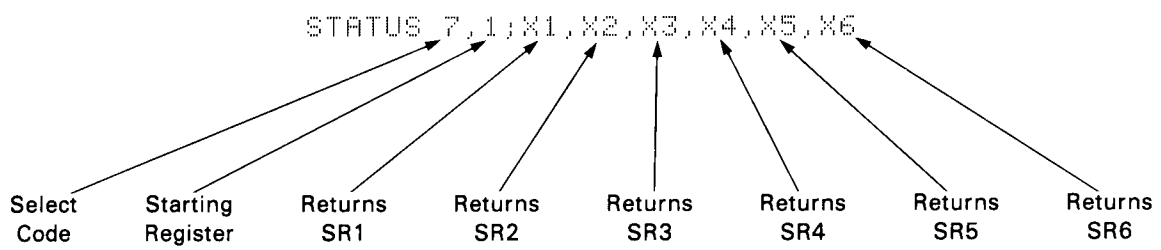
Checking the Status

Interface status (and also buffer status) is monitored by the STATUS statement. The STATUS statement is a very straightforward one to use, requiring only the interface's select code and desired register number to be specified. The following statement obtains the value of the HP-IB interface (select code 7) identification register, SR0:



In this case, the value returned is always 1, the identification code for an HP-IB interface. Each interface returns a different identification code, so consult the appropriate interface owner's manual for specific details.

The values of multiple status registers can be obtained with just one STATUS statement. Simply specify the first register number and give as many variables as necessary to obtain the status needed. For example, to read status registers SR1 through SR6 of the HP-IB interface at select code 7:



What you do with the contents of these registers depends upon your application, needs, and type of interface. The HP-IB interface above, for example, returns

- Interrupt Cause—SR1
- HP-IB Control Lines—SR2
- HP-IB Data Lines—SR3
- HP-IB Address—SR4
- HP-IB State—SR5
- Secondary Command—SR6

for the six registers just read. Obviously the BCD Interface returns different information for those same registers.

Most interfaces are capable of interrupting the computer when a specified condition occurs, and if more than one condition has been enabled it will be necessary to determine which interrupt occurred and caused the end-of-line branch. The following service routine serves as an example of how STATUS would be so used:

```
  10 ON INTR 7 GOSUB 1000
  15 ! Enable interrupts for Active Controller/Talker/
     Listener.
  20 ENABLE INTR 7;112
     :
1000 ! End-of-line branch service for HP-IB interrupts
1010 STATUS 7,1 ; S1 ! Read interrupt cause
1020 IF BIT(S1,4)=1 THEN GOTO 1100 ! Check for Active
     Talker
1030 IF BIT(S1,5)=1 THEN GOTO 1200 ! Check for Active
     Control
1040 IF BIT(S1,6)=1 THEN GOTO 1300 ! Check for Active
Listener
1050 !
1060 ! The following lines provide for unpredictable
     errors.
1070 PRINT "Illegal interrupt condition on HP-IB"
1080 PRINT "HP-IB interrupt cause = ";S1
1090 STOP
1100 ! Service for Active Talker
     :
1200 ! Service for Active Control
     :
1300 ! Service for Active Listener
     :
```

Line 1010 obtains the interrupt cause from SR1 of the interface, and lines 1020-1040 analyze the register bits for the specific cause of the end-of-line branch. Lines 1050-1090 are included to deal with possible program errors or illegal interrupt causes. No program or machine is perfect, so try to make provisions such as the one shown to deal with malfunctions that might occur.

Interface Control

The operating modes of an interface can generally be tailored to suit individual systems and their requirements. This may mean nothing more than selecting the type of interrupt conditions required to deal with events specific to your system. Or, it may mean selecting a handshake mode suitable for the device being connected to the interface. In any case, the CONTROL statement provides the means of programming the interface's control registers.

There are three primary items of interest in a general discussion about CONTROL and interface control registers. The first is the interface interrupt mask. This mask is CR1 for the HP-IB Interface. Therefore, the following two statements have *identical effects* for the HP-IB Interface (select code 7):

```
ENABLE INTR 7;8
CONTROL 7,1;8
```

Both statements enable an SRQ interrupt condition for the interface. They can be used interchangeably, however, the ENABLE INTR statement better documents the effects it will have on program execution.

The second item of interest is that of external I/O lines. In general, you can set or clear interface-specific control lines by writing to control register CR2 of the desired interface (however, consult the individual interface manuals for exact details). Both of the following statements write to HP-IB control register CR2, the HP-IB Control Lines register (select code 7):

```
ASSERT 7;32
CONTROL 7,2;32
```

Both statements set the HP-IB SRQ (Service Request) control line true, but there is a slight difference in the manner that they do so. The CONTROL statement is not immediately executed if there is an I/O operation already in progress (an interrupt transfer, for example). The I/O in progress first completes, then the CONTROL operation is performed. This is in contrast to the ASSERT statement, which *immediately* sets the control line (SRQ in this case) regardless of any I/O in progress. In either case, exercise due caution when writing to the interface control register: the consequences of an improper control operation may be an interface or device malfunction. (In fact, to assert SRQ you should really use the REQUEST statement so that proper bus protocol is ensured.)

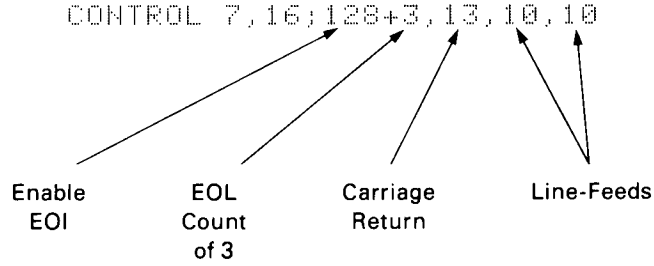
Note: Improper or invalid control line operation may cause loss of data or device malfunctions.

The third item of general interest in control register programming is the end-of-line (EOL) character sequence sent by the interface. This character sequence is essentially an end-of-record delimiter, which in the days of punched cards signaled that an entire card had been read or punched. For all interfaces, this EOL sequence defaults to carriage return/line feed, and is sent after every PRINT or OUTPUT operation (unless inhibited by the programmer). The EOL sequence is also sent for the "/" (slash) image specifier or for the "END" keyword of the SEND statement.

It is sometimes necessary to tailor the EOL sequence to suit the needs of a particular device. For instance, you might need double spacing on a printer (carriage return/double line feed) or carriage-return only for a CRT terminal that performs an automatic line-feed when a carriage-return is received.

The HP-IB Interface provides EOL sequence programmability with control registers CR16-CR23. Up to seven EOL characters can be sent if so desired, and are specified by writing the character values and EOL character count to those control registers. For example, to program a double line-feed EOL sequence with EOI (END or Identify) set on the last line-feed*, the following CONTROL statement could be used:

```
CONTROL 7,16;128+3,13,10,10
```



In summary, the control registers provide the flexibility required to tailor interface operation to your specific system requirements. They should be used with caution, however, so that you don't get any unpleasant surprises. Study the appropriate interface manual to determine which capabilities you need and how to implement them. It may be necessary to experiment a bit before the system works as you want it, but even the pros have to do that.

*You should note that it is not possible to send EOI with the last data byte of an OUTPUT or SEND operation, but it is possible with PRINT, DISP, or TRANSFER.

Section 9

Additional I/O Commands

There are several commands made available by the I/O ROM which have not been introduced in the preceding sections. Many of these commands are only used with certain interfaces and all of them have different meanings with respect to each interface. Several of these interface-specific commands are introduced here to illustrate their capabilities. For extensive examples the interface owner's manuals should be consulted.

Interface-Dependent Statements

The SEND Statement

There are times when it is convenient to be able to send an arbitrary data sequence from the computer to one or more peripherals. The SEND statement accomplishes this directly. The following are examples of properly formed SEND data statements:

```
SEND 705; DATA "welcome to the world of I/O"
SEND 9; DATA "OCT. 9", "NOV. 13"
```

If EOL is specified, the interface sends an end-of-line sequence following each set in the data listing.

With all but the serial interface, the SEND statement may be used to send commands. This allows sending an arbitrary sequence of bytes out over the interface bus, which may be tailored to command any device which requires special sequences for initialization or reconfiguration. These same commands could be sent with OUTPUT statements, of course, but by using the SEND statement judiciously program documentation may be enhanced. The following are examples of correctly formed SEND command statements:

```
SEND 7; CMD B$
SEND 9; MTA MLA UNL LISTEN 4,5 CMD F,H
SEND 4; CMD X$
```

In most cases the interface and device owner's manuals will need to be consulted to confirm the character programming sequences needed.

HALT, ABORTIO, and RESET

The HALT statement is used to terminate any transfer in progress. This is the best means of recovering from a *hung* condition (interface handshake problem). This will not have any effect on a FHS TRANSFER and other means will be needed to recover from that type of condition.

The next most commonly used statement, ABORTIO, will also halt any transfers in progress and will reset control lines on all interfaces and will reset data lines on some interfaces. This is a convenient method of interrupting an I/O operation and ending up in a known state.

RESET is the most drastic statement of this group since it not only terminates any transfers but also returns all configurations selected to their default (switch-selected) state. This should only be necessary when first beginning a program or when recovering from an interface failure.

If an EOT branch is enabled when any of these statements is executed, the branch will be taken. After a HALT or SEND statement, a RESUME is sometimes necessary to re-enable input or output operations. The CLEAR command is used with the GPIO, HP-IB, and HP-IL interfaces to reset peripherals. The interface itself is not reset. Examples of these statements are given below:

```
HALT 7
ABORTIO 7
RESET 7
CLEAR 705
RESUME 7
```

Bus-Controlling Functions

With the instrumentation-controlling interfaces (HP-IB and HP-IL) the REMOTE and LOCAL statements can be used to allow or not allow access to device settings by use of the front-panel controls supplied on the device.

The REMOTE statement places instruments under remote control. The instrument may be returned to front panel control by executing a LOCAL statement or by activating a switch on the instrument called *Return to Local* or *Manual Control.*

If system security requires, a LOCAL LOCKOUT statement may be executed which precludes any local control whatsoever until a LOCAL statement is subsequently executed.

For an example of the LOCAL statement, see the program illustrating the Interface Interrupts section of section 6.

These statements will not be covered in this manual since they have specific meanings (quite specific!) with each interface they are to be used with:

```
REQUEST      SPOLL      PPOLL      PASS CONTROL      TRIGGER
```

Refer to the syntax reference in the first appendix of this manual for a brief summary of the uses of these statements with each interface. The interface owner's manuals will contain more specific instructions and examples.

Section 10

Binary Functions

Introduction

I/O programming often involves the use of binary functions and bit-level operations. The I/O ROM provides several useful tools to assist you with these tasks. These binary functions are often used when manipulating status and control registers in the interface cards. They may also be very handy for processing I/O data in device control applications. This section explains the binary functions available with the I/O ROM.

It is important to remember that all the binary functions provided by the I/O ROM operate on 16-bit words. For example, the binary complement of zero is 11111111 11111111 (base 2); the range of bits that can be tested is 0 thru 15; and the range of values for binary arguments is −32 768 thru 32 767. There is no problem using these functions to operate on binary values with less than 16 bits. The unused high-order bits are simply assumed to be zero.

In the following explanations, the term *integer* is used frequently to identify the arguments for many of the functions. To the I/O ROM, an integer is a 16-bit binary number with a range of −32,768 thru 32,767. This contrasts to the definition of an integer given in BASIC where an integer is a 5-digit number with a range of −99,999 thru 99,999. Please keep this distinction in mind to avoid confusion about the term "integer".

If you are not familiar with the binary numbers and operators, study the following review before continuing with this section.

Review of Base 2

Before looking at base 2, it is helpful to take a careful look at the familiar base 10. The number one hundred twenty five is represented as follows:

125

The digits have a *place value* corresponding to powers of ten. The representation above really means:

$$1 \times 10^2 + 2 \times 10^1 + 5 \times 10^0$$

The concept of place value also exists in base 2. The difference being that powers of two are represented instead of powers of ten. The number one hundred twenty five is represented as:

11111101

Base 2 uses only the digits "1" and "0"; a 1 indicates that a place value is included, while 0 indicates that a place value is not used in the value. Therefore, the binary representation shown above means:

$$2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0$$

This is the same as:

$$64 + 32 + 16 + 8 + 4 + 1$$

The term *bit* comes from the words "binary digit". A bit is a single digit in base 2 that must be either a 1 or a 0. The grouping of 8 bits together is in such common usage for character representation, internal storage, and interfacing that it has been given a special name—a **byte**. The term byte refers to 8 bits processed as a unit.

Notice in both the previous examples that the right-most digit represents the "0th" power of the base. Because of this, bit patterns are usually numbered starting at Bit 0, instead of Bit 1. By doing this, the bit number and the power of two it represents are the same. The following table shows the bit positions in a byte and their corresponding values.

Bit Position	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Meaning	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Value	128	64	32	16	8	4	2	1

Examples: 130 in base 10 is 10000010 in base 2
3 in base 10 is 00000011 in base 2
25 in base 10 is 00011001 in base 2

The term *word* is also widely used in computer appliations. A "word" is usually the number of bits that can be handled in one operation by the internal architecture of the computer. Although the Series 80 Personal Computers have an 8-bit internal architecture, they also have operations defined for 16-bit and floating point numbers. Therefore, 16-bit integers are often referred to as words in the computer because the system can handle them as a basic data unit. Another characteristic of a word in the computer is that *2's complement* representation is used. 2's complement representation is a method of storing either positive or negative numbers in a word. It works like this:

Positive numbers: If bit 15 is 0, then the word is a positive number represented in normal binary form.

Negative numbers: If bit 15 is 1, then the word is a negative number represented in 2's complement form. To find the absolute value of a negative number, invert all the bits and add 1.

Problem: What is the value of 11111111 11110000?

Solution: Bit 15 tells that this is a negative number.
Inverting all the bits gives: 00000000 00001111
Adding 1 results in: 00000000 00010000
So the value of the given bit pattern is −16.

Review of Logical Operations

In this discussion, "logical operations" refers to operations from Boolean algebra, such as AND and OR. The outstanding I/O feature of these operations is that they can modify individual bits without affecting surrounding bits. In this respect, they can be contrasted to the arithmetic operations, such as addition and subtraction. Addition and subtraction generate carries and borrows that can propagate through an entire word, changing the state of numerous bits many places away from the bit where the arithmetic was performed. Although this is exactly what is desired for numerical quantities, many of the bytes and words used in I/O are not numerical quantities. When bits are used as individual control elements, the programmer must have access to tools that allow individual control of bits. This section reviews the action of the common logical operators.

The operators AND, OR, NOT, and EXOR are available in the standard language of the HP Series 80 computers. These operators treat an entire variable as one entity. A value of "0" is considered "false", while any other value is considered "true". Although these operators perform the same Boolean function as the binary logical operators, they do not operate on individual bits. The binary logical operators available with the I/O ROM work on a bit-by-bit basis across an entire word. Without these binary tools, isolating an individual bit requires an involved combination of tests, branches, and arithmetic operators.

The simplest logical operation is the *complement* operation. When binary data is complemented, all the 1's are changed to 0's, and all the 0's are changed to 1's. This operation is also known as "1's complement", or "inversion". The Boolean notation for this operation is a horizontal bar drawn over the variable. The truth table is as follows:

A	$\overline{A}$
0	1
1	0

When used on an entire byte, the complement operator inverts each bit individually.

Binary value of A: 10011101
Binary complement of A: 01100010

The other logical operators combine two inputs to create a result. Let's look at the AND operator first. A binary AND produces a "1" in the result only if both inputs are "1". The Boolean notation for this operation is ^, although you may also see the symbol · used. The truth table for AND is as follows:

A	B	A ^ B
0	0	0
0	1	0
1	0	0
1	1	1

The important thing to notice is that the result is 0 when A is 0, while the result is equal to B when A is 1. Because of this, a binary AND is a convenient method for clearing selected bits. For example, assume that you wanted to clear the two lowest bits in a byte without disturbing the other bits. This can be done by ANDing the byte with an appropriate mask.

Original bute: 10011101
Byte ANDed: 11111100
Result: 10011100

This operation not only preserves the state of the top six bits, but also clears the bottom two bits no matter what their original state. That saves alot of testing and branching.

The next operator is the binary OR, most correctly called the *inclusive OR*. In English this means: you can have pie OR ice cream for dessert, and it is possible to have both at the same time. To the computer this means that the result bit is "1" when either input bit is "1". The Boolean notation for an inclusive OR is v, although you may also see the symbol + used. The inclusive OR truth table is:

A	B	A v B
0	0	0
0	1	1
1	0	1
1	1	1

The important thing to notice is that the result is 1 when A is 1, while the result equals B when A is 0. Because of this, the inclusive OR is a convenient method for setting selected bits. For example, assume you wanted to set the two lowest bits in a byte without disturbing the other bits. This can be done by ORing the byte with an appropriate mask.

Original byte: 10011101
Byte ORed: 00000011
Result: 10011111

This operation sets the lower two bits no matter what their original state and also preserves the state of the top six bits.

The final operator is the binary EXOR, or *exclusive OR*. In English this means: you can take the plane OR the train to Chicago, but you can't do both at the same time. To the computer this means that the result bit is "1" if a single input bit is "1", but the result bit is "0" if both input bits are the same. The Boolean notation for an exclusive OR is v, although you may see the symbol + used. The exclusive OR truth table is:

A	B	A v B
0	0	0
0	1	1
1	0	1
1	1	0

This important thing to notice is that the result is equal to B when A is 0, while the result is the complement of B when A is 1. Because of this, the exclusive OR is a convenient method for inverting selected bits. For example, assume that you wanted to invert the lower two bits of a byte without disturbing the rest of the bits.

Original byte: 10011110
Byte EXORed: 00000011
Result: 10011101

This operation complements the lower two bits no matter what their original value and leaves the top six bits unchanged.

The Binary AND Function

The binary AND function performs a bit-by-bit AND using two integers as arguments and producing an integer result. Here are some examples of properly formed binary AND functions:

```
X=BINAND(Y,15)
A=C+BINAND(D,E)
PRINT BINAND(255,Z-32)
```

Notice that the arguments must be enclosed in parentheses and separated by a comma. The arguments may be numeric constants, numeric variables, numeric expressions, or any combination. The arguments are assumed to be in base 10 representation. If you wish to express the arguments in another base, refer to section 11. Each bit of the result is computed according to the following truth table:

First Argument	Second Argument	Function Result
0	0	0
0	1	0
1	0	0
1	1	1

The Binary Inclusive OR Function

The binary inclusive OR function performs a bit-by-bit inclusive OR using two integers as arguments and producing an integer result. Here are some examples of properly written binary inclusive OR functions:

```
X=BINIOR(Y,15)
A=C+BINIOR(D,E)
PRINT BINIOR(255,Z-32)
```

Notice that the arguments must be enclosed in parentheses and separated by a comma. The arguments may be numeric constants, numeric variables, numeric expressions, or any combination. The arguments are assumed to be in base 10 representation. If you wish to express the arguments in another base, refer to section 11. Each bit of the result is computed according to the following truth table:

First Argument	Second Argument	Function Result
0	0	0
0	1	1
1	0	1
1	1	1

The Binary Exclusive OR Function

The binary exclusive OR function performs a bit-by-bit exclusive OR using two integers as arguments and producing an integer result. Here are some examples of correctly stated binary exclusive OR functions:

```
X=BINEOR(Y,15)
A=C+BINEOR(D,E)
PRINT BINEOR(255,Z-32)
```

Notice that the arguments must be enclosed in parentheses and separated by a comma. The arguments may be numeric constants, numeric variables, numeric expressions, or any combination. The arguments are assumed to be in base 10 representation. If you wish to express the arguments in another base, refer to section 11. Each bit of the result is computed according to the following truth table:

First Argument	Second Argument	Function Result
0	0	0
0	1	1
1	0	1
1	1	0

The Binary Complement Function

The binary complement funcion performs a bit-by-bit complement of an integer argument, producing an integer result. Here are some examples of properly formed binary complement functions:

```
A=BINCMP(B)
X=BINCMP(Y-Z)
PRINT BINCMP(N*8)
```

Notice that the argument is enclosed in parentheses. The argument may be a numeric constant, a numeric variable, a numeric expression, or any combination. The argument is assumed to be in base 10 representation. Each bit of the result is computed according to the following truth table:

Argument	Result
0	1
1	0

You should keep in mind that the binary complement function operates on a full 16-bit word. This may, in some cases, give an unexpected result if you are dealing exclusively with 8-bit bytes. The 16-bit complement of an 8-bit byte is always a negative number. You can generate 8-bit complements by using the binary exclusive OR function. An exclusive OR with 255 complements the lower eight bits and leaves the upper eight bits as zeros. This technique prevents the unintentional generation of negative values when dealing with single bytes.

The Bit Test Function

The bit test function is used to indicate whether a specific bit in an integer is set (1) or clear (0). The general form for the bit test instruction is:

BIT(*integer , bit number*)

The integer argument must be the first expression and the two expressions must be separated by a comma. The bit number must be in the range 0 thru 15, where 0 is the least-significant bit and 15 is the most-significant bit. Either argument may be a numeric constant, a numeric variable, a numeric expression, or any combination. Here are some examples of properly stated bit test functions:

```
A=BIT(B,3)
X=BIT(Y,Z-1)
IF BIT(N,1) THEN GOSUB 220
```

The bit test function is very useful in decision making and branching. It is easily used with the IF statement to direct program flow based on the state of individual bits. The function returns a 0 (false) if the specified bit is 0 and returns 1 (true) if the specified bit is 1.

Section 11

Base Conversion Functions

Introduction

A programmer who works at the bit and byte level soon develops a preference for the base in which bytes and words are represented. In some cases, base 2 offers the clearest display of a bit pattern. Base 8 had a large following in the days when computers could not easily handle alphabetic characters as numeric input. Base 16 has gained much popularity in recent years because most computers use a word length that is an integral multiple of 4, and modern systems have no trouble converting the symbols A thru F used in base 16, also known as *hexadecimal* or simply *hex*.

To accommodate these various preferences, your computer provides conversion functions that allow theinput and output of integers using any of the alternate representations mentioned above. The base conversion functions have certain characteristics in common:

- All conversions go from base 10 to an alternate base or from an alternate base to base 10. You can't convert directly from one alternate base to another without passing through base 10.
- The base 10 side of the conversion is always a numeric quantity, while the alternate base side is always a string.
- Because the alternate base representations are string data, they can be input, output, compared, stored, and manipulated to some degree. However, the string representations cannot be used in arithmetic operations.
- All arguments for the base conversion functions must be in the range of 16-bit integers. This includes the alternate representations as well as the base 10 values.

If you are not familiar with alternate number bases, read the following review material.

Review of Alternate Representations

When text contains values represented in more than one base, it is extremely important to distinguish between the concepts of *value* and *representation*. Consider the number one hundred. The value is the number of beans in a jar of one hundred beans. The representation in base 10 is the digit "1" followed by two zeros. The value one hundred can also be represented as "64" in base 16, as "01100100" in base 2, and as "10" in base 100.

The representation of a number is merely the character set used to communicate the number's value. Numbers are often represented in bases other than 10 when the use of an alternate base more clearly communicates the number's value. For example, suppose that up to 16 small pumps and valves are controlled by a single 16-bit word from the computer. If the control pattern were represented in base 10, it could be very difficult to understand the number in terms of pumps and valves. However, suppose the number is represented in base 2, further defined so that the most-significant byte is pumps and the least-significant byte is valves. The base 2 representation "00100000 10000000" clearly shows one pump and one valve open. That same value is "8320" in base 10. How quickly does "8320" communicate to you that one pump is on and one valve is open?

The problem with using base 10 to represent a binary number is that one base 10 digit does not represent an integral number of bits. A base 10 pattern does not readily reflect which bits are "1" and which are "0". The problem with using base 2 is that there are simply too many characters to read and write. To circumvent these problems, persons who work at the bit and byte level in computers commonly use base 8 or base 16 to represent binary numbers. These bases have place values directly related to powers of two, making it easy to trace bits with a little practice. They also provide representations that are three and four times more compact than binary, reducing the number of characters needed to a more manageable small group. For example, an entire byte is never more than 2 characters in base 16.

Base 8, known as *octal*, uses one octal digit for three binary digits. Base 16, known as *hex*, uses one hex digit for four binary digits. The following tables show the decimal (base 10), binary (base 2), octal (base 8), and hex (base 16) representations for the numbers 0 thru 16.

Decimal	Binary	Octal	Hex
0	000	0	0
1	001	1	1
2	010	2	2
3	011	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1 000	10	8
9	1 001	11	9
10	1 010	12	A
11	1 011	13	B
12	1 100	14	C
13	1 101	15	D
14	1 110	16	E
15	1 111	17	F
16	10 000	20	10

Notice that the Arabic numeral system (designed for base 10) does not have any single-character symbols to represent quantities above nine. Single-character representation of all quantities less than the base value is essential to the concept of place value. Therefore, base 16 representation utilizes the characters A thru F to represent values from 10 thru 16. The following examples help to illustrate the way that octal and hex numbers use bit groupings to represent binary values.

Octal:	1 2 5	3 6 0	0 7 0	2 0 1	2 0 2
Binary:	01010101	11110000	00111000	10000001	10000010
Hex:	5 5	F 0	3 8	8 1	8 2

Conversions From Base 10 to an Alternate Base

These functions use a base 10 numeric quantity as an argument and produce a string as a result. The primary use of these functions is the printing, display, or output data in an alternate base, although other applications are possible. The argument for the function may be a numeric constant, numeric variable, numeric expression, or any combination. The argument must be in the range of −32 768 thru 32 767. Functions are available to convert to base 2, base 8, or base 16.

From Base 10 to Base 2

This is the **Decimal To Binary String** function. It converts an integer argument to a string of 16 ones and zeros. The string is the base 2 representation of the integer argument. If the argument is out of range and positive, the function yields "0111111111111111". If the argument is out of range and negative, the function yields "1000000000000000". The following are examples of well-stated conversion expressions:

```
PRINT DTB$(X)
A$=DTB$(N$2)
OUTPUT 701;DTB$(32+Y)
DISP DTB$(255)
```

From Base 10 to Base 8

This is the **Decimal To Octal String** function. It converts an integer argument to a 6-character string. The string is the octal representation of the integer argument. If the argument is out of range and positive, the function yields "077777". If the argument is out of range and negative, the function yields "100000". The following are examples of properly formed decimal-to-octal expressions:

```
PRINT DTO$(X)
A$=DTO$(N$2)
OUTPUT 701;DTO$(32-Y)
DISP DTO$(255)
```

From Base 10 to Base 16

This is the **Decimal To Hex String** function. It converts an integer argument to a 4-character string. The string is the hex (hexadecimal) representation of the integer argument. If the argument is out of range and positive, the function yields "7FFF". If the function is out of range and negative, the function yields "8000". The following are examples of correct decimal-to-hex expressions:

```
PRINT DTH$(X)
A$=DTH$(N$2)
OUTPUT 701;DTH$(32-Y)
DISP DTH$(255)
```

Conversions From an Alternate Base to Base 10

These functions use a string as an argument and produce a numeric result. The primary use of these functions is the input of data in an alternate base, although other applications are possible. The argument for the function may be a string constant (literal), string variable, string expression, or any combination. The argument must represent a value in the range of 16-bit integers. Functions are available to convert from base 2, base 8, or base 16.

From Base 2 to Base 10

This is the **Binary To Decimal** function. The argument is a string which is the binary representation of an integer. The argument cannot have more than 16 characters, and only the numerals "1" and "0" are valid. The result of the function is the base 10 value of the number represented by the argument. Since the function result is numeric, it can be used in arithmetic operations or numeric functions. The following are examples of properly formed expressions:

```
PRINT BTD("100101")
X=BTD(A$)
Y=255-BTD(N$)
```

From Base 8 to Base 10

This is the **Octal To Decimal** function. The argument is a string which is the octal representation of an integer. The argument cannot have more than 6 characters. Only the numerals "0" thru "7" are valid. If all six characters are used, the most-significant character can only be a "1" or a "0". The result of the function is the base 10 value of the number represented by the argument. Since the function result is numeric, it can be used in arithmetic operations or numeric functions. The following are examples of correctly stated expressions:

```
PRINT OTD("371")
X=OTD(A$)
Y=255-OTD(N$)
```

From Base 16 to Base 10

This is the **Hex To Decimal** function. The argument is a string which is the hex representation of an integer. The argument cannot have more than four characters. Only the numerals "0" thru "9" and the letters "A" thru "F" are valid. The result of the function is the base 10 value of the number represented by the argument. Since the function result is numeric, it can be used in arithmetic operations or numeric functions. The following are examples of properly written hex-to-decimal expressions:

```
PRINT HTD("1F4")
X=HTD(A$)
Y=255-HTD(N$)
```

Converting From One Alternate Base to Another

Conversions between alternate representations are easily done by nesting two conversion functions. The following short program is an example of this technique. It inputs a hex representation and displays the corresponding binary representation.

```
10 DISP "Hex Number";
20 INPUT A$
30 DISP "Binary = ";DTB$(HTD(A$))
40 GOTO 10
```

Appendix A

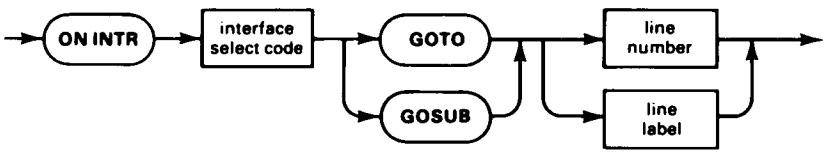
Syntax Reference

Conventions Used to Represent Syntax

This reference section uses two methods of representing the syntax of I/O ROM statements. The conventions of each form are as follows.

Pictorial Representation

All items enclosed by a rounded envelope must be entered exactly as shown. Items enclosed by a rectangular box are names of parameters used in the statement. A description of each parameter is given in the text following the drawing. Statement elements are connected by lines. Each line can only be followed in one direction, as indicated by the arrow at the end of the line. Any combination of statement elements that can be generated by following the lines in the proper direction is syntactically correct. A statement element is optional if there is a valid path around it.



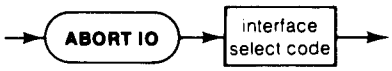
This form of syntax representation is easy to use, and in some cases, more formally correct than the alternate form described next.

Linear Representation

This form of syntax representation is included to be compatible with previous HP Series 80 manuals. Many users are accustomed to seeing this form.

DOT MATRIX	All items shown in dot matrix must be entered exactly as shown.
[ ]	Items within square brackets are optional.
item 1 *item 2*	A vertical placement of two items indicates that only one of the items may be included.
...	Three dots indicate that successive parameters are allowed.
italic	Italicized items are the parameters themselves.

ABORTIO



ABORTIO *interface select code*

Example Statements

```
100 ABORTIO 7
250 IF S<128 THEN ABORTIO S0
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

Actions Taken

All interfaces: Terminates any interrupt transfer in progress. Performing an ABORTIO on an interface with an active transfer and EOT branching enabled causes the branch to be taken.

HP-IB:

- System Controller: Sends Interface Clear (IFC) and Remote Enable (REN).
- Active Controller (but not System Controller): Sends Attention (ATN) and My Talk Address (MTA).
- Non controller: Stops handshaking data and becomes ready for next operation.

Serial: Turns off all modem control lines (control register 2).

BCD: Stops handshaking data, sets CTL line false, and places external data lines in high-impedance state.

GPIO: Stops handshaking data, sets control lines false, places ports A and B in high-impedance state, and sets lines from ports C and D to false state.

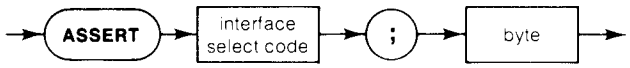
HP-IL:

- System or Active Controller: Sends Interface Clear.
- Non controller: Stops current operation and becomes ready for next operation.

Related Statements

```
HALT
ON EOT
RESET
```

ASSERT



ASSERT *interface select code* ; *byte*

Example Statements

```
100 ASSERT 7 ; 12
210 IF A1=128 THEN ASSERT S ; X
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

byte—a numeric expression that evaluates to an integer 0 thru 255. Binary value of the byte is used to set or clear the lines to be asserted.

Actions Taken

HP-IB: Immediately writes the value of the byte to control register 2. IFC bit (2^7, decimal 128) is ignored (use ABORTIO).

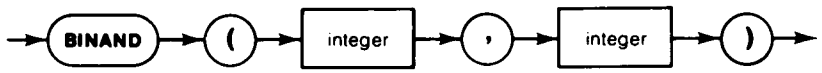
Serial, BCD, GPIO: Immediately writes the value of the byte to control register 2.

HP-IL: Sends a frame using the specified byte and the most recent control bits written to register 2. This statement is similar to CONTROL *isc* , 3 ; *byte* except that ASSERT interrupts the interface and sends the frame without checking loop status.

Related Statements

ABORTIO
CONTROL

BINAND



BINAND (*integer*, *integer*)

Example Statements

```
10 B1=BINAND(X1,15)
100 PRINT BINAND(I,N*2^3)
```

Parameters

integer—a numeric expression that evaluates to an integer −32,768 thru 32,767.

Action Taken

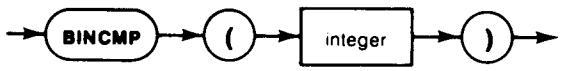
BINAND is a function that returns the 16-bit binary AND of two integer values. Each bit of the result is calculated using the corresponding bit of each argument, according to the following truth table:

Arg. 1	Arg. 2	Result
0	0	0
0	1	0
1	0	0
1	1	1

Related Statements

BINCMP
BINEOR
BINIOR
BIT

BINCMP




```
BINCMP (integer)
```

Example Statements

```
100 C=BINCMP(X1)
120 PRINT BINCMP(N*2^3)
```

Parameters

integer—a numeric expression that evaluates to an integer −32,768 thru 32,767.

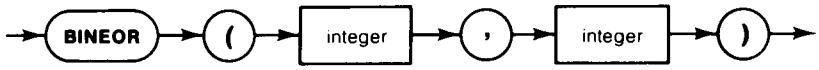
Action Taken

BINCMP is a function that returns the 16-bit binary complement of an integer value. 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.

Related Statements

```
BINAND
BINEOR
BINIOR
BIT
```

BINEOR



BINEOR (*integer*, *integer*)

Example Statements

```
20 B1=BINEOR(X1,15)
140 PRINT BINEOR(I,2^N)
```

Parameters

integer—a numeric expression that evaluates to an integer −32,768 thru 32,767.

Action Taken

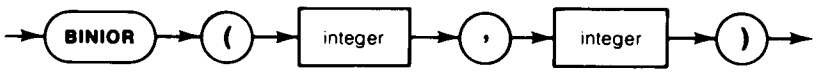
BINEOR is a function that returns the 16-bit binary exclusive OR of two integer values. Each bit of the result is calculated using the corresponding bit of each argument, according to the following truth table:

Arg. 1	Arg. 2	Result
0	0	0
0	1	1
1	0	1
1	1	0

Related Statements

BINAND
BINCMP
BINIOR
BIT

BINIOR



BINIOR (*integer*, *integer*)

Example Statements

```
30 Y=BINIOR(X1,255)
160 DISP BINIOR(I,2^N)
```

Parameters

integer—a numeric expression that evaluates to an integer −32,768 thru 32,767.

Action Taken

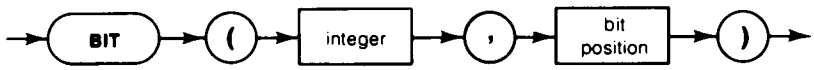
BINIOR is a function that returns the 16-bit binary inclusive OR of two integer values. Each bit of the result is calculated using the corresponding bit of each argument, according to the following truth table:

Arg. 1	Arg. 2	Result
0	0	0
0	1	1
1	0	1
1	1	1

Related Statements

BINAND
BINCMP
BINEOR
BIT

BIT



BIT (*integer , bit position*)

Example Statements

```
40 Y=BIT(X3,7)
180 IF BIT(N,2^I) THEN GOTO 220
```

Parameters

integer—a numeric expression that evaluates to an integer −32,768 thru 32,767.

bit position—a numeric expression that evaluates to an integer 0 thru 15. Least-significant bit is in position 0, most-significant in position 15.

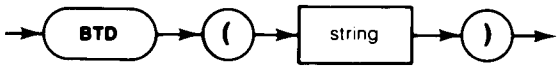
Action Taken

BIT is a function that returns the value of one bit in an integer argument. Result of the function is TRUE if bit is set, FALSE if bi is clear.

Related Statements

BINAND
BINCMP
BINEOR
BINIOR

BTD



BTD (*string*)

Example Statements

```
20 X=BTD(H$&L$)+A1
130 DISP BTD("11000001")
```

Parameters

string—a string expression that contains the base 2 representation of an integer. Limited to 16 significant characters that must be "1" or "0".

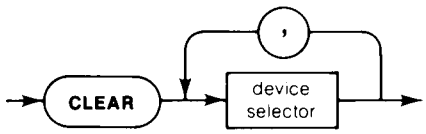
Action Taken

BTD is a function that returns the value of a base 2 representation contained in the string argument. The argument is a character representation and the result is a numeric quantity.

Related Statements

DTB$
DTH$
DTO$
HTD
OTD

CLEAR



CLEAR *device selector* [, *device selector*] ...

Example Statements

```
60 CLEAR 3
250 CLEAR S*100+D1,S*100+D2
```

Parameters

device selector—a valid interface select code or a valid combination of interface select code and primary address (see Choosing the Source or Destination in section 1). If multiple device selectors are specified, they must all be on the same interface select code.

Actions Taken

HP-IB and HP-IL: Must be Active Controller. (HP-IB leaves ATN true; use RESUME if you wish to set ATN false.)

- If device selector is only an interface select code, then Device Clear (DCL) is sent.
- If device selector contains a primary address, then Unlisten (UNL), Listen Address(es) (LAD), and Selected Device Clear (SDC) are sent.

Serial, BCD: Error

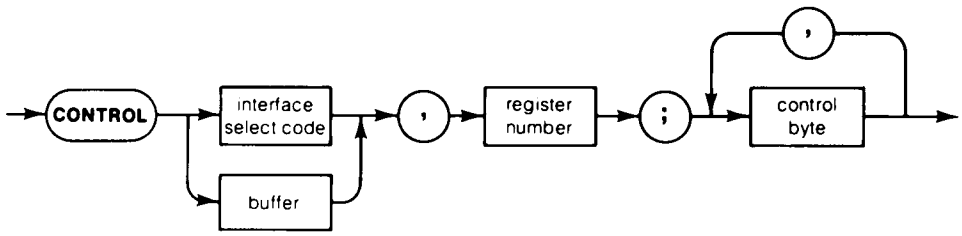
GPIO:

- If device selector is only an interface select code, interface pulses RESA and RESB.
- If device selector contains an even primary address, interface pulses RESA.
- If device selector contains an odd primary address, interface pulses RESB.

Related Statements

CONTROL
SEND

CONTROL



CONTROL *buffer / interface select code* , *register number* ; *control byte* [, *control byte*] ...

Example Statements

```
10 CONTROL 9,1 ; C1,C2,C3,C4,C5,C6
50 CONTROL S,R ; C(R)
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

buffer—the name of a string variable that has been declared an IOBUFFER.

register number—a numeric expression that evaluates to an integer 0 thru 23. Must specify a valid control register for the selected interface.

control byte—a numeric expression that evaluates to an integer 0 thru 255. Binary value of byte is used to set and clear bits in the control register.

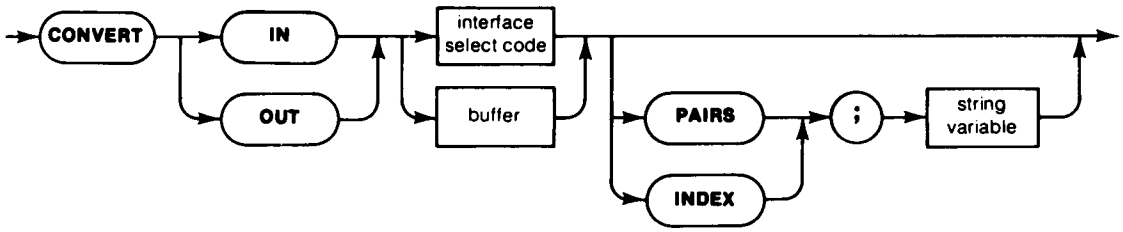
Action Taken

CONTROL writes one or more control bytes to interface or buffer control registers. The register number specifies the first register to be used. If multiple control bytes are specified, they are stored in consecutive control registers, beginning with the specified register number.

Related Statements

ABORTIO
ASSERT
ENABLE INTR
IOBUFFER
STATUS

CONVERT




```
CONVERT  IN  buffer  [PAIRS ; string variable]
         OUT interface select code  INDEX
```

Example Statements

```
20 CONVERT OUT 4 PAIRS ; A$
50 CONVERT IN 10 INDEX ; C$
110 CONVERT IN 7 ! Turn off conversion
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

buffer—the name of a string variable that has been declared an IOBUFFER.

string variable—the name of a string variable in which the conversion table has been previously stored.

Actions Taken

Enables or disables a character conversion process for a specified interface or buffer and a specified direction. Although you can CONVERT using either an interface or a buffer, conversions can only be performed with OUTPUT and ENTER statements. Conversions are **not** performed during SEND or TRANSFER.

If the optional parameters are not included (as in the 3rd example statement), a previously selected conversion is turned off for the specified interface and direction.

If direction is specified as IN, all bytes being input from the specified source are processed through a conversion table immediately after they are received from the source. If direction is specified as OUT, all bytes being output to the specified destination are processed through a conversion table immediately before they are sent to the destination. IN and OUT conversions may both be specified for a given interface select code or buffer.

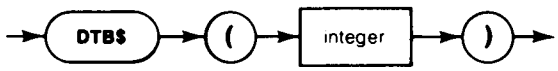
If conversion method PAIRS is specified, the conversion table is treated as a sequential list of character pairs, the second character in each pair being substituted for the first character. If the byte to be converted is not found as one of the first characters in a pair, it is passed through unchanged. Recommended when only a few characters need to be converted.

If the conversion method INDEX is specified, the numeric value of the byte to be converted is used as an index into the conversion table. The byte found as a result of this indexed lookup is substituted for the original byte. If the index value is greater than the length of the table, no conversion is performed. The first character in the string corresponds to the index value of 0. Recommended when a large number of characters need to be converted.

Related Statements

ENTER
IOBUFFER
OUTPUT

DTB$



DTB$(*integer*)

Example Statements

```
100 A$=DTB$(16+2*N)
200 PRINT DTB$(X1)
```

Parameters

integer—a numeric expression that evaluates to an integer −32,768 thru 32,767.

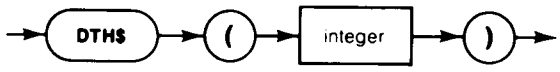
Action Taken

DTB$ is a function that returns the base 2 representation of an integer argument. The result is a 16-character string and the argument is a numeric quantity.

Related Statements

BTD
DTH$
DTO$
HTD
OTD

DTH$



DTH$(*integer*)

Example Statements

```
110 B$=DTH$(32+2^N)
210 PRINT DTH$(X2)
```

Parameters

integer—a numeric expression that evaluates to an integer −32,768 thru 32,767.

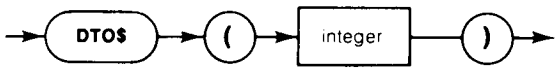
Action Taken

DTH$ is a function that returns the base 16 representation of an integer argument. The result is a 4-character string and the argument is a numeric quantity.

Related Statements

BTD
DTB$
DTO$
HTD
OTD

DTO$



DTO$(*integer*)

Example Statements

```
120 C$=DTO$(64+2^N)
220 PRINT DTO$(X3)
```

Parameters

integer—a numeric expression that evaluates to an integer −32,768 thru 32,767.

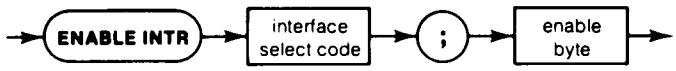
Action Taken

DTO$ is a function that returns the base 8 representation of an integer argument. The result is a 6-character string and the argument is a numeric quantity.

Related Statements

BTD
HTD
OTD
DTB$
DTH$

ENABLE INTR




```
ENABLE INTR interface select code ; enable byte
```

Example Statements

```
10 ENABLE INTR 7 ; 8 ! SRQ interrupt, HP-IB
50 IF S>2 AND S<11 THEN ENABLE INTR S ; X
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

enable byte—a numeric expression that evaluates to an integer 0 thru 255. Binary value of bytes is used to set and clear bits in the control register.

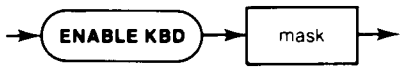
Action Taken

Enables the specified interface for interrupts according to the bits set in the enable byte. The enable byte is placed in control register CR1. The meaning of each bit in CR1 is interface dependent; refer to the appropriate interface owner's manual for details. This statement is identical to performing a CONTROL statement to control register 1.

Related Statements

```
CONTROL
ON INTR
STATUS
```

ENABLE KBD




```
ENABLE KBD mask
```

Example Statements

```
30 ENABLE KBD 33
180 IF X THEN ENABLE KBD K1
```

Parameters

mask—a numeric expression that evaluates to an integer 0 thru 255. Binary value of byte determines which keyboard modes are enabled and disabled.

Action Taken

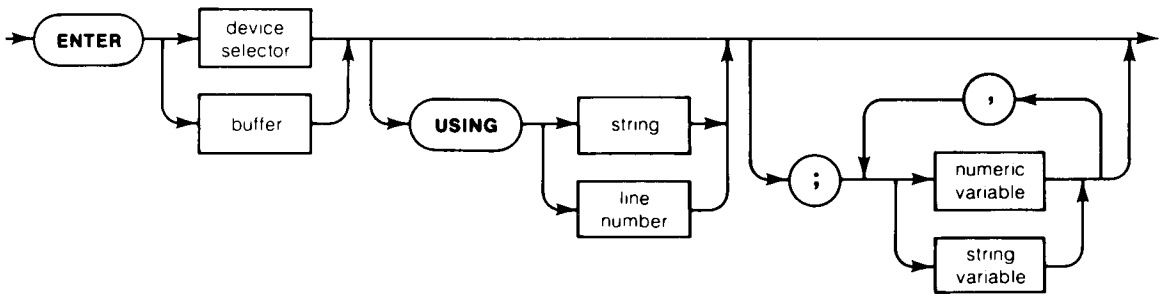
Bits in the mask byte correspond to various keyboard areas and program modes as shown in the following table. If a bit is set in the mask, its feature is enabled. If a bit is clear, its feature is disabled.

Bit	Mode	Keys Affected
7	RUN	(RESET)
6	RUN	(PAUSE)
5	RUN	SFKs and (KEY LABEL)
4	RUN	All other keys
3	INPUT	(RESET)
2	INPUT	(PAUSE)
1	INPUT	SFKs and (KEY LABEL)
0	INPUT	All other keys

Related Statements

```
INPUT
ON KEY#
```

ENTER



ENTER *buffer / device selector* [USING *string / line number / line label**][; [*variable*][, *variable*] ...]

Example Statements

```
70 ENTER 701 USING LBL: ; X,Y,Z
90 ENTER C$ ; N(I),Z$
120 ENTER 3 USING 30 ; A$
250 ENTER 100*S+A USING "#,B" ; N
```

Parameters

device selector—a valid interface select code or a valid combination of interface select code and primary address (see Choosing the Source or Destination in section 1).

buffer—the name of a string variable that has been declared as an IOBUFFER.

string—a string expression that contains a valid set of image specifiers.

line number/label—the line number or label or 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.

Action Taken

Inputs bytes from the specified buffer or device; uses those bytes to build a number or string; places the result into a BASIC variable. If a CONVERT is in effect, the conversion occurs immediately after the character is taken from the interface or buffer.

---

*Line labels are allowed on the HP-87 but not on the HP-85 or 83.

When USING is not specified, free-field format is used. A free-field entry into a string places incoming bytes into the variable until either a line feed is received, a carriage return/line feed sequence 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 up to 256 leading non-numeric characters. Blanks are ignored during number building. Entry into a numeric variable is terminated by the first trailing character that is non-blank and non-numeric.

When USING is specified, input operations are formatted according to the image specifiers used. Image specifiers may be enclosed in quotes 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 the IMAGE statement in this appendix or see Formatted ENTER in section 3.

ENTER requires a line-feed character to satisfy the statement after the variable list has been satisfied. This can be the same line-feed that satisfied the last variable in the list. If the source is a device selector and no line feed is detected, the computer will be "hung" on the ENTER statement. If the source is a buffer and no line feed is detected, a NO TERM error is generated. This requirement can be removed by using "#" as the first image specifier. For more detailed information on statement terminators, see Formatted ENTER. A "hung" condition can be trapped by use of the SET TIMEOUT and ON TIMEOUT statements.

Related Statements

CONVERT
IMAGE
IOBUFFER
ON TIMEOUT
SET TIMEOUT
TRANSFER

ERROM


```
ERROM
```

Example Statements

```
30 X=ERROM
70 IF ERROM=192 THEN GOTO 100
```

Parameters

None

Action Taken

ERROM is a function that returns the ID number of the option ROM associated with the last error generated by an option ROM. All option ROMs use error numbers greater than 100. The ID number of the I/O ROM is 192. Note that ERROM is modified only by the occurrence of another option ROM error.

Related Statements

```
ERRL
ERRN
ERRSC
```

ERRSC


```
ERRSC
```

Example Statements

```
40 Y=ERRSC
90 IF ERRSC=7 THEN GOSUB 200
```

Parameters

None

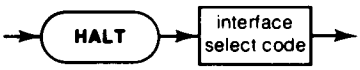
Action Taken

ERRSC is a function that returns the interface select code responsible for the most recent I/O error. Note that ERRSC is not cleared by a system reset and is modified only by the occurrence of another interface-dependent I/O error.

Related Statements

```
ERRL
ERRN
ERROM
```

HALT



HALT *interface select code*

Example Statements

```
100 HALT 7
200 HALT S1
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

Actions Taken

All interfaces: Stops current I/O operation. If an interface is HALTed with a TRANSFER active and an EOT branch enabled, the branch will be taken.

HP-IB: Leaves bus in present state.

Serial, BCD, GPIO: Does not affect external lines, so STATUS can be used to inspect line states. RESET or ABORTIO may be necessary after a halt to return handshake lines to the proper state for the next operation.

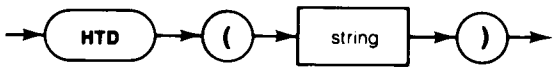
HP-IL:

- Active Controller: If a data transfer is in progress than a Not Ready for Data (NRD) is sent. If the interface is not involved in the transfer then RESUME may be used to continue the transfer.
- Non-controller: Leaves loop in present state.

Related Statements

```
ABORTIO
ON EOT
RESET
```

HTD



HTD(*string*)

Example Statements

```
20 Y=HTD(H$&L$)+A2
40 DISP HTD("F73A")
```

Parameters

string—a string expression that contains the base 16 representation of an integer. Limited to 4 significant characters that must be "0" thru "9" or "A" thru "F".

Action Taken

HTD is a function that returns the value of a base 16 representation contained in the string argument. The argument is a character representation and the result is a numeric quantity.

Related Statements

BTD
DTB$
DTH$
DTO$
OTD

IMAGE

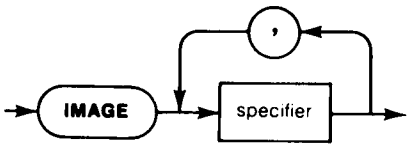


IMAGE *specifier* [, *specifier*] ...

Example Statements

```
10 LBL: IMAGE "Total =",4D.DD
100 IMAGE #,%K,2X,#K
```

Summary of OUTPUT Image Specifiers

Image	Meaning
A	Output one string character
B	Output number as one 8-bit byte
C	Output a comma separator in a number
D	Output one digit character; blank for leading zero
E	Output exponent information; five characters
e	Output exponent information; four characters
K	Output a variable in free-field format
M	Output number's sign if negative, blank if positive
P	Output a period separator in a number
R	Output a European radix point (comma)
S	Output number's sign, plus or minus
W	Output number as two 8-bit bytes (16-bit word)
X	Output one blank
Z	Output one digit character, including leading zeros
"..."	Output a literal
#	Suppress end-of-line sequence at end of statement
*	Output one digit character; asterisk for leading zero
.	Output an American radix point (decimal point)
/	Output and end-of-line sequence

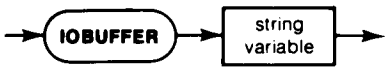
Summary of ENTER Image Specifiers

Image	Meaning
A	Demands one string character
B	Enter number as one 8-bit byte
C	Demand one character for a numeric field; allows commas to be skipped over
D	Demand one character for a numeric field
E	Demand five characters for a numeric field
e	Demand four characters for a numeric field
K	Enter a variable in free-field format
M	Demand one character for a numeric field
S	Demand one character for a numeric field
W	Enter number as two 8-bit bytes (16-bit word)
X	Skip one character
Z	Demand one character for a numeric field
#	Suppress requirement for a line-feed to terminate statement or field
%	Allow EOI to terminate statement or field
*	Demand one character for a numeric field
.	Demand one character for a numeric field
/	Demand a line feed

Related Statements

CONVERT
ENTER...USING
OUTPUT...USING

IOBUFFER



IOBUFFER *string variable*

Example Statements

```
10 DIM A$[88]
20 IOBUFFER A$
30 DIM B$[80]
```

Parameters

string variable—the name of a string variable with a dimensioned length 8 characters longer than the desired size of the buffer on the HP-83/85. Length may be the same as is desired on the HP-86/87.

Actions Taken

Eight characters of the string variable are reserved for control of buffer activity on the HP-83/85. On the HP-87 one of the ten IOBUFFER tables is reserved for control of buffer activity.

Buffer empty pointer:

- Initial value = 1. Accessed by Control/Status registers CR0, SR0. Characters are taken from the buffer (by ENTER or TRANSFER) using the following sequence:

  1. Read character
  2. Increment empty pointer

Buffer fill pointer:

- Initial value = 0. Accessed by Control/Status registers CR1, SR1. Characters are put into the buffer (by OUTPUT, TRANSFER, or string assignment) using the following sequence:

  1. Increment fill pointer
  2. Store character

Active-out select code:

- Initial value = 0. Accessed by Status register SR3. When active-out select code equals 0, there is no output TRANSFER operation active for this buffer. When an output TRANSFER is active for this buffer, the active-out select code is set equal to the interface select code that is the destination of the TRANSFER.

Active-in select code:

- Initial value = 0. Accessed by Status register SR2. When active-in select code equals 0, there is no input TRANSFER operation active for this buffer. When an input transfer is active for this buffer, the active-in select code is set equal to the interface select code that is the source of the transfer.

Conversion pointers:

- These pointers cannot be accessed from BASIC. When a CONVERT statement to the IOBUFFER is executed, pointers to the appropriate conversion tables are established. **These pointers are initialized by the IOBUFFER statement.** Therefore, execute CONVERT **after** executing IOBUFFER.

Full buffer:

- A buffer is full when the fill pointer equals the dimensioned length of the string (minus eight on the HP-85 or HP-83). Attempting to store data into a full buffer generates a BUFFER error.

Empty buffer:

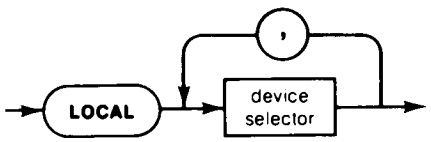
- A buffer is empty when the empty pointer equals the fill pointer plus one. When the buffer becomes empty, the fill pointer is reset to zero, and the empty pointer is reset to one. Active-out and active-in select codes are not affected by the buffer becoming empty; neither are the conversion pointers affected. Old data in the buffer is not lost, but the buffer fill pointer must be modified if you wish to re-access the data in the buffer (so the buffer will "look" full).

Buffer Status Registers		Buffer Control Registers	
Empty pointer	SR0	Empty pointer	CR0
Fill pointer	SR1	Fill pointer	CR1
Active in select code	SR2		
Active out select code	SR3		

Related Statements

CONTROL
CONVERT
ENTER
OUTPUT
STATUS
TRANSFER

LOCAL



LOCAL *device selector* [, *device selector*] ...

Example Statements

```
220 LOCAL 7
330 LOCAL 100*S+01 @ RESUME 7
```

Parameters

device selector—a valid interface select code or a valid combination of interface select code and primary address (see Choosing the Source or Destination in section 1). If multiple device selectors are specified, they must all be on the same interface select code.

Actions Taken

HP-IB:

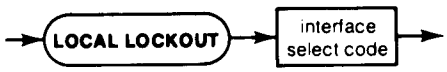
- If device selector is only an interface select code, Remote Enable (REN) is set false. Must be System Controller.
- If device selector contains a primary address, interface addresses specified device(s) and sends Go To Local (GTL) message. Leaves ATN true; use RESUME if you wish to set ATN false. Must be Active Controller.
- If device is in REMOTE with LOCAL LOCKOUT set, the device must receive the GTL message or have REN set false to be returned to local (front panel) control.

Serial, BCD, GPIO: Error

HP-IL: Must be active controller.

- If device selector is only an interface select code then Not Remote Enable (NRE) is sent.
- If device selector includes a primary address then Unlisten (UNL), Listen Address n (LADn), and Go To Local (GTL) are sent.
- If device is in REMOTE with LOCAL LOCKOUT set then the device must receive the Go To Local or Not Remote Enable message before it will return to local (front panel) control.

LOCAL LOCKOUT




```
LOCAL LOCKOUT interface select code
```

Example Statements

```
50 LOCAL LOCKOUT 50 @ RESUME 50
100 REMOTE 706,712 @ LOCAL LOCKOUT 7
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

Action Taken

HP-IB and HP-IL:

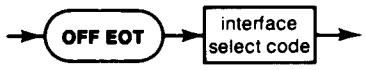
- Must be Active Controller. Sends Local Lockout (LLO) command. (HP-IB leaves ATN true; use RESUME if you wish to set ATN false.)
- Local Lockout remains in effect until the Remote Enable (REN) line is set false for HP-IB or, for HP-IL the Not Remote Enable (NRE) command is sent.

Serial, BCD, GPIO: Error

Related Statements

```
LOCAL
REMOTE
```

OFF EOT



OFF EOT *interface select code*

Example Statements

```
40 OFF EOT 3
120 IF S>128 THEN OFF EOT S1
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

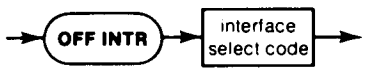
Action Taken

Disables end-of-line branching for termination of a transfer on the specified interface. OFF EOT does not cancel a branch permanently. For example, if the transfer has terminated and an ON EOT statement is re-executed, the branch will be taken at that time.

Related Statements

OFF INTR
OFF TIMEOUT
ON EOT
ON INTR
ON TIMEOUT

OFF INTR




```
OFF INTR interface select code
```

Example Statements

```
110 OFF INTR 7
180 IF X THEN OFF INTR S2
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

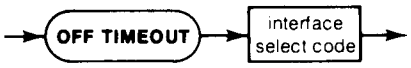
Action Taken

Disables end-of-line branching for interrupts from the specified interface. `OFF INTR` does not cancel a branch permanently. For example, if the interface has interrupted and an `ON INTR` statement is re-executed, the branch will be taken at that time.

Related Statements

```
CONTROL
ENABLE INTR
OFF EOT
OFF TIMEOUT
ON EOT
ON INTR
ON TIMEOUT
```

OFF TIMEOUT




```
OFF TIMEOUT interface select code
```

Example Statements

```
50 OFF TIMEOUT S DIV 100
120 OFF TIMEOUT 7
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

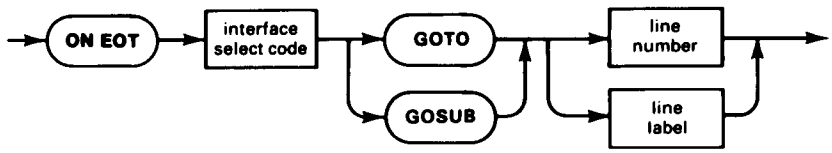
Action Taken

Disables end-of-line branching for occurrence of a timeout on the specified interface. OFF TIMEOUT does not cancel a branch permanently. For example, if the interface has timed out and an ON TIMEOUT statement is re-executed, the branch will be taken at that time.

Related Statements

```
OFF EOT
OFF INTR
ON EOT
ON INTR
ON TIMEOUT
SET TIMEOUT
```

ON EOT




```
ON EOT interface select code  GOTO line number
                              GOSUB line label
```

Example Statements

```
20 ON EOT 7 GOTO SERVICE 7
120 ON EOT S4 GOSUB 1000
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

line number—an integer constant from 1 thru 9999 that specifies a valid line number within the program.

line label—a name of up to 31 characters containing letters, numbers, or underscore symbol, whose first character must be a letter.

Actions Taken

Enables end-of-line branches to the specified line number when a transfer to or from the specified interface is terminated. A pending end-of-line branch from a previous, unserviced transfer termination (for the specified interface select code) is taken immediately. Only one transfer termination per interface select code is retained by the system.

Each interface may have alternate causes for transfer terminations that are user-programmable. Refer to the appropriate interface owner's manual for details about this capability.

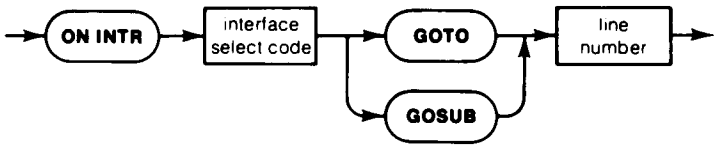
Overrides any previous ON EOT statement for the same interface select code.

Related Statements

```
CONTROL
OFF EOT
OFF INTR
OFF TIMEOUT
ON INTR
ON TIMEOUT
STATUS
TRANSFER
```

Also see Branch Precedence Table in section 6.

ON INTR




```
ON INTR interface select code GOTO line number
                              GOSUB line label
```

Example Statements

```
30 ON INTR 51 GOSUB 2000
60 ON INTR 3 GOTO EXIT
150 EXIT: ABORTIO
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

line number—an integer constant from 1 thru 9999 that specifies a valid line number within the program.

Actions Taken

Enables end-of-line branches to the specified line number when an interface interrupt occurs (see ENABLE INTR). A pending end-of-line branch from a previous, unserviced interface interrupt (for the specified interface select code) is taken immediately. Only one interrupt per interface select code is retained by the system.

Interrupt causes are specified by either ENABLE INTR or CONTROL statements. Interrupt causes are interface-dependent; refer to the appropriate interface owner's manual for details.

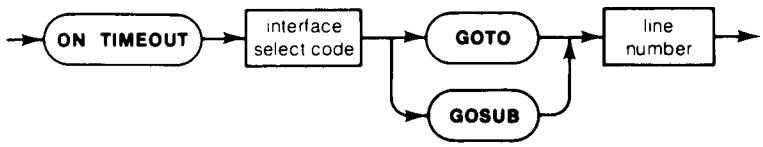
Overrides any previous ON INTR statement for the same interface select code.

Related Statements

```
CONTROL
ENABLE INTR
OFF EOT
OFF INTR
OFF TIMEOUT
ON EOT
ON TIMEOUT
```

Also see Branch Precedence Table in section 6.

ON TIMEOUT



ON TIMEOUT *interface select code* { GOTO / GOSUB } *line number*

Example Statements

```
20 ON TIMEOUT 53 GOSUB 2500
50 ON TIMEOUT 7 GOTO 320
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

line number—an integer constant from 1 thru 9999 that specifies a valid line number within the program.

Actions Taken

Enables end-of-line branches to the specified line number when an interface timeout occurs (see SET TIMEOUT). A pending end-of-line branch from a previous, unserviced interface timeout (for the specified interface select code) is taken immediately. Only one timeout per interface select code is retained by the system.

End-of-line branching for TIMEOUT is **not** applicable to the actual data movement portion of a TRANSFER (INTR or FHS) operation. A transfer can timeout if the interface or device cannot be addressed to start the transfer, but there will be no ON TIMEOUT branch if the peripheral device stops handshaking in the middle of the transfer.

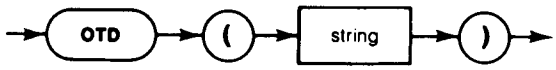
Overrides any previous ON TIMEOUT statement for the same interface select code.

Related Statements

```
OFF EOT
OFF INTR
OFF TIMEOUT
ON EOT
ON INTR
SET TIMEOUT
```

Also see Branch Precedence Table in section 6.

OTD



OTD(*string*)

Example Statements

```
80 X=OTD(H$&L$)+A3
110 DISP OTD("177345")
```

Parameters

string—a string expression that contains the base 8 representation of an integer. Limited to 6 significant characters that must be "0" thru "7" (except most significant character must be "0" or "1").

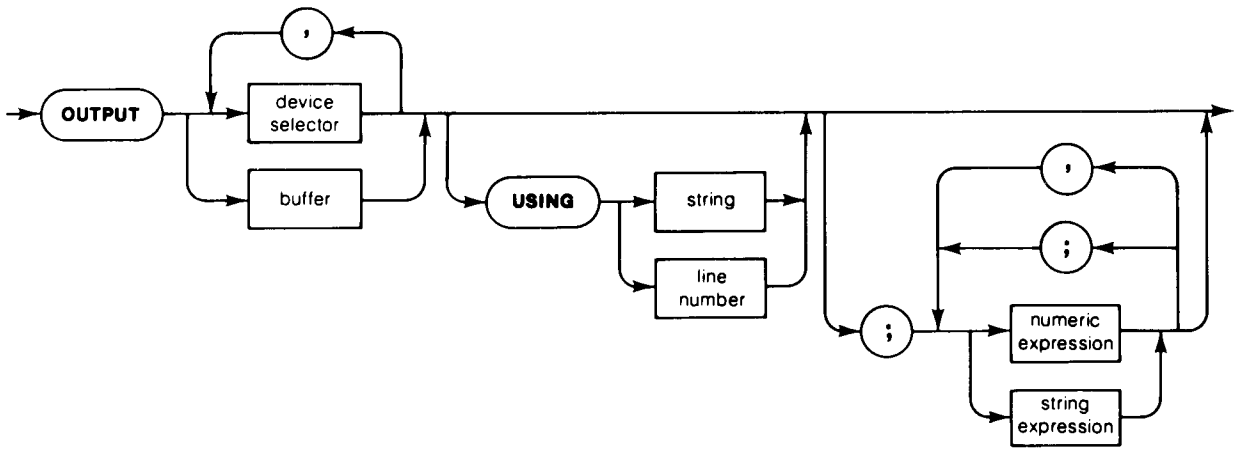
Action Taken

OTD is a function that returns the value of a base 8 representation contained in the string argument. The argument is a character representation and the result is a numeric quantity.

Related Statements

```
DTB$
DTH$
DTO$
BTD
HTD
```

OUTPUT



OUTPUT *buffer / device selector* [, *device selector* ...][USING *string / line number / line label*[[; *expression*[, *expression*] [; *expression*] ...]

Example Statements

```
70 OUTPUT 701 USING Outformat ; X,Y,Z
90 OUTPUT C$ ; N(I) ;Z$
120 OUTPUT 3 USING 30 ; A$
250 OUTPUT 100*S+A USING "#,B" ; N
```

Parameters

device selector—a valid interface select code or a valid combination of interface select code and primary address (see Choosing the Source or Destination). If multiple device selectors are specified, they must all be on the same interface select code. OUTPUT allows device selectors 1 and 2 for addressing the internal CRT and printer.

buffer—the name of a string variable that has been declared as an IOBUFFER.

string—a string expression that contains a valid set of image specifiers.

line number/label—the line number or label or 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 buffer or device(s); bytes may be string or numeric. If a CONVERT operation is specified, the conversion is performed immediately before the byte is sent to the interface or buffer.

When USING is not specified, and output items are separated by commas, free-field format is used. A free-field output of a string item causes it to be left-justified in a field with no more than 20 trailing blanks. A free-field output of a numeric item causes it to be left-justified in a field of 11, 21, or 32 characters.

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

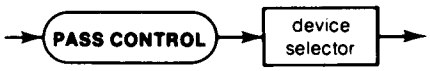
When USING is specified, output operations are formatted according to the image specifiers used. Image specifiers may be enclosed in quotes and placed in the OUTPUT statement, contained in a string variable named in the OUTPUT statement, or placed in an IMAGE statement referenced by the OUTPUT statement.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

OUTPUT sends an end-of-line sequence after the last item in the OUTPUT list. This sequence is interface-dependent, can be changed by the CONTROL statement, and defaults to carriage return/line feed. This sequence can be suppressed by using "#" as the first image specifier. For more detailed information on statement terminators, see Formatted OUTPUT. If the OUTPUT is to a buffer, a carriage return/line feed is placed in the buffer after the last data byte unless the "#" image is used.

Related Statements

CONTROL
CONVERT
IMAGE
IOBUFFER
TRANSFER

PASS CONTROL




```
PASS CONTROL device selector
```

Example Statements

```
100 PASS CONTROL 100%S+D
250 PASS CONTROL 721 @ ENABLE INTR 7;32
```

Parameters

device selector—a valid interface select code or a valid combination of interface select code and primary address (see Choosing the Source or Destination in section 1).

Actions Taken

HP-IB and HP-IL: Must be Active Controller. Passes Active Controller responsibility to the specified device.

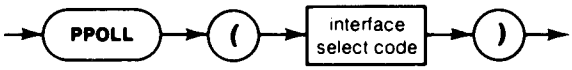
- If device selector is only an interface select code, interface sends the Take Control (TCT) message (and sets ATN false for HP-IB). Be sure that the device receiving control has been addressed to talk before using this form of PASS CONTROL.
- If device selector contains a primary address, interface sends the specified device's talk address, sends the TCT message (then sets ATN false for HP-IB).

Serial, BCD, GPIO: Error

Related Statements

```
ABORTIO
ENABLE INTR
ON INTR
REQUEST
RESET
```

PPOLL



PPOLL(*interface select code*)

Example Statements

```
310 X=PPOLL(7)
620 P9=PPOLL(S0)
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

Actions Taken

HP-IB and HP-IL:

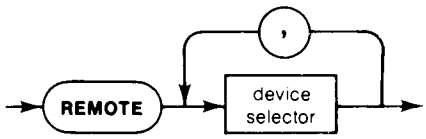
- Must be Active Controller. PPOLL is a function that returns the results of a Parallel Poll operation. Sends Identify (IDY) message. Devices capable of responding each assert one bit of the parallel poll response byte.

Serial, BCD, GPIO: Error

Related Statements

SPOLL

REMOTE



REMOTE *device selector* [, *device selector*] ...

Example Statements

```
50 REMOTE 720
130 REMOTE 100*S+D @ RESUME S
```

Parameters

device selector—a valid interface select code or a valid combination of interface select code and primary address (see Choosing the Source or Destination in section 1). If multiple device selectors are specified, they must all be on the same interface select code.

Actions Taken

HP-IB: Must be System Controller. Puts the bus into remote operation.

- If device selector is only an interface select code, interface sets Remote Enable (REN) true. Devices do not go into remote state until they are addressed to listen.
- If device selector contains a primary address, interface sets REN true, sends Unlisten (UNL) message, then sends the listen address of the specified device(s). REMOTE leaves ATN true; use RESUME if you wish to set ATN false.

Serial: Error

BCD: Sets partial field separator. Refer to the BCD Interface Owner's Manual for details.

GPIO: Error

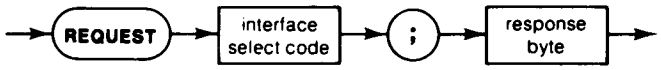
HP-IL: Must be Active Controller.

- If device selector is only an interface select code then the Remote Enable (REN) message is sent.
- If the device selector includes a primary address then Remote Enable (REN), Unlisten (UNL), and Listen Address(es) (LAD) are sent.

Related Statements

```
LOCAL
LOCAL LOCKOUT
RESUME
```

REQUEST



REQUEST *interface select code* ; *response byte*

Example Statements

```
50 REQUEST 7 ; 64+4
260 IF T>40 THEN REQUEST S ; 64+X
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

response byte—a numeric expression that evaluates to an integer 0 thru 255.

Actions Taken

HP-IB and HP-IL: Must be non-controller. Sets up a Serial Poll response byte. Sets Service Request (SRQ) true if bit 6 (decimal value 64) of the response byte is set. The response byte is sent to the Active Controller in response to an incoming Serial Poll operation. The Active Controller's Serial Poll operation clears SRQ, which can also be cleared by executing REQUEST with bit 6 of the response byte equal to zero.

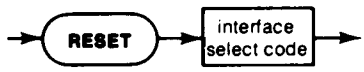
Serial: Sends a BREAK. The BREAK is defined by the response byte. A space (0-state) condition is held for the number of character times specified in the response byte. It is then followed by a mark (1-state) condition for five character times.

BCD, GPIO: Error

Related Statements

```
PASS CONTROL
SPOLL
```

RESET



RESET *interface select code*

Example Statements

```
30 RESET 7
300 IF S>128 THEN RESET S3
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

Actions Taken

All interfaces: Performs a hardware reset of the interface, returning it unconditionally to its power-on state. The interface performs a self test (failure causes ERROR 110), and the control registers are set according to the configuration switches on the interface circuit assembly. Resetting an interface with a transfer active and EOT branching enabled causes the branch to be taken.

HP-IB: If System Controller, sends Interface Clear (IFC), then Remote Enable (REN).

Serial: Modem control lines are turned off.

BCD: Data lines are set to high-impedance state, handshake lines are set false, and I/O lines are set to input state.

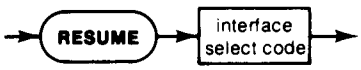
GPIO: Ports A and B are set to high-impedance state, Ports C and D are set to off state, CTL lines are set false, and OUTA and OUTB are set to indicate output.

HP-IL: If the interface is system controller then Interface Clear (IFC), Auto Address Unconfigure (AAU), and Auto Address 1 (AAD1) are sent, followed by Not Remote Enable (NRE) and Remote Enable (REN).

Related Statements

```
ABORTIO
HALT
ON EOT
```

RESUME




```
RESUME interface select code
```

Example Statements

```
110 RESUME 7
190 RESUME S DIV 100
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

Actions Taken

HP-IB: Must be Active Controller (CA = 1). Sets the Attention (ATN) line false. Statements that can leave the ATN line true are: CLEAR, LOCAL, LOCAL LOCKOUT, REMOTE, SEND, TRIGGER.

Serial: The transmitter is enabled. Refer to the *HP 82939A Serial Interface Owner's Manual* for details.

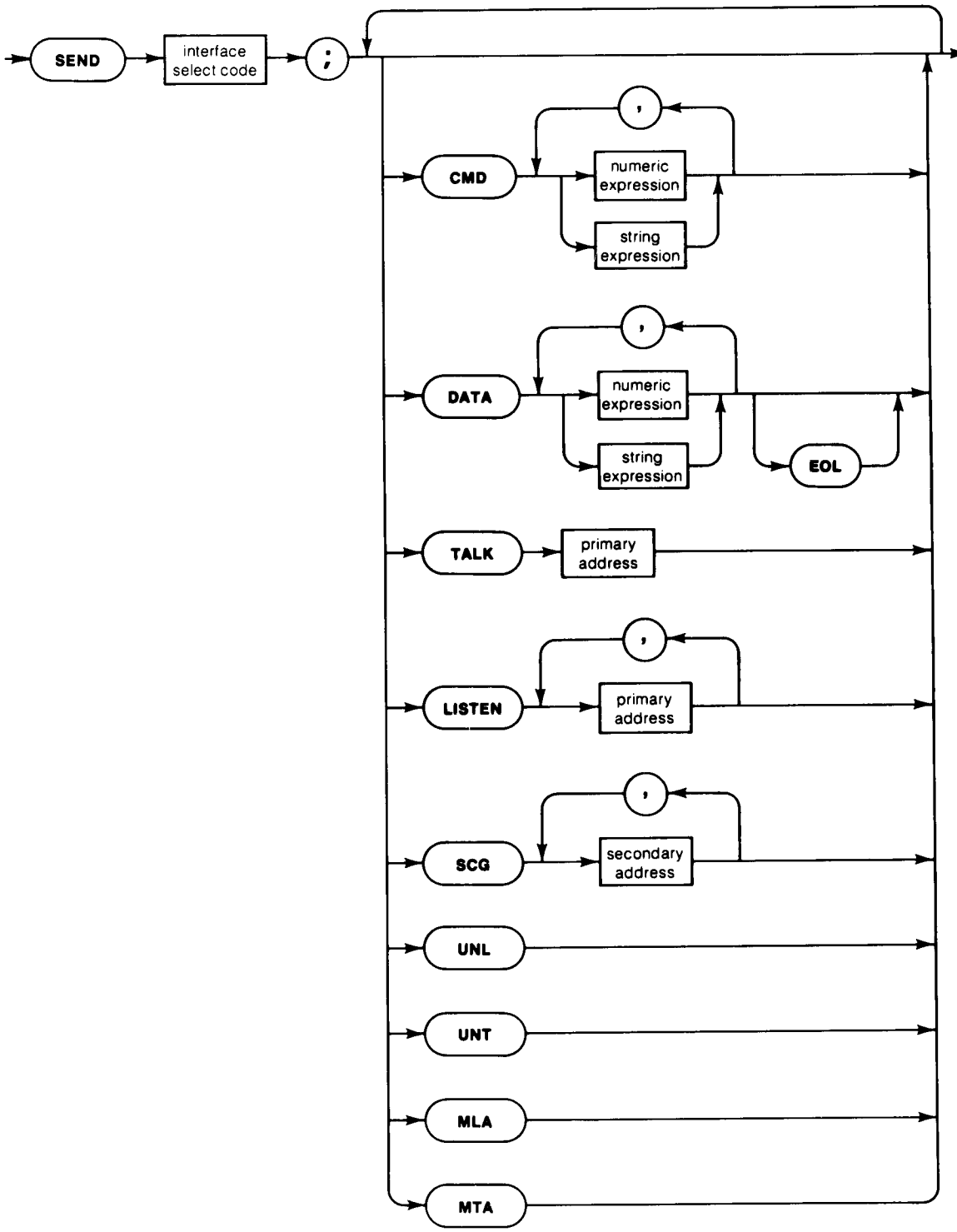
BCD, GPIO: Error

HP-IL: Must be active controller. The Send Data (SDA) message is sent if a transfer is not already in progress.

Related Statements

```
CONTROL
HALT
SEND
```

SEND


SEND
interface select code
;
,
CMD
numeric expression
string expression
,
DATA
numeric expression
string expression
EOL
TALK
primary address
,
LISTEN
primary address
,
SCG
secondary address
UNL
UNT
MLA
MTA

SEND *interface select code* ; [[CMD *list*][DATA *list*[EOL]][TALK *primary address*][LISTEN *primary address*[, *primary address*]...][SCG *secondary address*[, *secondary address*]...][UNL][UNT][MLA][MTA]...]

Example Statements

```
100 SEND 7 ; CMD "U?%" DATA "Hello"
200 SEND 7 ; CMD A$ SCG 14,18 DATA X$
300 SEND 5 ; MTA UNL LISTEN 6,14 CMD P,R SCG 6
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

list—a list of numeric or string expressions, separated by commas.

primary address—a numeric expression that evaluated to an integer 0 thru 31.

secondary address—a numeric expression that evaluates to an integer 0 thru 31.

Actions Taken

HP-IB: When sending any commands (CMD, TALK, LISTEN, SCG, UNL, UNT, MLA, MTA), the interface must be Active Controller. The ATN line is set true while sending commands. The ATN line is set false while sending DATA, even if no actual data is sent (i.e., DATA" ").

- CMD: Commands: send list of 8-bit expressions with ATN true. Primary commands have a bit pattern = X00CCCCC, where X = don't care, C = bits of command (decimal 0 thru 31). SEND CMD can be used to create odd parity on commands, if necessary.
- DATA: Send list of numeric or string expressions with ATN false. Any 8-bit pattern may be sent. If EOL is specified, the interface's end-of-line character sequence is sent following data (Control registers 17-thru 23).
- TALK: Send device's Talk Address (TAD), decimal 0 thru 31.
- LISTEN: Send device's Listen Address (LAD), decimal 0 thru 31.
- SCG: Secondary Command Group: Send secondary address to device.
- UNL: Send Unlisten command (UNL). Numeric value sent is 63; ATN is true.
- UNT: Send Untalk command (UNT). Numeric value sent is 95; ATN is true.
- MLA: Send My Listen Address (MLA). This is the listen address of the interface. Factory setting = 53.
- MTA: Send My Talk Address (MTA). This is the talk address of the interface. Factory setting = 85.

Serial: The only form that can be sent is DATA.

- DATA: Sends list of numeric or string expressions. If EOL is specified, the interface's end-of-line character sequence is sent (control registers 17 thru 23).

BCD: See BCD Owner's Manual for details:

- CMD: Primary addresses 0 thru 6 set partial field specifier.
- DATA: Lower 4 bits of data bytes are sent; control characters, spaces, and commas are ignored. If EOL is specified, data format checking is enabled.
- LISTEN, TALK: Primary addresses 0 thru 6 set partial field specifier.
- SCG: Error.
- UNL, UNT, MLA, MTA: Ignored.

GPIO: See GPIO Owner's Manual for details.

- CMD: Primary addresses 0 thru 15 select port configuration. Device Clear command pulses RESA and RESB. Selected Device Clear pulses RESA or RESB according to the most recent primary address.
- DATA: Send list of numeric or string expressions. Data is sent as 8-bit bytes. If EOL is specified, the interface's end-of-line character sequence is sent (control registers 17 thru 23).
- LISTEN, TALK: Primary addresses 0 thru 15 select port configuration.
- SCG: Error.
- UNL, UNT, MLA, MTA: Ignored.

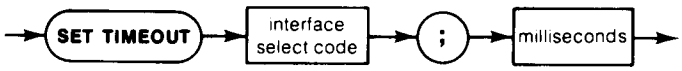
HP-IL: When sending commands (CMD, TALK, LISTEN, SCG, UNL, UNT, MLA, MTA) the interface must be active controller.

- CMD: Sends a list of 8 bit expressions as command frames.
- DATA: Sends a list of 8 bit expressions as data frames. If EOL is specified then the interface end-of-line sequence is sent following the data.
- TALK: Sends a device's Talk Address, decimal 0 to 31.
- LISTEN: Sends device Listen Address(es), decimal 0 to 31.
- SCG: Sends a secondary address frame, decimal 0 to 31.
- UNL: Sends an Auto Address sequence and the Unlisten command frame.
- UNT: Sends an Untalk command frame.
- MLA: Addresses the interface to Listen.
- MTA: Sends the Talk Address of the interface.

Related Statements

OUTPUT

SET TIMEOUT



SET TIMEOUT *interface select code* ; *milliseconds*

Example Statements

```
100 SET TIMEOUT 50 ; 2*1000
280 ON TIMEOUT 7 GOTO 550 @ SET TIMEOUT 7 ; 4500
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

milliseconds—a numeric expression that evaluates to an integer 0 thru 32,767.

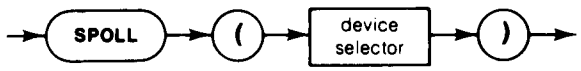
Action Taken

Establishes an approximate time limit (in milliseconds) that the interface will wait to complete a handshake with its peripheral device. If the specified time limit is exceeded and ON TIMEOUT end-of-line branching is enabled, the branch is taken. If no ON TIMEOUT is currently in effect, there is **no** indication that a timeout has occurred until an ON TIMEOUT is subsequently executed.

Related Statements

OFF TIMEOUT
ON TIMEOUT

SPOLL



SPOLL(*device selector*)

Example Statements

```
50 P=SPOLL(S4)
250 IF SPOLL(701)>63 THEN GOTO 750
```

Parameters

device selector—a valid interface select code or a valid combination of interface select code and primary address (see "Choosing the Source or Destination").

Actions Taken

HP-IB:

- Conducts a Serial Poll of a device on the bus and returns the device's status byte. If bit 6 of the status byte is set (decimal value 64), it indicates that the device is requesting service (asserting SRQ).
- If device selector is only an interface select code, interface sends Serial Poll Enable (SPE), sets ATN false, receives the status byte, sends Serial Poll Disable (SPD), then sends Untalk (UNT).
- If device selector contains a primary address, interface sends Unlisten (UNL), My Listen Address (MLA), devices Talk Address (TAD), Serial Poll Enable (SPE), then sets ATN false. It receives the status byte, sends Serial Poll Disable (SPD), then sends Untalk (UNT).

Serial, BCD, GPIO: Error.

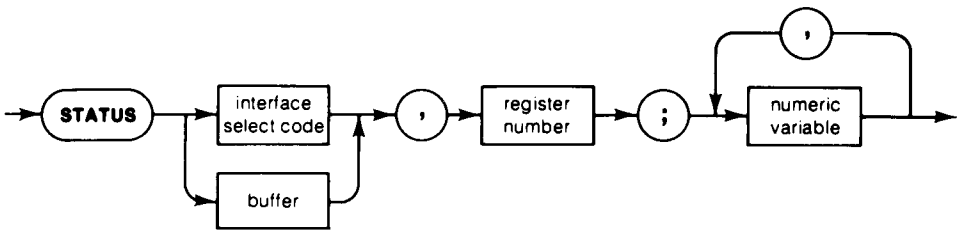
HP-IL: Must be active controller.

- SPOLL returns the first byte received in response to a serial poll of a device.
- If the device selector is just an interface select code then the interface sends the Send Status (SST) message, then waits to receive a data byte followed by end of transmission (EOT) and then sends (UNT).
- If the device selector includes a primary address then unlisten (UNL), my talk address (MTA), the device's talk address (TAD) and send status (SST) are sent. The data byte is received followed by end of transmission (EOT) and then untalk (UNT) is sent.

Related Statements

PPOLL

STATUS



STATUS { *interface select code* / *buffer* } , *register number* ; *numeric variable* [, *numeric variable*] ...

Example Statements

```
20 STATUS 7,0 ; C0,C1,C2,C3,C4
70 STATUS S1,5 ; .S1-5*
```

Parameters

interface select code—a numeric expression that evaluates to an integer 3 thru 10.

buffer—the name of a string variable that has been declared as an IOBUFFER.

register number—a numeric expression that evaluates to an integer 0 thru 15. Must specify a valid status register for the selected interface.

numeric variable—any numeric variable intended as a destination for the status information.

Actions Taken

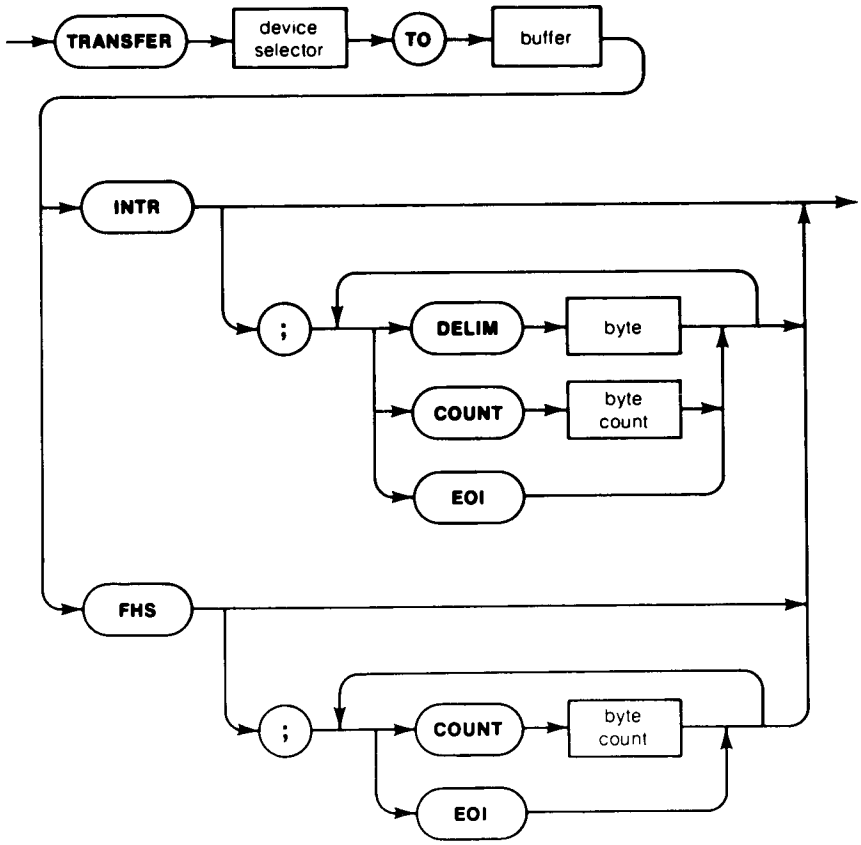
Reads one or more status register(s) and assigns the value(s) to the specified variable(s). When more than one variable is specified, consecutive status registers are read, starting at the specified register number. Status values returned are integers 0 thru 255.

Related Statements

```
ASSERT
CONTROL
ENABLE INTR
IOBUFFER
```

*Labels are allowed on the HP-87.

TRANSFER (in)



TRANSFER *device selector* TO *buffer* INTR[;[COUNT *byte count*][DELIM *byte*][EOI]]

TRANSFER *device selector* TO *buffer* FHS[;[COUNT *byte count*][EOI]]

Example Statements

```
100 TRANSFER 706 TO B$ INTR
200 TRANSFER 100*S+D TO B$ INTR ; COUNT 80 DELIM 10 EOI
300 TRANSFER 3 TO A$ FHS ; COUNT 16
```

Parameters

device selector—a valid interface select code or a valid combination of interface select code and primary address (see "Choosing the Source or Destination").

buffer—the name of a string variable that has been declared as an IOBUFFER.

byte count—a numeric expression that evaluates to an integer 0 thru 32 767. Specifies maximum number of bytes to be input.

byte—a numeric expression that evaluates to an integer 0 thru 255. Specifies the ASCII value of a character which can terminate the transfer.

Actions Taken

Takes data bytes from the specified device and places them into the specified buffer. Characters are placed into the buffer according to the buffer fill pointer. The transfer terminates when the buffer is full or when the first one of the specified terminating conditions is met. The interface may also have a programmable terminating condition; refer to the appropriate interface owner's manual for details. Specifying COUNT sets a maximum limit on the number of characters to be transferred. DELIM specifies the numeric value of a character that can terminate the transfer. Specifying EOI (End or Identify) allows the transfer to terminate when an interface-dependent "END" signal is detected (such as the EOI line on HP-IB). The terminating condition for buffer full is always in effect. If an ON EOT branch is enabled, the branch is taken when the transfer terminates.

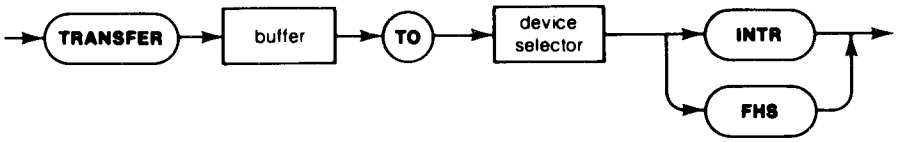
If INTR (Interrupt) is specified, the interface is automatically enabled to interrupt the computer each time it is ready with a new character. The transfer continues to completion even though program execution may have stopped. A WARNING 101 is issued if the program stops with a transfer still active. Be certain that the transfer has terminated (use RESET, HALT, or ABORTIO) before attempting to modify the program. This transfer type (under ideal conditions) is capable of a maximum data transfer rate of about 400 bytes per second.

If FHS (Fast Handshake) is specified, the interface and computer are dedicated to the transfer until it is complete. No interrupts of keypresses (not even the RESET key) are detected until the transfer terminates. If the computer "locks up" on a FHS TRANSFER, only a power-on or special interface-specific termination (i.e., Interface Clear on HP-IB) can return the computer to its normal state. This transfer type (under ideal conditions) is capable of data transfer rates in excess of 20 000 bytes per second.

Related Statements

ABORTIO
CONTROL
ENTER
HALT
IOBUFFER
ON EOT
RESET
STATUS

TRANSFER (out)



TRANSFER *buffer* TO *device selector* $\begin{matrix} \text{INTR} \\ \text{FHS} \end{matrix}$

Example Statements

```
150 TRANSFER B$ TO 721 INTR
280 OUTPUT B$ USING "#,K" ; D$ @ TRANSFER B$ TO 9 FHS
400 ON EOT S1 GOSUB 660 @ TRANSFER X$ TO S1 INTR
```

Parameters

buffer—the name of a string variable that has been declared as an IOBUFFER.

device selector—a valid interface select code or a valid combination of interface select code and primary address (see Choosing the Source or Destination in section 1 of this manual). With HP-IB, and only a select code specified, a multiple device transfer may be obtained.

Actions Taken

Takes data bytes from the specified buffer and sends them to the specified device. Data is taken from the buffer according to the buffer empty pointer. If the device selector contains a primary address, addressing is performed prior to sending the first byte. The interface's programmable end-of-line sequence is sent after the last byte from the buffer has been sent. Note that the buffer may contain an additional carriage-return/line-feed placed there by an OUTPUT statement. The transfer terminates when the buffer is empty. If ON EOT branching is enabled, the branch is taken when the transfer terminates.

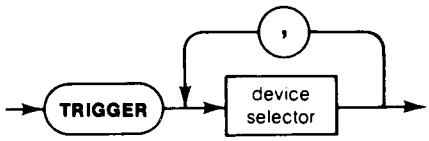
If INTR (Interrupt) is specified, the interface is automatically enabled to interrupt the computer each time it is ready for a new character. The transfer continues to completion even though program execution may have stopped. A WARNING 101 is issued if the program stops with a transfer still active. Be certain that the transfer has terminated (use STATUS, RESET, HALT, or ABORTIO) before attempting to modify the program. This TRANSFER type (under ideal conditions) is capable of a maximum data transfer rate of about 400 bytes per second.

If FHS (Fast Handshake) is specified, the interface and computer are dedicated to the transfer until it is complete. No interrupts or keypresses (not even the RESET key) are detected until the transfer terminates. If the computer "locks up" on a FHS TRANSFER, only a power-on or special interface-specific termination (i.e., Interface Clear on HP-IB or HP-IL) can return the computer to its normal state. This transfer type (under ideal conditions) is capable of data transfer rates in excess of 20 000 bytes per second.

Related Statements

```
ABORTIO
CONTROL
HALT
IOBUFFER
ON EOT
OUTPUT
RESET
STATUS
```

TRIGGER



TRIGGER *device selector* [, *device selector*] ...

Example Statements

```
70 TRIGGER 706,715 @ ENTER 706 ; V1
190 TRIGGER S1
```

Parameters

device selector—a valid interface select code or a valid combination of interface select code and primary address (see Choosing the Source or Destination in section 1). If multiple devices are selected, they must all be on the same interface select code.

Actions Taken

HP-IB: Must be Active Controller. Sends the Group Execute Trigger command (GET).

- If device selector is only an interface select code, interface sends the GET command. Those devices already addressed to listen respond to the GET command.
- If device selector contains a primary address, interface sends Unlisten (UNL), then the Listen Address (LAD) of the specified device(s). Sends the GET command (HP-IB leaves ATN true; use RESUME if you wish to set ATN false).

Serial, BCD, GPIO: Error.

HP-IL: Must be active controller. If the device selector is only an interface select code then the group execute trigger (GET) message is sent. If the device selector includes a primary address then unlisten (UNL), listen address (LADn) and group execute trigger are sent.

Related Statements

RESUME
SEND

Appendix B

Maintenance, Service, and Warranty

Maintenance

The I/O ROM does not require maintenance. However, there are several areas of caution that you should be aware of. They are:

WARNING: Do not place fingers, tools, or other foreign objects into the plug-in ports. Such actions may result in minor electrical shock hazard and interference with some pacemaker devices. Damage to plug-in port contacts and the computer's internal circuitry may also result.

CAUTION: Always switch off the computer and any peripherals involved when inserting or removing modules. Use only plug-in modules designed by Hewlett-Packard specifically for the HP Series 80 Personal Computer that you are using. Failure to do so could damage the module, the computer, or the peripherals.

CAUTION: If a module or ROM drawer jams when inserted into a port, it may be upside down or designed for another port. Attempting to force it may damage the computer or the module. Remove the module carefully and reinsert it.

CAUTION: Handle the plug-in ROMs very carefully while they are out of the ROM drawer. Do not insert any objects in the contact holes on the ROM. Always keep the protective cap in place over the ROM contacts while the ROM is not plugged into the ROM drawer. Failure to observe these cautions may result in damage to the ROM or ROM drawer.

For instructions on how to insert and remove the ROM and ROM drawer, please refer to the *ROM Drawer Instruction Sheet* or the HP Series 80 owner's manuals.

Service

If at any time you suspect that the ROM drawer or I/O ROM may be malfunctioning, do the following:

1. Turn the computer and all peripherals off. Disconnect all peripherals and remove the ROM drawer from the computer ports. Turn the computer back on. If the computer does not respond or displays `Error 23 : SELF TEST`, the computer requires service.

2. Turn the computer off. Install the ROM drawer, with the I/O ROM installed, into any port. Turn the computer back on.

- If `Error 112 : I/O ROM` is displayed, indicating that the ROM is not operating properly, turn the computer off and try the ROM in another ROM drawer slot. This will help you determine if particular slots in the ROM drawer are malfunctioning, or if the ROM itself is malfunctioning.
- If the cursor does not appear, the system is not operating properly. To help determine what is causing the improper operation, repeat step 2 with the ROM drawer installed in a different port, both with the I/O ROM installed in the ROM drawer and with the I/O ROM removed from the ROM drawer.

3. Refer to How to Obtain Repair Service for information on how to obtain repair service for the malfunctioning device.

Warranty Information

The complete warranty statement is included in the information packet shipped with your ROM. Additional copies may be obtained from any authorized HP dealer, or the HP sales and service office where you purchased your system.

If you have questions concerning the warranty, and you are unable to contact the authorized HP sales and service office where you purchased your computer, please contact:

In the U.S.:

Hewlett-Packard
Corvallis Division Customer Support
1000 N.E. Circle Blvd.
Corvallis, OR 97330
Tel. (503) 758-1010
Toll Free Number: (800) 547-3400 (except in Oregon, Hawaii and Alaska).

In Europe:

Hewlett-Packard S.A.
7, rue du Bois-du-lan
P. O. Box
CH-1217 Meyrin 2
Geneva
Switzerland
Tel. (22) 82 70 00

Other Countries:

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

How to Obtain Repair Service

Not all Hewlett-Packard facilities offer service for the HP Series 80 Personal Computer and its peripherals. For information on service in your area, contact your nearest authorized HP dealer or the nearest Hewlett-Packard sales and service office.

If your system malfunctions and repair is required, you can help assure efficient servicing by having the following items with your unit(s) at the time of service:

1. A description of the configuration of the computer, exactly as it was at the time of malfunction, including any plug-in modules, tape cartridges or other accessories.
2. A brief description of the malfunction symptoms for service personnel.
3. Printouts or any other materials that illustrate the problem area.
4. A copy of the sales slip or other proof of purchase to establish the warranty coverage period.

General Shipping Instructions

Should you ever need to ship any portion of your computer system, be sure it is packed in a protective package (use the original case), to avoid in-transit damage. Hewlett-Packard suggests that the customer always insure shipments.

If you happen to be outside of the country where you bought your computer or peripheral, contact the nearest authorized HP Series 80 dealer or the local Hewlett-Packard office. All customs and duties are your responsibility.

Appendix C

Error Messages

Error Message	Meaning	Possible Corrective Action
101 XFR	This is only a warning. It is issued when a program is paused with an I/O transfer still active. Do not attempt to modify a program when a transfer is active.	Before you modify or rerun the program, stop all active transfers with a RESET, HALT, or ABORTIO instruction; or press the (RESET) key.
110 I/O CARD	An interface has failed self-test. This indicates a probable hardware problem.	ERRSC can be used to determine which interface has failed. Try recycling the power (turn computer off, then back on again). If the interface still fails, contact the authorized dealer or the HP sales and service office from which you purchased your computer.
111 I/O OPER	The I/O operation attempted is not valid with the type of interface being used. Some examples are: specifying a status or control register that does not exist, using a primary address with a Serial interface, or using an I/O statement that is not defined for the interface being used.	ERRL can be used to identify the improper statement. Check this statement in the Syntax Reference section to determine if it is defined for the interface being used. If the statement is valid, check the appropriate interface owner's manual to get details on the proper mode or configuration required for the statement used.
112 I/O ROM	The I/O ROM has failed the checksum self-test. This indicates a probable hardware problem.	Try recycling the power (turn the computer off, then back on again). If the error keeps recurring, contact the authorized dealer or the HP sales and service office from which your purchased your computer.
113	An interface-dependent error. HP-IB: The statement used requires the interface to be system controller. Serial: UART receiver overrun; data has been lost. BCD: Attempting to put the interface into an illegal mode. GPIO: An odd number of bytes was transferred when the interface was configured for 16-bit words. HP-IL: The Take Control (TCT) message was ignored by the device.	ERRSC can be used to determine the source of the error. Refer to the appropriate interface owner's manual to get details on the error and possible corrective actions.

Error Message	Meaning	Possible Corrective Action
114	An interface-dependent error. HP-IB and HP-IL: The statement used requires the interface to be active controller. Serial: Receiver buffer overrun; data has been lost. BCD: Port 10 not currently available. GPIO: FHS TRANSFER aborted by STO.	ERRSC can be used to determine the source of the error. Refer to the appropriate interface owner's manual to get details on the error and possible corrective actions.
115	An interface-dependent error. HP-IB and HP-IL: The statement used requires the interface to be addressed to talk. Serial: Automatic disconnect forced. BCD: FHS TRANSFER aborted by FLGB. GPIO: Interface configuration does not allow an output enable or output operation on Port A or Port B.	ERRSC can be used to determine the source of the error. Refer to the appropriate interface owner's manual to get details on the error and possible corrective actions.
116	An interface-dependent error. HP-IB and HP-IL: The statement used requires the interface to be addressed to listen. Serial: This error number not currently used. BCD: Data direction mismatch on current operation. GPIO: Cannot start operation because handshake CTL line is not in proper state.	ERRSC can be used to determine the source of the error. Refer to the appropriate interface owner's manual to get details on the error and possible corrective actions.
117	An interface-dependent error. HP-IB and HP-IL: The statement used requires the interface to be non-controller. Serial: This error number not currently used. BCD: Interface command has been directed to a non-existent field. GPIO: This error number not currently used.	ERRSC can be used to determine the source of the error. Refer to the appropriate interface owner's manual to get details on the error and possible corrective actions.
118	An interface-dependent error. HP-IB: This error number not currently used. Serial: This error number not currently used. BCD: Cannot start operation because CTL line is not in the proper state. GPIO: This error number not currently used. HP-IL: Protocol violation or loop transmission error.	ERRSC can be used to determine the source of the error. Refer to the appropriate interface owner's manual to get details on the error and possible corrective actions.

Error Message	Meaning	Possible Corrective Action
119	An interface-dependent error. HP-IB: This error number not currently used. Serial: This error number not currently used. BCD: Data format does not match the mode of the interface. GPIO: This error number not currently used. HP-IL: Addressed talker ignored the start of transmission (SOT) frame.	ERRSC can be used to determine the source of the error. Refer to the appropriate interface owner's manual to get details on the error and possible corrective actions.
123 NO ";"	Syntax error. A semicolon delimiter was expected in the statement.	Put the semicolon where it belongs.
124 ISC	Either the interface select code specified is out of range, or there is no interface present set to the specified select code. Interface select codes must be in the range of 3 thru 10. Select codes 1 (CRT) and 2 (internal printer) are allowed for OUTPUT statements only.	Be sure that the interface select code is within the proper range. Pay special attention to variables that are used to hold interface select codes. If the interface select code is OK, be sure that the interface is plugged in properly. Finally, check the switch settings on the interface. (Someone might have changed them last weekend.)
125 ADDR	The primary address specified is improper. Only addresses 00 thru 31 are allowed, but not all interfaces use this entire range.	Be sure that the primary address is within the proper range. Pay special attention to variables that are used to hold addresses or device selectors.
126 BUFFER	Four possible buffer problems: (1) The string variable specified has not been declared as an IOBUFFER. (2) Attempting to ENTER from a buffer which is out of data. (3) Attempting to OUTPUT to a buffer which is already full. (4) Attempting an output TRANSFER with an empty buffer.	Be sure you have included the necessary IOBUFFER statement. Check the logical flow of your program (in what order the statements are executed). Buffer contents can be examined at any time by simply printing or displaying the string variable being used as the buffer. If this doesn't provide enough information, the buffer pointers can be examined with the STATUS statement.
127 NUMBER	An incoming character sequence does not constitute a valid number, or a number being output requires three exponent digits and an "e" format was specified.	If the error is from an output operation, check the magnitude of the number and the format used. If the error is from an input operation, there are many possible causes. Here are some things to look for: more than 255 leading non-numeric characters, unexpected spaces in a character stream when a character-count format is used, punctuation sequences that include potentially numeric characters used in an order that is numerically meaningless.

Error Message	Meaning	Possible Corrective Action
128 EARLY TERM	A buffer was emptied before all the ENTER fields were satisfied, or a field terminator was encountered before the specified character count was reached.	Check your incoming character stream, ENTER list, and image specifiers.
129 VAR TYPE	The type (string or numeric) of a variable in an ENTER list does not match with the image specified for that variable.	Check your ENTER list and image specifiers.
130 NO TERM	A required terminator was not received from an interface or buffer during an ENTER statement. Remember that there is a default requirement for a line-feed statement terminator.	Check your incoming character stream, ENTER list, and image specifiers.

Appendix D

ASCII Character Set

EQUIVALENT FORMS				EQUIVALENT FORMS				EQUIVALENT FORMS				EQUIVALENT FORMS			
Char.	Binary	Octal	Dec	Char.	Binary	Octal	Dec	Char.	Binary	Octal	Dec	Char.	Binary	Octal	Dec
@[c]	00000000	000	0	SPACE	00100000	040	32		01000000	100	64	KEY LABEL [s]	01100000	140	96
A[c]	00000001	001	1	!	00100001	041	33		01000001	101	65		01100001	141	97
B[c]	00000010	002	2	"	00100010	042	34		01000010	102	66		01100010	142	98
C[c]	00000011	003	3	#	00100011	043	35		01000011	103	67		01100011	143	99
D[c]	00000100	004	4	$	00100100	044	36		01000100	104	68		01100100	144	100
E[c]	00000101	005	5	%	00100101	045	37		01000101	105	69		01100101	145	101
F[c]	00000110	006	6	&	00100110	046	38		01000110	106	70		01100110	146	102
G[c]	00000111	007	7	'	00100111	047	39		01000111	107	71		01100111	147	103
H[c]	00001000	010	8	(	00101000	050	40		01001000	110	72		01101000	150	104
I[c]	00001001	011	9	)	00101001	051	41		01001001	111	73		01101001	151	105
J[c]	00001010	012	10	*	00101010	052	42		01001010	112	74		01101010	152	106
K[c]	00001011	013	11	+	00101011	053	43		01001011	113	75		01101011	153	107
L[c]	00001100	014	12	,	00101100	054	44		01001100	114	76		01101100	154	108
M[c]	00001101	015	13	-	00101101	055	45		01001101	115	77		01101101	155	109
N[c]	00001110	016	14	.	00101110	056	46		01001110	116	78		01101110	156	110
O[c]	00001111	017	15	/	00101111	057	47		01001111	117	79		01101111	157	111
P[c]	00010000	020	16	0	00110000	060	48		01010000	120	80		01110000	160	112
Q[c]	00010001	021	17	1	00110001	061	49		01010001	121	81		01110001	161	113
R[c]	00010010	022	18	2	00110010	062	50		01010010	122	82		01110010	162	114
S[c]	00010011	023	19	3	00110011	063	51		01010011	123	83		01110011	163	115
T[c]	00010100	024	20	4	00110100	064	52		01010100	124	84		01110100	164	116
U[c]	00010101	025	21	5	00110101	065	53		01010101	125	85		01110101	165	117
V[c]	00010110	026	22	6	00110110	066	54		01010110	126	86		01110110	166	118
W[c]	00010111	027	23	7	00110111	067	55		01010111	127	87		01110111	167	119
X[c]	00011000	030	24	8	00111000	070	56		01011000	130	88		01111000	170	120
Y[c]	00011001	031	25	9	00111001	071	57		01011001	131	89		01111001	171	121
Z[c]	00011010	032	26	:	00111010	072	58		01011010	132	90		01111010	172	122
[c]	00011011	033	27	;	00111011	073	59		01011011	133	91	/ [s]	01111011	173	123
\[c]	00011100	034	28	<	00111100	074	60		01011100	134	92		01111100	174	124
][c]	00011101	035	29	=	00111101	075	61		01011101	135	93	- [s]	01111101	175	125
^[c]	00011110	036	30	>	00111110	076	62		01011110	136	94	* [s]	01111110	176	126
_[c]	00011111	037	31	?	00111111	077	63		01011111	137	95	+ [s]	01111111	177	127

Notes

Notes

Notes

Personal Computer Division
1010 N.E. Circle Blvd., Corvallis, OR 97330 U.S.A.

Reorder Number
00087-90121

Printed in U.S.A. 1/8
00087-9026