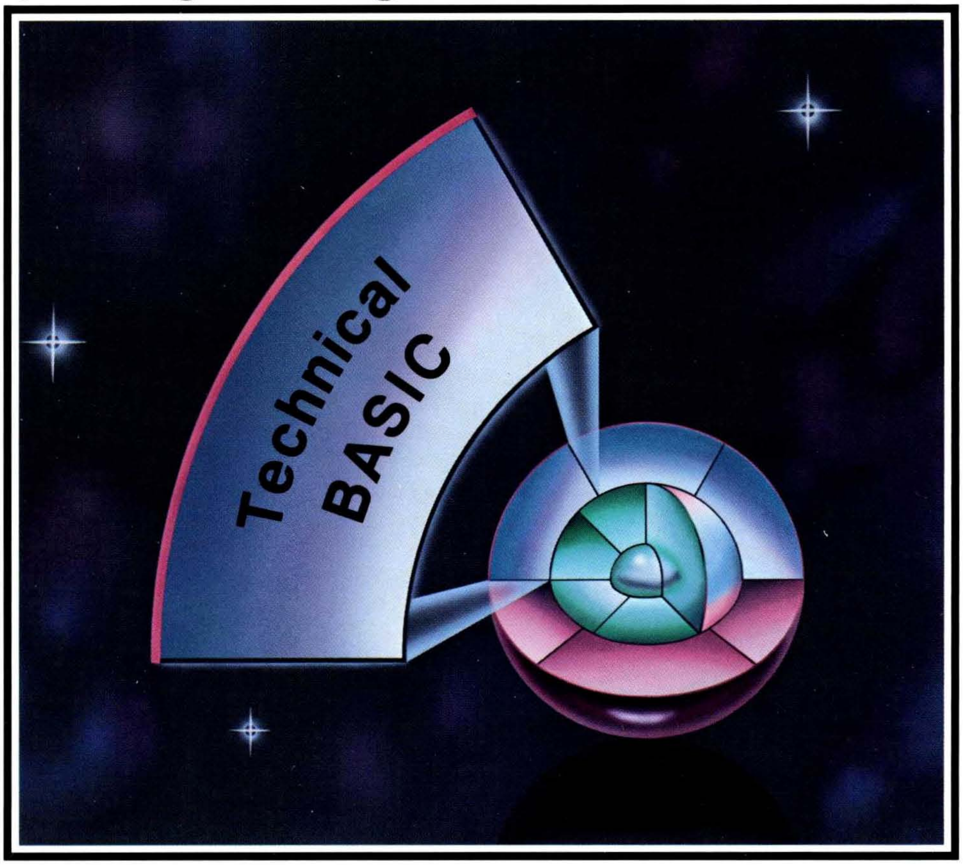
HP 9000 Computers

HEWLETT PACKARD

HP-UX Technical BASIC I/O Programming Guide



HP-UX Technical BASIC I/O Programming Guide

for HP 9000 Computers

HP Part Number 97068-90030

Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

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

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

February 1986...Edition 1

Table of Contents

An Introduction to I/O 1

This manual covers I/O programming techniques for HP-UX Technical BASIC. You can use this language on certain Hewlett-Packard computers that use the HP-UX operating system. For specific information on which computers support HP-UX Technical BASIC, ask your Hewlett-Packard dealer.

The term I/O stands for Input/Output, and refers to the process of communication between a computer and its resources. We shall use the term *computer* in this manual to mean the processor, its support hardware, the BASIC-language, and the operating system. The term *resource* refers to all of the "data-handling" elements of the system. Computer resources include the display, keyboard, interfaces, and peripheral devices.

HP-UX Technical BASIC provides more than 30 I/O statements. These statements can be used to establish communication between the computer and its resources. Some statements are used to send data to peripheral devices such as printers or to input data from peripheral devices such as voltmeters. Other statements are used to control the branching in an I/O program or to program the operation of the interfaces. This manual presents some selected programming techniques that show how you might use these statements in your own application programs. The *HP-UX Technical BASIC Reference Manual* provides syntax definitions of all of the BASIC statements, including the I/O statements.

Chapter Contents

This chapter covers the following tasks and topics:

The Role of an Interface

Before you can establish communication between the computer and its peripheral devices, you must establish a hardware link. An *interface* provides this hardware link. The role of the 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 function of an interface and its position between the computer and the peripheral device:

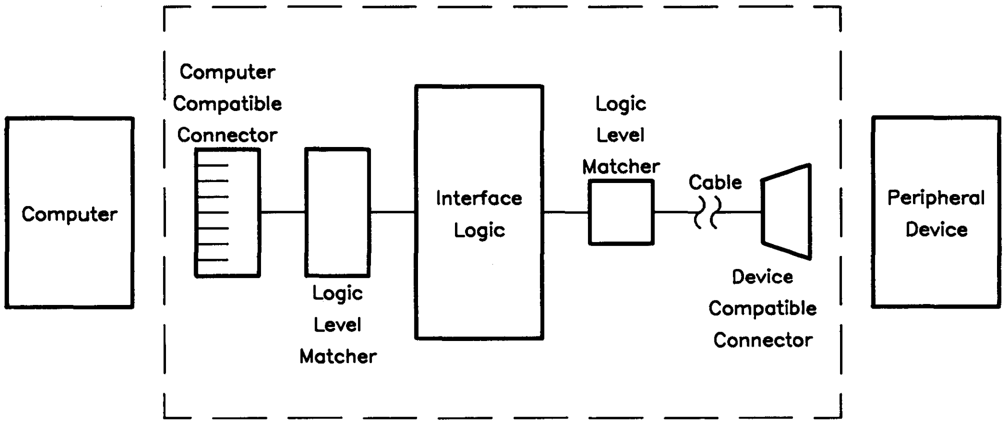


Figure 1-1. Functional Diagram of an Interface

Mechanical and Electrical Compatibility

Mechanical compatibility means that the plugs and connectors must fit together. Accessory interfaces are designed to plug directly into your computer or into a bus expansion module. Some interfaces, such as the HP-IB, are always mechanically compatible with their peripheral devices. Other interfaces, such as the GPIO, are supplied without peripheral connectors. For such an interface, you need to install a mechanically compatible connector (refer to the interface owner's manual).

Electrical compatibility means that the interface must change the voltage levels used by the computer to those used by the peripheral device. The drivers and receivers match logic levels to ensure that the interface is compatible with the computer and (usually) the peripheral devices. If you have questions about electrical compatibility with a particular device, refer to the manuals provided with the interface and peripheral device.

Data Compatibility

Mechanical and electrical compatibility alone does not guarantee that the computer and peripheral device will be able to communicate. Each device must also 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. To handle those cases where the computer performs the data translation or conversion, HP-UX Technical BASIC provides a wide variety of special functions and conversion capabilities. These capabilities are covered in subsequent chapters of this manual.

Timing Compatibility

Computers and their peripheral devices have such a wide range of operating speeds that an orderly mechanism is needed for the successful transfer of data. This timing mechanism is called handshaking. Although there are several types of handshaking, the typical sequence is 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 is 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, the process repeats.

The following simplified diagram further illustrates the concept of handshaking:

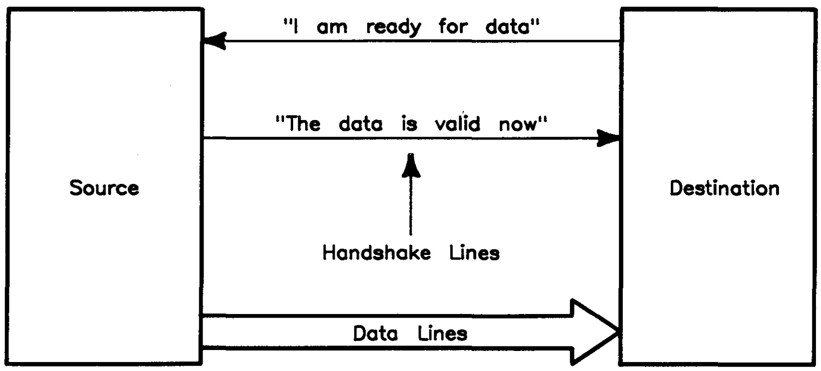


Figure 1-2. Handshaking

Specifying a Resource

An interface and associated cable(s) provide the hardware link for communication between your computer and its peripheral devices. To perform an I/O operation with a device connected to the interface (or with the interface itself), the system must be able to uniquely identify this resource. In this system, resources are identified by *device selectors* or *file selectors*.

Using Device Selector 1

Device selector 1 is pre-assigned to the CRT screen. This is the default destination for `DISP` and `PRINT` statements, since the defaults are `CRT IS 1` and `PRINTER IS 1`.

Thus, if you wanted to send some text to the display, you could use any of the following statements:

```
OUTPUT 1;"This goes to the display."
DISP  "This goes to the display."
PRINT "This goes to the display."
```

Device selector 1 uniquely specifies where the information is to be sent.

Assigning Your Own Device Selector

Since there are no pre-assigned device selectors for I/O devices other than the CRT, you will need to use the `ASSIGN` statement to create your own association between a device selector and a resource. Here is the general form of the `ASSIGN` statement used for I/O purposes:

`ASSIGN` *device selector* `TO` "*I/O driver type, special (device) file*"

- The *device selector* is an **arbitrary** integer in the range 3 through 10. (It should not currently be assigned to a resource.)
- The *I/O driver type* identifies the type of interface driver that is to be used with this resource. (A driver is a program that is used by the system to communicate with a particular device or interface.) On this system, the I/O driver type must be either `hpib` or `gpio`, since these are the only type of interfaces supported for general I/O operations.
- The *special (device) file* is the actual name of the driver file, which is assumed to be in the `/dev` directory—unless explicitly specified otherwise. This file must be a device file of the preceding I/O driver type. (This parameter is the name of the HP-UX file that the System Administrator associated with an interface or device using the `mknod`, "make node," command described on the following pages.)

Let's look at some examples for GPIO and HP-IB interfaces.

GPIO Device Selectors

In order to communicate with a GPIO device (or with the interface itself), you will need to assign a device selector to it. You will specify it by the I/O device type, **gpio**, and an appropriate device file. The following (equivalent) statements assign a *device selector* of 8 to a GPIO card:

```
ASSIGN 8 TO "gpio"
ASSIGN 8 TO "gpio,/dev/gpio"
```

Here is an example in which the device file is named **gpio_device** (since it is in the **/dev** directory, you need not specify its pathname):

```
ASSIGN 8 TO "gpio,gpio_device"
```

This assignment assumes that a node has been created for this device using the following HP-UX commands (which require *superuser* capabilities, usually possessed only by the System Administrator[1]):

```
# /etc/mknod /dev/gpio_device c 18 0x030000 Return
# chmod 666 /dev/gpio_device Return
```

where:

/dev/gpio_device	is the name of the special file to be created.
c	specifies "character" mode of operation on the interface or device.
18	specifies the *driver number* (also called the *major number*). For GPIO interfaces on Series 200/300 computers, this number is 22; on Series 500 computers, the GPIO driver number is 18.
0x030000	is the *minor number*, which contains the following fields: 0x indicates that the minor node number is specified using *hexadecimal* notation. 03 is the hardware *interface select code* of the GPIO interface. This is a number either set by switches on the interface itself (Series 200/300) or determined by the slot in which the interface is currently installed (Series 500). 0000 is always used for GPIO devices (since there is no addressing capability on this type of interface).
chmod 666	performs a "change mode" on the file access capabilities to allow read/write capabilities for user, group, and public use of the device file.

[1] For further explanation of special (device) files and associated major and minor node numbers, see the *HP-UX System Administrator Manual*.

You can check to see what the current device files are by using the HP-UX `ll` command. While in BASIC, type:

```
SHELL [Return]
```

You should now be in the HP-UX Bourne shell. Use the *long list* command to look at the directory that contains the device files currently set up. The following command searches for all device files with **gpio** in their names.

```
$ ll /dev/*gpio* [Return]
```

The system should return information similar to this for Series 500 systems:

```
crw-rw-rw-   1 root  other 18 0x030000  Feb 4  15:24  /dev/gpio_device
```

and similar to this for Series 200/300 systems:

```
crw-rw-rw-   1 root  other 22 0x080000  Feb 4  15:24  /dev/gpio_device
```

The results show that there is a device file called **gpio_device** that is currently associated with the GPIO interface at select code 3 (select code 8 for Series 200/300). If the device file's name did not contain the letters **gpio**, you could use the `ll` command and look for the appropriate major number—22 for Series 200/300 computers, and 18 for Series 500 computers.

When you are ready to return to the BASIC system, type [CTRL][D].

```
$ [CTRL][D]
```

BASIC responds as follows:

```
Basic ready
```

NOTE

The examples throughout this manual, except where otherwise noted, assume that device selector 8 has been assigned to the GPIO driver `/dev/gpio`.

HP-IB Device Selectors

For HP-IB device selectors, there are two types of nodes that you can use:

- Those **with primary addressing ("auto-addressed" mode)**—used to communicate with specific devices.
- Those **without primary addressing ("raw" mode)**—used to communicate with the interface. (Addressing can be supplied in the BASIC I/O statements to address specific HP-IB devices.)

Here is the general way of assigning a device selector to an HP-IB resource. The following (equivalent) examples assign a value of `3` (the *device selector*) to a resource with a *driver type* of **`hpib`** and *special (device) file* named **`/dev/hpib`**:

```
ASSIGN 3 TO "hpib,/dev/hpib"
ASSIGN 3 TO "hpib"
```

This assignment assumes that a node has been created for this device, but it does not indicate whether the "raw" mode or the "auto-addressed" mode is being used.

An HP-IB node without addressing ("raw" mode) is created using the following HP-UX commands (which require *superuser* capabilities, usually possessed only by the System Administrator[1]):

```
# /etc/mknod /dev/hpib  c  12  0x021f00 [Return]
# chmod 666 /dev/hpib [Return]
```

where:

`/dev/hpib`	is the name of the special file to be created.
`c`	specifies "character" mode of operation.
`12`	specifies the *driver number* (also called the *major number*). For HP-IB interfaces on Series 200/300 computers, this number is 21; on Series 500 computers, the HP-IB driver number is 12.
`0x021f00`	is the *minor number*, which contains the following fields: `0x` indicates that the minor node number will be specified in *hexadecimal* notation. `02` is the hardware *interface select code* of the HP-IB interface. This is a number set either by switches (Series 200/300) or is determined by the slot into which the interface is currently installed (Series 500). `1f` is a value that specifies that the node is in "raw" mode. (If a number other than `1f` is specified, such as `01`, then this number identifies the *HP-IB primary address* of the device.) `00` is always used for HP-IB interfaces and devices.
`chmod 666`	performs a "change mode" on the file access capabilities to allow user, group, and public to use the device file.

[1] For further explanation of special (device) files and associated major and minor node numbers, see the *HP-UX System Administrator Manual*.

You can check to see what the current device files are by using the HP-UX *long list* (`ll`) command. While in the BASIC system, type:

```
SHELL [Return]
```

which should take you to the HP-UX Bourne shell. Now you can search for the device files with `hpib` in their name:

```
$ ll /dev/*hpib* [Return]
```

The system should respond with the following information (for Series 500 systems):

```
crw-rw-rw-   1 root  other 12 0x021f00  Feb 4  15:18  /dev/hpib
```

The results show that a device file called `hpib` is currently associated with the HP-IB interface at select code 2. If the device file's name did not contain the letters `hpib`, you could use the HP-UX `ll` command and look for the appropriate major number—21 for Series 200/300 systems, and 12 for Series 500 systems. The `1f` shows that there is no primary address associated with this device file—instead, the interface will be used in "raw" mode.

To return to BASIC, type:

```
$ [CTRL][D]
```

BASIC should notify you that you have returned:

```
Basic ready
```

Now you can `ASSIGN` a device selector to the device file, and then use the device selector in I/O statements. For instance:

```
ASSIGN 7 TO "hpib,/dev/hpib"
OUTPUT 701;"This is sent to the HP-IB device at address 01."
ENTER 722;Voltage
STATUS 7,1;Register_1
CONTROL 7,2;SRQ_line
```

Note that when using a device file in "raw mode," you can specify the primary address of HP-IB devices, such as in the above `OUTPUT 701;...` and `ENTER 722;...` statements.

An HP-IB node with an address ("auto-addressed" mode) is created using the following HP-UX commands (which require *superuser* capabilities, usually possessed only by the System Administrator[1]):

```
# /etc/mknod /dev/hpib_01  c  12  0x020100 Return
# chmod 666 /dev/hpib_01 Return
```

With a device selector assigned to this type of node, you could **not** use any primary addressing in I/O statements that use this device selector. For instance, suppose that you search the /dev directory and find the following node (created with the preceding commands):

```
SHELL Return
$ ll /dev/hpib* Return
crw-rw-rw-   1 root  other 12 0x020100  Feb 4  15:22  /dev/hpib_01
$ CTRL D
Basic ready
```

The device file /dev/hpib_01 is also associated with the HP-IB interface at select code 2, but there is also a primary address included—01. Thus, you could only use statements such as the following (which do not specify any primary addressing information):

```
ASSIGN 3 TO "hpib,/dev/hpib_01"
OUTPUT 3;"This goes to select code 2, address 01."
ENTER 3;Value_from_201
```

Note that the CONTROL and STATUS statements will not work, since they require that the device selector be assigned to a "raw" node.

You can use whichever method works best for your purposes.

NOTE

The examples throughout this manual, except where otherwise noted, assume that device selector 7 has been assigned to the HP-IB device file named /dev/hpib. Also, the device file is assumed to be used in "raw" mode—no primary addressing—as shown in the first example above.

[1] For further explanation of special (device) files and associated major and minor node numbers, see the *HP-UX System Administrator Manual*.

Using File Selectors

You can also use the ASSIGN statement to assign a *file selector* to an HP-UX file.

```
ASSIGN 11 TO "TextFile"
OUTPUT 11;"Text for the file."
OUTPUT 11;"12345.67"
ASSIGN 11 TO "*"

ASSIGN 11 TO "TextFile"
ENTER 11;StrVar$
DISP StrVar$
ENTER 11;StrVar$
DISP StrVar$
```

Refer to the "Data Storage and Retrieval" chapter of the *HP-UX Technical BASIC Programming Guide* for further examples, or to the *HP-UX Technical BASIC Reference Manual* for the complete syntax of the ASSIGN statement.

Simple I/O Operations 2

The `PRINTER IS` and `PRINT` statements, which are described in the *HP-UX Technical BASIC Programming Guide*, provide you with simple output operations. However, these operations fall short of the mark in many circumstances. The most obvious shortcoming is that there is no corresponding statement to allow you to enter data from external devices. The `PRINT` statement is convenient when you simply want to output data to a previously defined printer. However, `PRINT` becomes cumbersome when you want to communicate with multiple devices, since you must specify `PRINTER IS` each time you want to output to a different device.

The principal tools for using interfaces to move data into and out of the computer are the `OUTPUT` and `ENTER` statements. These statements are the *core* of I/O operations. They are usually the fastest and most convenient 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`.

Chapter Contents

This chapter covers the following tasks and topics:

Simple `OUTPUT` and `ENTER` statements (described in this chapter) 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. ASCII provides a standard correspondence between binary codes that are easily understood by the computer and alphanumeric symbols that are easily understood by humans.

The chapter "Formatted I/O Operations" covers the `OUTPUT USING` and `ENTER USING` statements. These forms are very convenient if you need special formatting.

NOTE

Before you can use the `OUTPUT` or `ENTER` statements, you must assign device selector numbers to each interface (refer to the chapter "An Introduction to I/O").

Using Simple OUTPUT Statements

You can use a simple OUTPUT statement 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 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 701 ; "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 that commas or semicolons may 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. After the last item in the output list has been output, an end-of-line sequence is output. You can suppress this final end-of-line sequence by following the last item in the output list with a semicolon.

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 and PRINT statements both use the same fields. The semicolon calls for a compact field, while the comma produces free field. These fields are summarized in the following table.

Table 2-1. Data Field

	Numeric Data	String Data
Compact Field (;)	Digits of the number are output, preceded by a space (if positive) or a minus sign (if negative), and followed by one space.	Characters of the string are output with no leading or trailing spaces.
Free Field (,)	Digits of the number (with leading space or minus sign) are output left justified in a field of 21 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.

Free field format divides the current line length into 21-column fields. The fields begin in columns 1, 22, 43, and 64 if the 80-column default line length is in use. 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 chapter "Formatted I/O Operations."

Using Simple ENTER Statements

A simple ENTER statement may be used to enter data 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$
ENTER 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 to remove much of the mystery from I/O. Suppose that 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 device selector 3:

M	O	N	D	A	Y		J	U	N	E		1	1	,		1	9	8	4	cr	lf

The computer ignores all leading spaces and non-numeric characters, so the MONDAY JUNE characters do nothing. Then the 11 is read. Once the computer has started to read a number, a space or non-numeric character signals the end of that number. Therefore, the comma after the 11 causes the computer to place the value eleven into variable X and start looking for the next value. The space in front of 1984 is ignored and the computer reads the 1984. The carriage-return character causes the computer to place the value nineteen hundred eighty four 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=1984.

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

1. Ignores leading spaces and non-numeric characters.
2. Uses numeric characters to build a numeric value.
3. Terminates the building of a value when an embedded or trailing space or non-numeric character is encountered.
4. 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 through 9 are always numeric characters. Also, the decimal point, plus sign, minus sign, and the letter E can be numeric if they occur at a meaningful place in a number. For example, assume that the following character sequence is read by an ENTER statement:

-	-	T	E	S	T		1	2	.	5	E	-	3

If a numeric value is being entered, the leading minus signs and the E in TEST will be ignored. They have no meaningful numeric value when surrounded by non-numeric characters. However, the characters 12.5E-3 will be interpreted as 12.5x10-3. 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 705 ; 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$. Since neither the carriage return or the line feed is placed into B$, 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, command entry or program execution (if running a program) continues on 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$[3]
ENTER 705 ; X$
```

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$, which fills the dimensioned length of 3. The computer continues to read (and "throw away") the incoming characters until a line feed is encountered. The line feed character terminates the ENTER statement, and the computer goes on to the next program line with X$="BOY".

Formatted I/O Operations 3

Although free-field format works well for some I/O operations, 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 are many reasons for desiring format controi during I/O operations.

The format of information sent or received through interfaces is controlled by the use of `OUTPUT` or `ENTER` *images*. These images consist of *image specifiers*, and can be either placed in an `IMAGE` statement or included directly in an `OUTPUT...USING` or `ENTER...USING` statement.

Chapter Contents

This chapter covers formatted I/O operations. It includes the following tasks and topics:

Tasks/Topics	Page
Formatted OUTPUT	3-2
Formatted ENTER	3-9
Converting I/O Data	3-19

Formatted OUTPUT

When you use the `OUTPUT...USING` statement, the data is output according to an `OUTPUT` *format image*. The `OUTPUT` image consists of one or more individual image specifiers that describe the type and number of data bytes to be output. You can include the `OUTPUT` image directly in the `OUTPUT` statement or you can reference it by the name of a string variable. You can also include an `OUTPUT` image in an `IMAGE` statement. The `OUTPUT...USING` statement can then refer to the `IMAGE` statement by its line label or line number. The following statements are examples of these four possibilities:

1.
   ```
   100 OUTPUT 701 USING "SDDD.DD" ; 123.45
   ```

2.
   ```
   100 A$="SDDD.DD"
   110 OUTPUT 701 USING A$ ; 123.45
   ```

3.
   ```
   100 FMAT: IMAGE 6A,SDDD.DD
   110 OUTPUT 701 USING FMAT ; B$,Y
   ```

4.
   ```
   100 IMAGE 6A,SDDD.DD
   110 OUTPUT 701 USING 100 ; B$,Y
   ```

Notice that in each case a destination device is specified with a device selector, and the items to be output are included in an output list. The output list is preceded by a semicolon, but it does not matter whether commas or semicolons are used to separate items in the output list. The output format is controlled by the output image. The complete syntax of the `OUTPUT` statement, including the `OUTPUT USING` form, is covered in the *HP-UX Technical BASIC Reference Manual*. The rest of this section explains the effect of the `OUTPUT` image specifiers. These image specifiers are used not only by `OUTPUT USING`, but also by `DISP USING` and `PRINT USING`.

Numeric Images

Numeric images are used to control the form of numbers to be output. Three kinds of image specifiers are used in numeric images:

- digit specifiers
- sign specifiers
- punctuation specifiers

Digit Specifiers

These are the image specifiers that form the digits of a number to be output. They allow you to determine the number of digits before and after the decimal point, display or suppress leading zeros, and control the output of exponent information.

Table 3-1. OUTPUT Digit Specifiers

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 Z, except leading zeros are replaced with 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. Digits are output, preceded by a space (if positive) or a minus sign (if negative), and followed by one space.

NOTE

Only the D, E, and e specifiers may be used after the radix symbol.

Sign Specifiers

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

Table 3-2. `OUTPUT` Sign Specifiers

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 Specifiers

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

Table 3-3. `OUTPUT` Punctuation Specifiers

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 the numeric image specifiers and the resulting output when numbers are sent to a typical printer:

Table 3-4. Combining the Numeric Image Specifiers

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 "***Z.DD" ; 0.336`	`***0.34`
`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 "SZ.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 placing that number in front of the specifier. The use of parentheses, as in 3(D), changes the meaning. The image 3D means "output one numeric quantity in a three-digit field." The image 3(D) means "output three numeric quantities, each in a one-digit field."

Be careful of overflow conditions when using these image specifiers. An overflow occurs when the number of digits required to accurately represent a number is greater than the number of digits specified 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.

The above examples all show numeric constants in the output list. You may, of course, include numeric variables in the output list. For example, if X=12.123, the statement:

```
OUTPUT 701 USING "ZZZZ.DD" ; X
```

will result in the printed output 0012.12.

String Images

The image specifiers in this group deal with the output of string characters. They may be used in combination with the numeric image specifiers for spacing and labeling purposes.

Table 3-5. OUTPUT String Specifiers

Image Specifier	Meaning
`A`	Causes the output of one string character. If all 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 quotation marks, by using the CHR$ function, or by a combination of the two. When a literal image is encountered, the specified character sequence is output. When the literal is enclosed in quotes, the quotation 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 to use string image specifiers. The resulting output is shown for a typical printer.

Table 3-6. Using String Image Specifiers

Statement	Printed Output
`OUTPUT 701 USING "5A,A" ; "X","Y"`	`XY`
`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 `OUTPUT` image (usually to label the output). If literal (string constant) images are desired, they must be placed in an `IMAGE` statement. Other `OUTPUT` images may either be placed directly in an `OUTPUT` statement, or in an `IMAGE` statement.

Binary Images

These image specifiers are used to cause information to be output as one or two binary bytes, rather than as a character representation. The items to be output using binary 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.

Table 3-7. OUTPUT Binary Specifiers

Image Specifier	Meaning
B	Outputs a value as a single 8-bit byte. The value must be in the range 0 through 255. If the value 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 -32,768 to +32,767. Negative numbers are output in 16-bit twos complement form. If the value is out of range and positive, 32,767 is output. If the value is out of range and negative, -32,768 is output.

The following examples show how binary images may be used.

Table 3-8. Using Binary Image Specifiers

Statement	Interface output (Bit Pattern)
`OUTPUT 8 USING "B" ; 127`	01111111
`OUTPUT 8 USING "B" ; 3`	00000011
`OUTPUT 8 USING "W" ; 3`	00000000 00000011
`OUTPUT 8 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.

End-of-Line Sequence Images

These image specifiers control the output of end-of-line sequences. An end-of-line sequence may consist of one or more characters that are normally output after the last item in an output list and/or a signal on an interface line concurrent with the last byte output. Exactly what sequence or signal is used depends upon the programming of the interface. Refer to the appendix for information on selecting an end-of-line sequence. If your program does not change the end-of-line sequence, the default is a two-character sequence: a carriage return followed by a line feed.

The following images do not alter the end-of-line sequence. They just control whether or not it is output.

Table 3-9. `OUTPUT` EOL Sequence Specifiers

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 / specifier 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 # specifier 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 embedded 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 8 USING "#,B" ; X
```

This statement outputs the binary representation of `X` with no carriage return, line feed, or 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

Formatted ENTER

The ENTER USING statement is used to enter data from an interface into the listed variables using the format image referenced by the USING secondary keyword. The ENTER image, like the OUTPUT image, consists of one or more image specifiers that describe the type and number of data bytes to enter. The ENTER image can be included directly in the ENTER statement or it can be referenced by the name of a string variable. The ENTER image can also be included in an IMAGE statement. The ENTER USING statement can then refer to the IMAGE statement by its line label or line number. Examples of these four possibilities are listed below:

1.
   ```
   100 ENTER 701 USING "SDDD.DD" ; X
   ```

2.
   ```
   100 A$="SDDD.DD"
   110 ENTER 701 USING A$ ; X
   ```

3.
   ```
   100 FMAT: IMAGE 6A,SDDD.DD
   110 ENTER 701 USING FMAT ; B$,Y
   ```

4.
   ```
   100 IMAGE 6A,SDDD.DD
   110 ENTER 701 USING 100 ; B$,Y
   ```

Each ENTER statement specifies an interface or device as the source, and a list of variables (string or numeric) as the destination of the data to be entered. The variables in the enter list are separated with commas and the list is preceded by a semicolon. The entry of data into the listed variables is controlled by the ENTER image. ENTER images are analogous to OUTPUT images, but consist of ENTER image specifiers. These specifiers differ in function from the OUTPUT image specifiers, and are covered in the remainder of this section. The complete syntax of the ENTER statement is covered in the *HP-UX Technical BASIC Reference Manual.*

ENTER images consist of *data images* and *terminator images*. These images give you a high degree of control, respectively, in the following areas:

1. Accurately describing to the computer what the incoming data looks like and what should be done with it.
2. Precisely specifying what conditions terminate an entry into a variable and what conditions terminate the ENTER statement itself.

Data Images

The image specifiers in this group are used to indicate what the computer should 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.

Table 3-10. ENTER Numeric Specifiers

Image Specifier	Meaning
D Z * . S M	These specifiers all accept one character to be used to build a number. The incoming characters do not have to follow the specified format, but the number of characters must be correct. 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.*
E	Accepts five characters to be used to build a number. The five characters need not be exponent information, but they can be.
e	Same as E, but four characters are accepted to be used to build a number.
C	This specifier also accepts one character to be used to build a number. However, if a C is present anywhere in a number's image, all commas will be ignored while the number is being entered. Without this specifier, a comma would end numeric input.
K	Enters a numeric or string variable using free field format (explained in chapter 2).
*	Only the D, E, and e specifiers may be used after the radix symbol.

NOTE

The R and P specifiers, used with the OUTPUT statement to provide a European radix point and digit separator, may *not* be used with the ENTER statement. If you need to enter European format numbers, you can use the CONVERT statement (covered later in this chapter) to change the number into American format.

String Image Specifiers

These specifiers are used to enter characters into string variables.

Table 3-11. ENTER String Specifiers

Image Specifier	Meaning
A	Accepts one character to enter into a string variable.
K	Enters a string or numeric variable using free field format (explained in the chapter "Simple I/O Operations").

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 "*Z.D,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 nA image if n (the number of characters) is known, or with a K if the number of characters is not known.

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.

Table 3-12. `ENTER` Binary Specifiers

Image Specifier	Meaning
`B`	Accepts one byte of binary data and enters its equivalent decimal value into a numeric variable.
`W`	Accepts two bytes of binary data to be used to build a 16-bit twos 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.

Suppose that the computer receives two bytes of binary information: "00001000" followed by "00001010". These two bytes could be read with either of the following statements:

```
ENTER 8 USING "B" ; X,Y
ENTER 8 USING "W" ; Z
```

In the first statement the `B` image is used to read the first byte ("00001000"). Its decimal equivalent (8) is entered into the variable `X`. The `B` image is then used to read the second byte ("00001010") and enter its decimal equivalent (10) into `Y`.

The second statement demonstrates how the `W` image differs from `B`. The `W` image reads the two bytes as a 16-bit twos complement word ("0000100000001010"). The decimal equivalent of that word (2058) is entered into the variable `Z`.

Skipping Unwanted Characters

The following specifiers can be used with incoming numeric or string data to skip over any unwanted characters.

Table 3-13. ENTER Character Skipping Specifiers

Image Specifier	Meaning
X	Causes one character to be skipped.
/	Causes all characters to be skipped until the next line feed character 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, X gets the value 123 and Y$ becomes BYE. The / specifier causes the computer to skip all characters after X was satisfied until the line feed is received. Entry into Y$ begins with the first character after the line feed. Without the / specifier, entry into Y$ begins as soon as the 3D field is satisfied.

Terminator Images

Terminator images allow you to precisely define the condition or conditions that terminate entry into a variable (field termination) and/or terminate the ENTER statement itself (statement termination). Before we discuss the terminator image specifiers, let's see just what is meant by field and statement termination.

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*. An incoming record often consists of multiple *fields*. A *field terminator* terminates entry of a data field into a variable. Consider the entry of the following record:

A	B	C	lf	D	E	F	lf	G	H	I	cr	lf

The record consists of three fields and could be read with the following ENTER statement:

```
ENTER 720 USING "K" ; A$,B$,C$
```

The K image reads each field into a variable in free-field format. ABC is entered into variable A$, then the first line feed is encountered. The line feed acts as a field terminator and ends the entry of data into A$. DEF is entered into B$, and the second line feed also acts as a field terminator. Finally, GHI is entered into C$, and the carriage return/line feed terminates both the field and the ENTER statement.

NOTE

To allow a carriage return/line feed sequence as a statement terminator, ENTER ignores a carriage return if it is immediately followed by a line feed. A carriage return will be entered into the string if it is not followed by a line feed.

In the above example it was not necessary to specify a terminator image. In most cases an ENTER statement will function properly without a terminator image because the system has built-in field terminators and a default statement terminator. These normal terminators are:

- A string field ends when the string is full (check your DIM statements), the character count in an image field is satisfied, or a line feed is received.
- A numeric field ends when a space or non-numeric character is encountered, or the character count from an image field is satisfied.
- 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.

Eliminating the Line Feed Requirement

Normally, 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 use a terminator image to allow the ENTER statement to terminate properly. You can use the # specifier to eliminate the requirement for a line feed to terminate the ENTER statement. The ENTER statement terminates when 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 ENTER image. For example:

```
ENTER 8 USING "#,K" ; A$
ENTER 720 USING "#,4D,6D" ; X,Y
```

The first 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 entering ten characters. This use of the # specifier is the simplest (and most common) terminator image application.

Advanced Use of Terminator Images

The ENTER image specifiers discussed thus far are sufficient to handle the great majority of requirements. However, there are some special situations that demand even greater flexibility. Most of these special cases involve the EOI (End or Identify) line on the HP-IB. The following discussion is of no concern to most programmers. However, if you have an application that involves the EOI line, or if you have an unusual problem with line feeds, then read carefully.

The following image specifiers can be used to modify both field and statement terminating conditions.

Table 3-14. ENTER **Termination Specifiers**

Image Specifier	Modifying Field Termination	Modifying Statement Termination
#	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 EOI must be received to terminate the statement, and that line feed is not a terminator.

Each of these image specifiers can control either field termination or statement termination depending upon its position in the ENTER image. The following statement illustrates this:

```
10 ENTER 720 USING "#,#K" ; A$
```

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

The use of the # specifier to eliminate the requirement for a line feed statement terminator has already been discussed (refer to "Eliminating the Line Feed Requirement"). The #K image in the above statement eliminates the line feed character as a field terminator. All incoming characters (including line feeds) are entered into A$ until the string is filled. If A$ has a dimensioned length of 10, the first 10 characters are entered. This satisfies the variable A$ and also terminates the ENTER statement.

The statement:

```
20 ENTER 720 USING "#K" ; A$
```

also enters all incoming characters (including line feeds) into A$ until the string is full. However, this statement requires a line feed statement terminator. Once A$ is full, this statement continues to read (but "throw away") incoming characters until a line feed is encountered.

If you need to terminate fields and/or the statement with the EOI signal, use the % specifier:

```
30 ENTER 720 USING "%,%K" ; A$
```

The %K image causes the computer to terminate a free field string entry with the EOI signal. The initial % specifies that the statement may be terminated with an EOI signal (or with a line feed character).

The statement:

```
50 ENTER 720 USING "%,#%K" ; A$
```

allows line feed characters to be entered into the string, the field being terminated by the EOI signal. The statement will terminate on either the EOI signal or a line feed.

The image %K can also be used to terminate a free field numeric entry with the EOI signal. The statement:

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

enters numeric data into X until an EOI signal is received. Numeric data is then entered into Y until a second EOI signal is received, which terminates both the field and the statement.

The preceding examples cover only a few of the possible uses of terminator images. You may find that some other combination of image specifiers suits your particular application. Note that the built-in field terminators are always in effect. A string field always ends if the string is filled. A numeric field always ends if a non-numeric character is received. Any field is terminated when the character count from an image field is exceeded.

There Is Always an Exception

Not all terminator problems can be solved with terminator images. For example, suppose that you want to enter a name field (string) followed by an age field (numeric). The names are variable in length and separated from the age by a comma. If the ages came first, this would not be a problem since the comma would terminate entry into the numeric variable. The task is a bit trickier because the string data is entered first. You might be able to use a CONVERT statement (explained in the end of this chapter) to change the comma into a line feed, then use the line feed to terminate the string. If the application does not permit the blanket conversion of commas to line feeds, 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.

Inspecting Unknown Data

The choice of a proper format image is critical to the success of any formatted I/O application. The choice of a proper OUTPUT image is only a matter of definition. You know what data is generated by your program, so you need only choose a desirable form for its output. The main problem is to avoid image overflow conditions.

The choice of an ENTER image is made more difficult by the fact that you cannot determine the exact nature of the incoming data if you cannot get it into the computer to study it. 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 decimal equivalent of the binary value of each byte will be entered. You can use the CHR$ function to determine the exact character sequence that is being received. Then, knowing the exact nature of the incoming data, you can choose an appropriate ENTER image. The following program is an example of this technique:

```
100 ! Program to inspect incoming data
110 ASSIGN 8 TO "gpio"
120 S1=8 ! Device selector for the gpio interface
130 ! Establish a terminating condition
140 SET TIMEOUT S1;3000
150 ON TIMEOUT S1 GOTO 240
160 !
170 I=1 ! Initialize counter
180 ! Input 1 byte; display analysis
190 ENTER S1 USING "#,B" ; X
200 DISP "BYTE";I;TAB(11);"VALUE =";X;TAB(24);"CHAR = ";CHR$(X)
210 I=I+1 ! Count bytes
220 GOTO 190
230 !
240 DISP "DATA INPUT HAS STOPPED"
250 RESET S1 ! Stop I/O operation
260 END
```

Converting I/O Data

The final form of I/O data formatting involves changing the characters that are entered or output. The `CONVERT` statement allows you to convert characters into other characters during an `ENTER` or `OUTPUT` operation. The syntax of the `CONVERT` statement is covered in the *HP-UX Technical BASIC Reference Manual*. Some examples of the `CONVERT` statement follow:

1. `CONVERT IN 7 PAIRS ; A$`
2. `CONVERT OUT 7 PAIRS ; A$`
3. `CONVERT IN 7 INDEX ; A$`

These examples show that the conversion can take place during an `ENTER` operation (specify `IN`) or during an `OUTPUT` operation (specify `OUT`). You can also specify that the conversion table be accessed by either the `PAIRS` or the `INDEX` method. The actual conversion table must be contained in a string variable (`A$` in the above examples).

One problem that requires the use of `CONVERT` is the entry of European format numbers (with periods separating digit groups and a comma for the radix point). There is no `ENTER` image to accept this type of data directly, but the `CONVERT` statement can be used to convert commas to periods and periods to spaces. The statements for doing this with an interface at device selector 7 are:

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

For example, "12.345,6" is converted to "12 345.6" by the above statements.

The statement:

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

could be used to read the number into variable `X` (the conversion is in effect for any data entered through the interface at device selector 7). The converted data can be entered without an `ENTER` image since free-field format ignores spaces within a number and recognizes a decimal point. 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 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 following statement cancels the conversion:

```
CONVERT IN 7
```

Giving only the direction and device selector number, 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 to convert a carriage return to a line feed. This conversion is needed when entering data from a device that gives only a carriage return, without a line feed, as a delimiter.

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

The following program is an example of using the secondary keyword PAIRS with the CONVERT OUT command. In this program, the string ConStr$ is used as a conversion table for the CONVERT OUT statement. Notice that for each upper-case character in the string ConStr$ there is a lower-case character paired with it. The upper-case letter in the output string is then converted to the lower-case value of the character it was matched with in the ConStr$ string.

```
100 DIM ConStr$[52]
110 ConStr$="AaBbCcDdEeFfGgHhIiJjKkLl"
120 CONVERT OUT 1 PAIRS ; ConStr$
130 OUTPUT 1 ; "SOME UPPERCASE LETTERS."
140 END
```

Executing the above program will produce the following results:

```
SOMe UPPeRcaSe leTTeRS.
```

Using I/O Buffers 4

The previous two chapters have covered the OUTPUT and ENTER statements, and their use in conducting data transfers to and from *devices*. This chapter provides a discussion on conducting data transfers to and from *I/O buffers*.

Chapter Contents

Data transfers can be made to and from I/O buffers. This chapter covers how these data transfers are made. It includes the following tasks and topics:

What are I/O Buffers?

A buffer is a section of read/write memory set aside for the purpose of temporary data storage. I/O buffers can be used as the source of data for ENTER, and as the destination for OUTPUT.

The IOBUFFER Statement

Before you can OUTPUT to or ENTER from an I/O buffer, you must declare it with the IOBUFFER statement. You may declare up to 10 I/O buffers. To declare an I/O buffer, dimension a string variable to the desired length, then execute an IOBUFFER statement for that variable. To dimension the string variable Z$ to a length of 100 characters, then declare it to be an I/O buffer, use the following statements:

```
DIM Z$[100]
IOBUFFER Z$
```

Once you have declared an I/O buffer, you may use an ENTER or OUTPUT statement to exchange data with the I/O buffer and the program variables. The figure shown below illustrates how ENTER, OUTPUT, conversion, formatting, and I/O buffers work together.

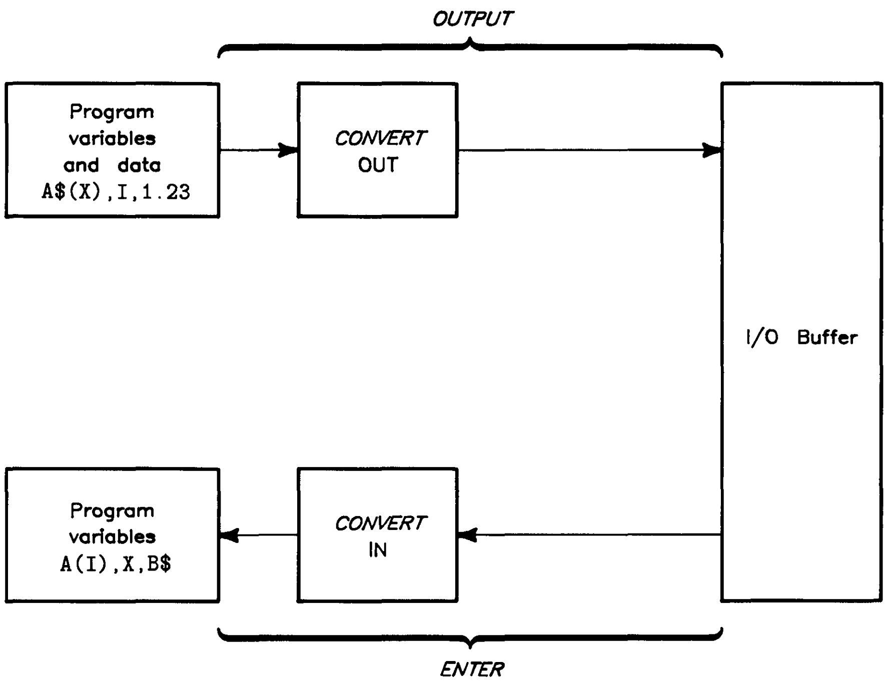

OUTPUT
Program variables and data A$(X),I,1.23
CONVERT OUT
I/O Buffer
Program variables A(I),X,B$
CONVERT IN
ENTER

Figure 4-1. Data Exchange with the I/O Buffer

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. Note that default OUTPUT operations to a buffer place a carriage return and a line feed at the end of the data. This means that the buffer should be dimensioned to a length greater by two when using the default OUTPUT operation.

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, you can avoid errors by using the form:

```
ENTER Z$ USING "#,#K";A$
```

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

Registers

When you declare a string variable to be an I/O buffer (by executing the IOBUFFER statement), two registers are created for that buffer. Registers 0 and 1 contain the buffer pointers. You can read these registers with the STATUS statement and write to them with the CONTROL statement (refer to the chapter "Direct Interface Communication").

Pointers

Register 1 contains the *fill pointer*. The *fill pointer* is initially equal to zero and indicates the number of characters written into the buffer (the same value as that returned by the LEN function). Placing a character into the buffer proceeds as follows:

- increment the fill pointer,
- store the character.

This operation is automatically handled by the OUTPUT 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.

Register 0 contains the *empty pointer*. The *empty pointer* is initially equal to one and indicates the number of characters that have been read from the buffer. Taking a character from the buffer proceeds as follows:

- read the character,
- increment the empty pointer.

This operation is performed automatically by the ENTER statement.

A buffer is full when the *fill pointer* equals the string's dimensioned length. Any OUTPUT operation to a full buffer will result in an error. A buffer is empty when the *empty pointer* is equal to the *fill pointer* plus one (regardless of the dimensioned length of the string). When a buffer is emptied, the *fill pointer* is reset to 0 and the *empty pointer* is reset to 1. This is the same initial condition as that set by executing the IOBUFFER statement, except that IOBUFFER also initializes conversion table pointers.

Buffer Status and Control

You can monitor the status of the buffer pointers, by reading the appropriate registers with the STATUS statement. The following table shows the appropriate statements to read the status of the I/O buffer Z$:

Table 4-1. I/O Buffer Status

Register	Default Value	Register Function	STATUS Statement
0	1	Empty pointer	`STATUS Z$,0;T0`
1	0	Fill pointer	`STATUS Z$,1;T1`

You may read these registers at any time for an I/O buffer. However, an error results if you attempt to read the status of a string variable that has not been declared as an I/O buffer.

You can assign new values to the empty pointer and fill pointer by writing to Registers 0 and 1 with the CONTROL statement. This gives you the capability of sending the same data over and over again without having to recompute the data. The following table shows the I/O buffer control registers and how they are accessed.

Table 4-2. I/O Buffer Control

Register	Default Value	Register Function	CONTROL Statement
0	1	Empty pointer	`CONTROL Z$,0;`*value*
1	0	Fill pointer	`CONTROL Z$,1;`*value*

You can write a new buffer pointer value (represented by *value* in the above statements) to these registers at any time, but an error will result if you execute a `CONTROL` statement to a non-buffer string variable.

You can do several operations by exercising control over the buffer empty and fill pointers. Some typical applications are:

- Retransmit data.
- Transmit any portion of the data in the buffer.
- Write data into any section of the buffer.
- Read data out of any section of the buffer.

Notes

Interrupt Programming 5

An interrupt to a computer is very much like a telephone ringing while you are working. It is a means of diverting your attention from whatever you are doing. "Servicing" an interrupt is similar to the act of answering 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, you can disable or enable that interrupt by setting the switch. If your telephone has more than one line, you can enable or disable more than one interrupt. The computer has essentially the same facility.

The computer has two types of interrupts to deal with:

- *low-level*, or hardware interrupts,
- *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. The software interrupt, or *end-of-line branch*, is provided so that you can write custom service routines for interrupts that are specific to your system.

The effect of an end-of-line branch is to check, at the end of each program line, for an *event*. The event may be an error, a hardware interrupt, an interface timeout, a function key-press, or a timer timeout. When one of these events occurs, a special portion of the program the *service routine* is executed to deal with the event. The programmer can enable or disable the end-of-line branch.

Chapter Contents

This chapter deals with end-of-line branching on interrupts. It covers the following tasks and topics:

End-of-Line Branch Programming

The rest of this chapter shows how you can write service routines to deal with end-of-line branches for various I/O related events. Refer to the *HP-UX Technical BASIC Reference Manual* for the complete syntax of the statements and functions covered in the following discussion.

Timeouts

If an external device is turned off, becomes disconnected, or otherwise fails to send or receive data during an `ENTER`, `OUTPUT`, or `SEND` operation, the result is a loss of handshake. (Handshaking is a means of reliably transferring data between two devices. Refer to the chapter "An Introduction to I/O.") Loss of handshake during an `ENTER`, `OUTPUT`, or `SEND` operation prevents data from being transferred and "hangs" the program until the operator becomes aware that something has gone wrong. This is not acceptable for most I/O applications, especially those where unattended operation is frequent.

The **SET TIMEOUT** statement allows you to establish a maximum time period for the computer to wait on an interface to handshake data. If an interface exceeds the timeout period, your program can follow either of two courses of action. The default method just aborts the I/O operation in progress and continues program execution at the next line. The other method uses an ON **TIMEOUT** statement to execute a timeout service routine after the I/O operation is aborted. As part of the syntax for the ON **TIMEOUT** statement, it is necessary to specify where the service routine for an event is, and also whether the routine is a subroutine (**GOSUB**) or a program segment (GOTO). Note that the computer must know where to branch to and what to do before an interrupt occurs, or else it will be forced to ignore it. In the following example, separate service routines are used for timeouts at device selectors 7 and 8:

```
100 ASSIGN 7 TO "hpib" ! "Raw" HP-IB device.
110 ASSIGN 8 TO "gpio" ! GPIO device.
120 ! Device selector : 7 = HP-IB; 8 = GPIO interface
130 !
140 SET TIMEOUT 7;2000 ! 2 second HP-IB timeout.
150 SET TIMEOUT 8;4000 ! 4 second GPIO timeout.
160 !
170 ON TIMEOUT 7 GOSUB HpibBranch ! Branch on HP-IB timeout.
180 ON TIMEOUT 8 GOSUB GpioBranch ! Branch on GPIO timeout.
190 !
200 !
210 Loop1: ENTER 724 ; Voltage ! Enter data from device 724.
220 OUTPUT 8 ; Voltage ! Output data to GPIO device.
230 GOTO Loop1
240 !
250 !
260 HpibBranch: ! Timeout service for device selector 7.
270 DISP "HP-IB Timeout"
280 RESET 7 ! Attempt to recover.
290 RETURN
300 GpioBranch: ! Timeout service for device selector 8.
310 disp "GPIO Timeout"
320 RESET 8
330 RETURN
340 END
```

NOTE

When you enable timeout end-of-line branches for more than one select code, the order of precedence of the branches is important. Refer to “How Interrupts Interact” at the end of this chapter.

Obviously, in a dedicated system more significant action could be taken than just 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 could be checked before attempting an I/O operation on that device selector. This flag might indicate a printer out of paper, for example. The following example shows how such a flag might be used to determine whether an operation (data entry from device selector 7) has been successfully completed.

```
100 ASSIGN 7 TO "hpib" ! "Raw" device.
110 SET TIMEOUT 7;2000 ! Timeout = 2 seconds.
120 ON TIMEOUT 7 GOSUB HpibTimeout ! Sets up branch.
130 !
140 Flag1=0 ! Clears timeout flag.
150 DIM Voltage(50) ! Dimensions variable V.
160 !
170 FOR I=1 TO 10
180 IF Flag1<>0 THEN GOTO Mesg ! Test timeout flag.
190 ENTER 724 ; Voltage(I) ! Enter the value of # I.
195 OUTPUT 1 ; Voltage(I) ! Output the value of # I.
200 NEXT I
210 Mesg: DISP I;" Values entered" @ STOP
220 !
230 !
240 HpibTimeout:
250 DISP "Timeout : end of data"
260 Flag1=1
270 RETURN
280 END
```

When a timeout occurs, the flag `Flag1` is set equal to 1, and the `IF` statement of line 180 causes the `ENTER` statement of line 190 to be skipped. The loop is exited and the program continues. (In this example, only a `PRINT` and `STOP` are shown for simplicity.) In this case, the cause of the timeout is not a device failure, but rather the end of data to be received.

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. For more information regarding the syntax of these statements, refer to the *HP-UX Technical BASIC Reference Manual.*

Errors

The ON ERROR statement allows you to set up an end-of-line branch to an error service routine. ON ERROR works much like ON TIMEOUT. In this case the event that causes the branch is an error. However, HP-UX Technical BASIC consists of five modules, each containing BASIC statements, and each capable of generating error conditions. Thus, in ON ERROR servicing it is necessary to determine which module is the source of the error. Error numbers up to 100 pertain to the standard BASIC statements and functions. The I/O, Advanced Programming, Matrix, and Graphics modules of BASIC share the error numbers above 100. If an error number above 100 occurs, you can identify the module that produced the error with the ERROM function. If the error in question involves an I/O module statement, ERROM will return a value of 192. Refer to the *HP-UX Technical BASIC Reference Manual* for the syntax of the ERROM function.

Once you have identified an error as an I/O module error, you should check to see if it is an interface error. Error numbers 101 and 112 will not occur during a running program. All other I/O module errors below 123 are interface-dependent errors. Therefore, a simple test for ERRN<123 will determine whether the error is an interface error. If an interface error has occurred, you can determine the select code of the interface that caused it by executing the ERRSC function.

In trapping errors with service routines it is important to run the tests in the proper order. The following simple program segment shows the recommended order of function checks 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 application.

```
10 ON ERROR GOSUB 100
20 ! Body of program goes here.
      .
      .
      .
100 ! Test for non-I/O errors.
110 IF (ERRN<101) OR (ERROM<>192) THEN GOTO NonIOerror
120 !
130 !
140 ! Test for interface errors.
150 IF ERRN<123 THEN GOTO 260
160 !
170 !
180 ! Process I/O errors. (error #'s 123 through 131)
190 DISP "I/O ERROR";ERRN;"at line";ERRL
200 !
210 ! Recovery routine goes here.
220 !
230 RETURN
240 !
250 !
260 ! Process interface errors. (error #'s 101 through 122)
270 DISP "I/O ERROR";ERRN;"at line";ERRL
280 DISP "Problem on select code";ERRSC
290 !
300 ! Recovery routine goes here.
310 !
320 RETURN
330 !
340 !
350 NonIOerror: ! Process non-I/O error here (error #'s 1 through 100)
360 RETURN
```

Refer to the *HP-UX Technical BASIC Reference Manual* for a complete list of errors and their meanings.

ON KEY Interrupts

The ON KEY statement enables a program to execute an end-of-line branch when you press a function key. For example, if a program contains the statement:

```
ON KEY# 1 GOTO 100
```

it will branch to line number 100 when you press function key number 1 (f1).

The ON KEY statement is often used in conjunction with the ENABLE KBD statement for *key mask programming*. In key mask programming, all keys are disabled except for those needed for operator input. Thus, for example, your program can prevent an operator from accidentally aborting an I/O operation in progress.

You can set up the desired key mask by specifying a key mask parameter in the ENABLE KBD statement. The key mask parameter is a decimal number from 0 to 255 representing an eight-bit byte. The upper four bits of this byte specify the key mask during program execution. The lower four bits specify the key mask during the keyboard input mode of operation. Refer to the *HP-UX Technical BASIC Reference Manual* for a complete discussion of the ENABLE KBD statement.

The following example illustrates the use of ON KEY and ENABLE KBD. ENABLE KBD 1+32 masks out all keys except the function keys during program execution. Alphanumeric input is enabled (along with the function keys) in the keyboard input mode.

```
10 ENABLE KBD 1+32 ! Select allowed keys.
20 Start: ON KEY# 1 GOTO InputData
30 ON KEY# 2 GOTO SelectOpt
40 ON KEY# 3 GOTO EndProg
50 !
60 GOTO Start
70 !
90 !
100 InputData:
110 DISP "Enter date (Month,Day,Year)" @ INPUT M,D,Y
120 DISP "Month ";M;"Day ";D;"Year ";Y
130 GOTO Start
140 !
150 SelectOpt:
160 DISP "Select operating mode." @ GOTO Start
170 !
180 EndProg:
190 DISP "End of program."
200 END
```

You can select the desired program section by pressing the appropriate function key. When you press function key number 1 (f1), the program branches to the subroutine `InputData`. When the `INPUT` statement of line `120` is executed, you can enter the date as requested.

How Interrupts Interact

Some comments are in order to help you predict the operation of programs that use 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 order of precedence for combinations of branch type and select code.

Table 5-1. Branch Precedence

	Device Selector							
Branch Type	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
`ON ERROR`	←——— 1 ———→							
`ON TIMEOUT`	2	3	4	5	6	7	8	9
`ON TIMER`	←——— 10 ———→							
`ON KEY`	←——— 11 ———→							
`ON KYBD`	←——— 12 ———→							

For example, a pending `ON TIMEOUT` branch for select code 5 would be taken before a pending `ON TIMEOUT` branch for device selector 9. Any pending `ON ERROR` branch would be taken after any pending `ON TIMEOUT` branch (regardless of the select codes).

You should 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 is completed.) If two end-of-line branches are pending, 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!

You can guarantee uninterrupted execution of the first service routine by using its first line to disable the other routine's end-of-line branch. An example will illustrate this procedure:

```
10 ASSIGN 7 TO "hpib"
20 ASSIGN 8 to "gpio"
30 SET TIMEOUT 7;4000
40 SET TIMEOUT 8;4000
50 ON TIMEOUT 7 GOSUB 120
60 ON TIMEOUT 8 GOSUB 220
70 !
80 ! Program body goes here.
90 X=X+1 @ DISP X ! Dummy loop.
100 GOTO 90
110 !
120 OFF TIMEOUT 8 ! Now device selector 8 cannot get TIMEOUT service.
130 DISP "The HP_IB interface has timed out."
140 ! This routine would do necessary service functions
150 ! for device selector 7.
160 !
170 !
180 !
190 ! Now reenable TIMEOUT service for device selector 8.
200 ON TIMEOUT 8 GOSUB 220 @ RETURN
210 !
220 OFF TIMEOUT 7 ! Now device selector 7 cannot get TIMEOUT service.
230 DISP "The GPIO interface has timed out."
240 DISP
250 ! This routine would do necessary service functions
260 ! for device selector 8.
270 !
280 !
290 ! Now reenable TIMEOUT service for device selector 7.
300 ON TIMEOUT 7 GOSUB 120 @ RETURN
310 !
320 END
```

Assume that both interrupts occur while line 90 is executing. Both end-of-line branches are pending when line 90 completes. The device selector 7 service routine has precedence, so line 120 is executed. Line 120 disables the device selector 8 service routine, so the end-of-line branch that would have occurred is not taken. Line 200, the last line of the device selector 7 service routine, reenables the device selector 8 service routine. When the RETURN is executed, the branch to line 220 occurs. Note that the OFF TIMEOUT does not *eliminate* the pending end-of-line branch, it just *defers* the branch until an ON TIMEOUT is executed later.

Events such as timeouts and errors are "remembered" indefinitely, until a RESET or STOP occurs. Therefore, once an event has occurred and remains unserviced, execution of an ON TIMEOUT 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. You should use GOTO only when the program is simple and only one or two end-of- line branches are expected. One very simple example of this is:

```
ON TIMEOUT 7 GOTO 10000
ON TIMEOUT 8 GOTO 10000
```

These two statements direct program execution to line 10000 (presumably an END statement). This kind of "bail-out" provision can be useful when developing certain types of programs that could tie up the HP-IB or GPIO bus indefinitely.

Direct Interface Communication 6

HP-UX Technical BASIC provides statements to access the control and status registers of your interface, and thus to tailor the operation of the interface to the specific requirements of your system.

You can monitor the status of interface operations with the **STATUS** statement. This status may reflect the actual high/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. You can also use **STATUS** to check the status of an I/O buffer (refer to the chapter "Specialized Data Transfers").

You can use the **CONTROL** statement to direct the mode of operation of an interface. The interface generally provides automatic control of external I/O lines according to the mode selected, and also may provide for manual override of certain of the I/O lines for custom sequences.

Each interface has its own status and control registers. The functions of these registers are interface dependent, and are covered in the appendix.

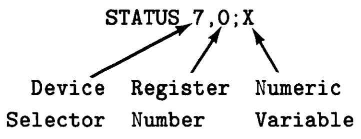
Chapter Contents

This chapter discusses the accessing of control and status registers for tailoring the operation of your particular interface to the system needs. Tasks and topics which are covered in this chapter are as follows:

Tasks/Topics	Page
Checking Interface Status	6-2
Interface Control	6-4

Checking Interface Status

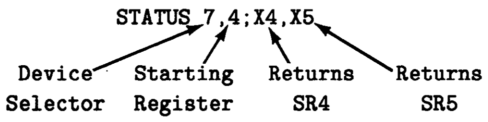
You can obtain the status of an interface register by specifying the select code of the interface, the number of the desired register, and the name of a numeric variable in a **STATUS** statement. The **STATUS** statement returns the value of the specified register. The following statement obtains the value of the identification register (Status Register 0) for the HP-IB interface at device selector 7:



```
STATUS 7,0;X
```

The value of Status Register 0 is always the interface's identification code (1 for the HP-IB). For the other status registers, **STATUS** returns a value that depends on the status of the interface (refer to the appendix).

You can use one **STATUS** statement to obtain the values of several *consecutive* status registers. Just specify the register number of the first status register in the series and list one numeric variable for each status register to be read. For example, the following statement reads Status Registers 4 and 5 (SR4 and SR5) of the HP-IB interface at device selector 7:



```
STATUS 7,4;X4,X5
```

What you do with the contents of these registers depends upon your application, needs, and type of interface. Status Registers 1 through 6 for the HP-IB interface indicate the following:

Status Register 4	HP-IB address.
Status Register 5	HP-IB state.

The GPIO interface returns different information for these same registers.

Interface Control

The `CONTROL` statement allows you to program the operation of the interface by writing to its control registers. Two items of general interest are:

- setting and clearing interface control lines,
- selecting an end-of-line (`EOL`) sequence.

Control lines are interface dependent. In general, you can set or clear interface-specific control lines by writing to Control Register 2 of the desired interface. Refer to the appendix for information on Control Register 2 for each interface.

CAUTION

DO NOT WRITE TO THE INTERFACE CONTROL LINE REGISTER (CONTROL REGISTER 2) UNLESS YOU ARE COMPLETELY FAMILIAR WITH THE PROTOCAL FOR YOUR INTERFACE. AN IMPROPER CONTROL LINE OPERATION MAY CAUSE A LOSS OF DATA, A DEVICE MALFUNCTION, OR DAMAGE TO YOUR INTERFACE.

Suppose that you want to set the `REN` (Remote Enable) control line on the HP-IB. You can do this by writing the value 64 to Control Register 2 for the HP-IB. You can use either of the following statements (assuming select code 7):

```
CONTROL 7,2;64
ASSERT 7;64
```

Both statements set the `REN` line true and *immediately* set the control line (REN in this case) regardless of any I/O operation in progress.

A second item of general interest in control register programming is the end-of-line (`EOL`) character sequence sent by the interface. This `EOL` sequence defaults to carriage return/line feed for all interfaces, and is sent after every `PRINT` or `OUTPUT` operation (unless you suppress it). The `EOL` sequence is also sent if you include the / (slash) image specifier in a `PRINT` or `OUTPUT` statement, or if you specify `EOL` in a `SEND` statement.

You may occasionally want to change the `EOL` sequence to suit the needs of a particular device. For example, you could use "carriage return/line feed/line feed" to obtain double spacing on a printer. For a CRT terminal that performs an automatic line-feed when a carriage return is received, you could use just a carriage return.

The HP-IB interface provides **EOL** sequence programmability with Control Registers 16 through 23. Up to seven **EOL** characters can be sent. You specify the number of characters by writing to Control Register 16. You then specify the consecutive characters by writing to CR17 through CR23. For example, you could use the following **CONTROL** statement to program a double line-feed **EOL** sequence with EOI (End Or Identify) set on the last line feed.

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

      Enable     EOL    Carriage Line-Feeds
        EOI     Count    Return
                 of 3
```

NOTE

Control Registers 16 through 23 are used in exactly the same way to select an **EOL** sequence for the GPIO interface. The method is similar for the GPIO interface, but the function of Control Register 16 is slightly different (refer to the appendix).

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 register descriptions in the appendix to determine which capabilities you need and how to implement them. You may need to experiment a bit to obtain the result you desire.

Notes

Interface Dependent Statements 7

The BASIC I/O statements that we have covered so far have not been interface dependent. That is, the operation of these statements is the same regardless of the interface that you are using.[1] The remaining BASIC I/O statements have *interface dependent* operation. Many of these statements are not applicable to all of the interfaces, and they may operate differently with each interface.

The following table lists the interface-dependent BASIC I/O statements and indicates which interfaces implement each statement. An asterisk (*) in an interface column indicates that the statement is implemented by that interface. If the column contains "NA", the statement is not applicable to that interface.

Table 7-1. Interface Dependent I/O Statements

	Interface	
BASIC Statement	**HP-IB**	**GPIO**
`ASSERT`	*	*
`CLEAR`	*	NA
`LOCAL`	*	NA
`LOCAL LOCKOUT`	*	NA
`PASS CONTROL`	*	NA
`PPOLL`	*	NA
`REMOTE`	*	NA
`REQUEST`	*	NA
`RESET`	*	*
`SEND`	*	*
`SPOLL`	*	NA
`TRIGGER`	*	NA

The *HP-UX Technical BASIC Reference Manual* defines the syntax of all BASIC statements, and defines the operation of the above statements for each interface.

[1] The *operation* of the **STATUS** and **CONTROL** statements is not interface dependent, although the *effect* of these statements is. This is because the control and status registers are different for each interface (refer to the appendix).

Chapter Contents

This chapter covers the `SEND` and `RESET` statements. These statements apply to all of the interfaces, though their operation is interface dependent.

The tasks and topics which are covered in this chapter are as follows:

Tasks/Topics	Page
The SEND Statement	7-2
The RESET Statement	7-3

The SEND Statement

In some applications you may want to send an arbitrary data sequence from the computer to one or more peripherals. You can use the `SEND` statement to send data to any of the interfaces. The following are examples of properly formed `SEND` data statements:

```
SEND 7; DATA "Welcome to the world of I/O" EOL
SEND 8; DATA "OCT. 9", "NOV. 13"
```

An end-of-line sequence is sent by the interface if you specify `EOL`.

The commands that you may send depend on the interface that you are using. The *HP-UX Technical BASIC Reference Manual* defines the syntax of the `SEND` statement. The following are examples of correctly formed `SEND` command statements:

```
SEND 8; DATA 23
SEND 7; MTA UNL LISTEN 4,5 CMD F,H
SEND 7; CMD X$
```

Both commands and data may be included in the same `SEND` statement. For example:

```
SEND 710; DATA "ABC" CMD 23
```

The RESET Statement

`RESET` terminates the current I/O operation and returns all interfaces to their default states. This should only be necessary when first beginning a program or when recovering from an interface failure.

The remaining interface dependent statements are quite specific to the interface being used. The operation of each statement for each interface is defined in the *HP-UX Technical BASIC Reference Manual*. The following two chapters of this manual cover selected I/O programming techniques for the various interfaces (the chapter “Using the HP-IB Interface” covers the HP-IB interface and the chapter “Using the GPIO Interface” covers the GPIO interface). These chapters give several examples involving the interface dependent statements.

Notes

Using the HP-IB Interface 8

This chapter presents some *selected programming techniques* for the HP-IB interface. The chapter "The HP-IB Interface" describes the HP-IB interface itself. Refer to the appendix for a description of the HP-IB status and control registers. All of the applications presented in this chapter assume that your computer is the system controller (except the examples in "Non-Controller Operations").

NOTE

Before you can perform any I/O operations with an interface, you must assign an device selector number to it (refer to the chapter "An Introduction to I/O"). The examples in this section assume that device selector number 7 has been assigned to the HP-IB interface.

Chapter Contents

This chapter covers the following tasks and topics:

Controlling the Bus

Typically, the first operation necessary for HP-IB systems is to program all devices for *remote* operation. Most HP-IB devices are capable of manual (front panel) operation or of remote (bus controlled) operation. You can put a device into remote mode by setting the REN (Remote Enable) line true and addressing the device to listen. (HP-IB control lines are covered in the chapter "The HP-IB Interface.") The computer sets the REN line when you turn on the power, execute a hardware reset, or execute the `REMOTE` statement. You can address a device to listen by executing any statement that includes the device's listen address. The `REMOTE` statement is usually the most convenient method of setting devices to the remote mode. For example, to place HP-IB devices 713 and 722 under remote control, execute the following statement:

```
10 REMOTE 713,722
```

The `LOCAL LOCKOUT` statement sends the `LLO` (Local Lockout) message. This prevents devices from being returned to local control from the front panel (by the Return-to-Local switch). The `LOCAL LOCKOUT` statement affects all devices connected to the interface at the specified device selector number. The above example now looks like this:

```
10 REMOTE 713,722
20 LOCAL LOCKOUT 7
```

Now the system is set up for remote control with the front panel controls disabled. The next step is to program each device for the desired mode of operation (for example, the appropriate range setting of a voltmeter). In remote mode HP-IB devices interpret incoming bytes as command sequences. Thus, you can program an HP-IB device with a simple `OUTPUT` statement directed to its device selector. For example, suppose an HP 3437A Digital Voltmeter is to be set to the 0.1 dc voltage range in the continuous sample mode. The ASCII character sequence `F1R1T1` will set the instrument to the desired range and mode. If the voltmeter has the device selector 722, the following statement will accomplish this:

```
   50 OUTPUT 722 ; "F1R1T1"
```

The owner's manual for each HP-IB device gives the ASCII character sequences needed to select different ranges and modes. The meaning of such a character sequence is device dependent.

Now let's look at a specific example. The figure below shows a monitoring configuration in which an HP 3437A Digital Voltmeter and an HP 5328A Frequency Counter are used to record the input voltage and the output frequency of a voltage to frequency converter.

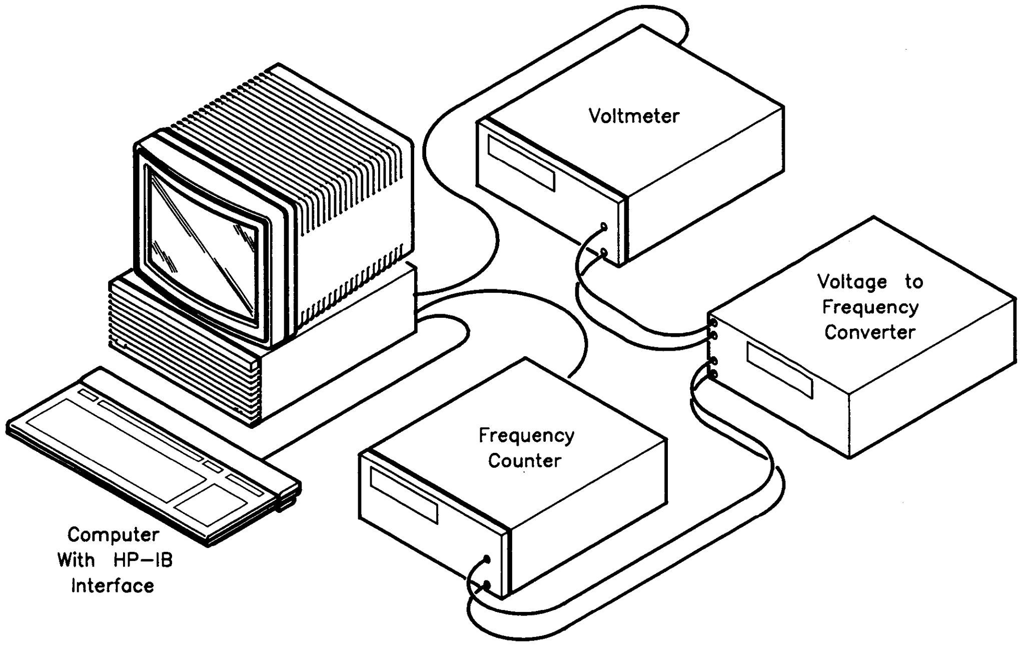


Figure 8-1. Monitoring Voltage and Frequency

The following program could be used to enter voltage and frequency readings into the computer:

```
10 ASSIGN 7 TO "hpib" ! "Raw" mode.
20 !
30 REMOTE 713,724
40 LOCAL LOCKOUT 7
50 ! Select .1vdc range on voltmeter.
60 OUTPUT 724;"F1R1T1"
70 ! Select 1K hz resolution on frequency counter.
80 OUTPUT 713;"PF4G3S1S3S5T"
90 ! Take reading from voltmeter and place into V.
100 ENTER 724;V
110 ! Take reading from counter and place into F.
120 ENTER 713;F
130 PRINT "Voltage =";V
140 PRINT "Frequency =";F
150 LOCAL 713,724 ! Return devices to local control.
160 END
```

The above program just enters the most recent readings taken by the voltmeter and counter. The following program obtains several voltage/frequency pairs, reading voltage and frequency simultaneously, by triggering the instruments. The `OUTPUT` statements are changed to put the instruments in the "triggered sample" operating mode, and a `TRIGGER` statement is inserted just before the `ENTER` statements. The `TRIGGER` and `ENTER` statements are put into a loop to obtain 100 pairs of readings:

```
10 ASSIGN 7 TO "hpib"
20 !
20 REMOTE 713,724
30 LOCAL LOCKOUT 7
40 ! Select .1vdc range, "single trigger" mode on device 724 (voltmeter).
50 OUTPUT 724;"F2R2T2T3"
60 ! Select 1K hz resolution, "single trigger"
70 ! mode on device 713 (freq. counter).
80 OUTPUT 713;"PF4G3R"
90 !
100 FOR I=1 TO 100
110 TRIGGER 713,724
120 ! Take reading from voltmeter and place into V.
130 ENTER 724;V
140 ! Take reading from counter and place into F.
150 ENTER 713;F
160 PRINT "Voltage =";V
170 PRINT "Frequency =";F
180 NEXT I
190 !
200 LOCAL 713,724  ! Return devices to local control.
210 END
```

Advanced I/O Operations

You can use the OUTPUT and ENTER statements (found in the chapters “Simple I/O Operations” and “Formatted I/O Operations”) in most I/O applications. These statements normally operate without any dependence on the interface being used. This section covers a few I/O programming situations involving OUTPUT, ENTER, and SEND that *are* specific to the HP-IB interface.

Burst Mode I/O (only for Series 200/300)

Burst mode provides low-overhead I/O on some HP-IB interfaces. This can result in much faster data transfer.

NOTE

Burst mode **only** works for the HP 98624 HP-IB interface and the internal HP-IB.

In *burst mode*, memory-mapped I/O address space assigned to the interface card select code is mapped directly into user address space such that the user can transfer data directly to and from the interface card. This eliminates the need for system calls and their associated overhead. An example which uses the *burst mode* to input data from a fast voltmeter located at primary address 24 is given below.

```
100 ASSIGN 7 TO "hpib"
110 CONTROL 7,9 ; 1 ! Enable "burst mode".
120 !
130 !
140 REMOTE 724
150 LOCAL LOCKOUT 7
160 ! Select .1vdc range on voltmeter.
170 OUTPUT 724 ; "F1R1T1"
180 ! Take reading from the voltmeter and place into V.
190 ENTER 724 ; V
200 DISP "Voltage = ";V
210 LOCAL 724 ! Return devices to local control.
220 END
```

When *burst mode* is enabled, the interface is locked and no other processes are allowed to use the interface until *burst mode* is disabled.

TIMEOUTs which may have been set up are disabled when *burst mode* is enabled. The only way to interrupt a burst OUTPUT or ENTER is by pressing CTRL - \ which resets the program executing these commands by returning it back to its non-executing state.

WARNING

ENABLING *BURST MODE* LOCKS THE HP-IB INTERFACE AND **SHOULD NOT** BE USED WITH ANY INTERFACE SUPPORTING A SYSTEM DISC OR SWAP DEVICE.

Non-controller Addressing

Non-controller addressing is used when the computer has passed control to another device and is no longer the active controller, or when the HP-IB interface has been set to the non-system controller mode (and has not received active control from the system controller).

When the computer (or any device, for that matter) is not the active controller, it must wait to be addressed to talk or listen before it may output or enter data on the bus. There are two ways the computer can wait to be addressed. The first method is automatic and would be the method of choice for a simple system. If you specify the HP-IB select code (with no primary address) in an `ENTER` or `OUTPUT` statement, the computer will wait to be addressed before transferring the data. The following statements show the conditions necessary to begin a data transfer:

`ENTER 7;A$`	The non-controller computer waits to be addressed by the controller to listen, then reads the data into A$.
`OUTPUT 7;A$`	The non-controller computer waits to be addressed by the controller to talk, then sends A$.

The problem with this first method is that, although the computer does wait to be addressed before starting the transfer, the program is held up at the `ENTER` or `OUTPUT` statement while it is waiting.

In the second method the computer waits to be addressed by performing a status check for the proper condition (addressed to talk or addressed to listen). This allows the computer to accomplish some useful task while waiting to be addressed. (Refer to the appendix for a complete discussion of HP-IB status registers.) The program proceeds with its normal task, then performs the data transfer when the status check indicates that the computer has been addressed. For example, the following program increments and displays a counter, then checks to see if the computer has been talk addressed. If not, the program loops back to the increment and display statements. Otherwise, the value of the counter is transferred to the HP-IB:

```
10 ASSIGN 7 TO "hpib,/dev/lp"
20 ! Lines 60, 70 check for talk-address state.
30 Start1: I=0
40 Start2: I=I+1
50 DISP I
60 STATUS 7,5 ; S
70 IF BIT(S,4)=0 THEN GOTO Start2 ! Is "TA" (Talker Active) bit set?
80 OUTPUT 7 ; I
90 GOTO Start1
100 END
```

You can apply the same procedure to an ENTER operation. In this case the status check indicates whether the computer has been addressed to listen. The difference is in the bit to be tested, and is shown in the following statements:

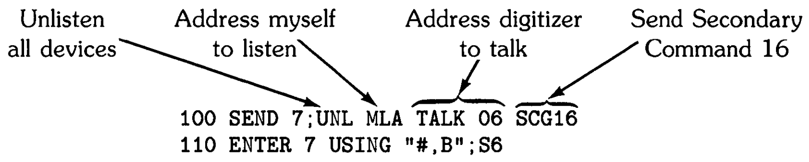
```
10 ASSIGN 7 TO "hpib"
20 STATUS 7,5 ; S
30 IF BIT(S,6)=0 THEN DISP "Not addressed to listen."
40 END
```

Bit 6 of Status Register 5 indicates the listen-addressed (LA) state when it is set to a 1. Bit 4 of the same register indicates the talk-addressed (TA) state when it is set to a 1.

Custom Bus Sequences

You may occasionally need to send a custom bus sequence to an HP-IB device under development or to one that requires a sequence different from those normally sent by the computer. The SEND statement makes such custom operations possible. With SEND you must specify every character of the sequence yourself (other statements generate them automatically).

As one example of a custom bus sequence, the following statements send a Secondary Command to an HP 9111 Graphics Tablet (device 06) on device selector 7. The Secondary Command "16" causes the digitizer to output its status byte. This byte is read by the following ENTER statement (notice that ENTER does not specify addressing).



```
100 SEND 7;UNL MLA TALK 06 SCG16
110 ENTER 7 USING "#,B";S6
```

Other operations that could be performed include Parallel Poll Configure and Parallel Poll Unconfigure. The chapter entitled, "The HP-IB Interface," discusses HP-IB control lines, messages, and bus sequences.

Multiple Listener Transfers

If you want more than one device to receive the data being transferred by an OUTPUT or ENTER statement, you may include these devices in a listen address group. You can use the SEND statement to set up such a group. The procedure differs for input and output transfers, as described in the following paragraphs.

In an OUTPUT operation the controller computer is automatically addressed to talk. If you want more than one device to listen, you need only address the desired devices to listen in a SEND statement. You may then execute the OUTPUT statement, specifying just the device selector with no primary address. Note that all listeners and the talker must be on the same select code. For example, assume that a string of characters (B$) is to be sent to a printer (device 04) and a non-controller computer (device 20). The devices are connected to the HP-IB interface (device selector 7) of the controller computer. You could use the following sequence of program statements:

```
50 SEND 7; UNL LISTEN 4,20
60 OUTPUT 7;B$
```

If you are executing an ENTER operation, you may specify a talker plus multiple listeners with SEND. Once you have configured the bus with SEND, you may execute the ENTER statement, specifying only the device selector with no primary address. For example, suppose device 06 (a voltmeter) is to be the talker and device 13 (a printer), device 04 (a disc drive), and your computer (the controller) are to be listeners. The following sequence unlistens all previous listeners, sends the new talk and listen addresses, then enters the data. Note that the computer is addressed with MLA (My Listen Address).

```
50 SEND 7; UNL TALK 6 LISTEN 13,4 MLA
60 ENTER 7;V
```

NOTE

If you have configured the bus for a Listen Address Group with a SEND statement, specify only the device selector with no primary address in the OUTPUT or ENTER statement that follows. If you specify a primary address, the bus will be reconfigured.

In some cases you may want to transfer data from a talker to one or more listeners on the bus without the computer being involved. Although it is possible to initiate such a transfer, the computer will not be able to tell when it is complete.

Handling Service Requests

This section deals with asynchronous requests for service. The cause of a service request (SRQ) is device dependent (different devices have different reasons for requesting service). For example, a printer may request service because it has just run out of paper, or a digitizer may request service because an error has occurred. In any event, if a service request is received, it is necessary to determine what device requested service and what the problem with that device is.

Sensing Service Requests

Your program can detect service requests by periodically checking the status of an HP-IB status register (refer to the chapter "The HP-IB Interface"). Bit 5 of Status Register 2, when equal to "1", indicates that a service request (SRQ) has been received. You can use the following program lines to determine this:

```
10 STATUS 7,2;S
20 IF BIT(S,5)=1 THEN DISP "SRQ received"
```

Determining the Problem

The purpose of the HP-IB serial poll is to provide the active controller with specific, device dependent information about the device being polled. The bits of the device's serial poll response byte can have any meaning assigned to them and are generally used to indicate some problem or special condition within the device. Bit 6, however, is reserved to indicate that the device is currently requesting service.

To keep the discussion simple, assume that only one device is capable of requesting service, device selector number 7, a printer. Assume that the bits of the status byte for this printer have the following meanings:

Bit 0	Out of paper
Bit 1	Cover off
Bit 2	Parameter out of range
Bit 3	Improper escape sequence
Bit 4	Always 0
Bit 5	Always 0
Bit 6	SRQ Active
Bit 7	Always 0

The following is a possible service routine for the printer:

```
100 ASSIGN 7 TO "hpib" ! "Raw" device.
110 !
120 ! Service for device 701
130 SpollResponse=SPOLL(701)
140 IF BIT(SpollResponse,6)=1 THEN GOSUB SrqActive
150 IF BIT(SpollResponse,0)=1 THEN GOSUB OutPaper
150 IF BIT(SpollResponse,1)=1 THEN GOSUB CoverOff
170 ! And so forth
       .
       .
       .
1240 SrqActive:
1250 DISP "The HP-IB printer is requesting service."
1260 RETURN
1270 !
1280 OutPaper:
1290 DISP "HP-IB printer is out of paper."
1300 DISP "Replace paper and type CONT <carriage return>."
1310 PAUSE
1320 RETURN
1330 !
1340 CoverOff:
1350 DISP "The printer's cover is off."
1360 DISP "Put the cover back on and type CONT <carriage return>."
1370 PAUSE
1380 RETURN
1390 END
```

In the more usual system, with more than one device, the program would just perform sequential polls to determine which device requested service, and to determine the current status of that device. The following routine shows how this is done:

```
100 S=SPOLL(715)
110 IF NOT BIT(S,6) THEN GOTO 200
120 ! Lines 120 to 190 service device 715
    .
    .
    .
200 S=SPOLL(723)
210 IF NOT BIT(S,6) THEN GOTO 300
220 ! Lines 220 to 290 service device 723
    .
    .
    .
300 ! And so forth
    .
    .
    .
```

Non-controller Operations

When the computer is not the active controller, you must observe certain rules in your programs to avoid violating bus protocols. That is, certain operations can only be performed by the system controller, others only by an active controller. A non-controller is only allowed to talk, listen, and request service.

Passing Control

The computer, as active controller, can pass HP-IB controller responsibilities to another device by executing the PASS CONTROL statement. This allows the computer to direct its attention to activities other than bus control. The following statement passes control to device 20 on the bus, another computer (currently a non-controller):

```
70 PASS CONTROL 720
```

The new controller may pass control back to the first computer. Thus, you must make a provision for the first computer to determine that control has been passed back. This is described under "Receiving Control." Even the system controller can become a non-controller. On a system reset, the system controller becomes the active controller again. Only one system controller is allowed.

Receiving Control

You can determine that the computer has received active control by checking interface status — the "controller active" (CA) bit. The following program passes control to the peripheral device located at device selector 724. The program continues to loop until the device receives control.

```
100 ! Checks to see whether or not
110 ! an HP-IB interface is currently in the
120 ! 'Controller Active' (CA) state.
130 !
140 ASSIGN 7 TO "hpib" ! Must be a 'raw' device.
150 !
160 PASS CONTROL 724 ! Now pass control to someone else.
170 !
180 DISP "Waiting for control."
190 Loop: STATUS 7,5 ; HpibState ! Keep checking for CA state.
200 IF BIT(HpibState,5)=0 THEN NotCA=1
210 IF NotCA THEN GOTO Loop
220 DISP "Received control."
230 ! ...
```

Non-controller Responses

The example program in this section monitors non-controller status. While the controller active (CA) bit of status register 5 remains 0, the program continues to loop and display the following messages:

```
Not in 'Listener Active' state.
Not in 'Talker Active' state.
```

As soon as a peripheral device receives control, the program exits the loop and displays the message:

```
Received control.
```

The program is as follows:

```
100 ASSIGN 7 TO "hpib" ! Must be in 'raw' mode.
110 !
120 DISP "Monitoring non-controller status."
130 !
140 Loop: STATUS 7,5 ; HpibState ! Keep checking HP-IB state.
150 IF BIT(HpibState,6)=0 THEN NotLA=1
160 IF BIT(HpibState,5)=0 THEN NotCA=1
170 IF BIT(HpibState,4)=0 THEN NotTA=1
180 IF NotLA THEN DISP "Not in 'Listener Active' state."
190 IF NotTA THEN DISP "Not in 'Talker Active' state."
200 IF NotCA THEN GOTO Loop
210 !
220 DISP "Received control."
230 ! ...
```

Sending Service Requests

Some condition in a non-controller computer may require the attention of the active controller, so the computer needs to be able to send a service request to the controller. The REQUEST statement allows the computer to assert SRQ and to send a serial poll response byte when the controller polls it. Bit 6 of this byte should be set to a 1 to indicate that the computer indeed requested service, but the other bits may indicate anything that you (as the system designer) deem to be important.

As an example, assume that a non-controller computer has passed control to device 724 and needs to indicate that it is ready with data. Assume that bit 0 of the serial poll response byte has been chosen to indicate the ready with data condition. The following statement sets SRQ true for device selector 7 and sets bits 0 and 6 for the serial poll response byte:

```
REQUEST 7;1+64
```

Handling Interface Problems

This section describes some of the techniques you can use to avoid and deal with problems that may arise when using the HP-IB interface. You should not *expect* problems, but good programming practice anticipates problems and deals with them in advance.

Avoiding Bus Hang-ups

Generally, when an HP-IB device develops a problem, it holds up the current data transfer, sends an SRQ (service request) to the controller, or it does both. End-of-line branches normally can handle these conditions, but suppose the device stops handshaking in the middle of a data transfer and at the same time sends an SRQ. The computer cannot branch to service the SRQ since the end-of-line branch does not occur until the current program line is finished. The computer is "hung" on the `ENTER` or `OUTPUT` statement that was being executed when the loss of handshake occurred. The computer can recover from the loss of handshake, however, by using an `ON TIMEOUT` branch.

The following example shows a sequence of operations that enables a program to recover from a bus hang-up:

```
10 ASSIGN 7 TO "hpib" ! "Raw" mode.
20 ! First set up the timeout branch.
30 ON TIMEOUT 7 GOSUB TimeOutBranch
40 ! Then set a handshake time limit (2.0 sec).
50 SET TIMEOUT 7;2000
60 ! Now program the HP 3437 for triggered readings.
70 OUTPUT 724 ; "F1R1T2T3"
80 ! Now trigger and read the DVM.
90 Loop1: TRIGGER 724
100 ENTER 724 ; X
110 DISP X;"volts"
120 GOTO Loop1
130 !
135 !
140 TimeOutBranch:
141 ! This is the timeout service routine.
142 ! Its purpose is to allow you to correct
143 ! the problem which caused the timeout.
144 !
145 !
150 RETURN
155 !
160 END
```

It is up to you, as the programmer, to determine what actions the program takes in response to the status check. In most cases it is necessary to `RESET` the interface, avoid transfers with the device causing the hang-up, and signal to the system operator that a malfunction has occurred in that device.

Dealing With Problems

While it is best to anticipate problems and make provisions for them in your program, some condition may cause your system to fail. If the HP-IB system appears to be locked up (everything is running, but nothing is happening) you can perform a hardware reset to regain control of the computer.

You may now wish to perform a serial poll of the device in question to determine its condition. This, however, requires that you know how to obtain the correct information for that particular device. It may be sufficient to turn the device off, then back on to return it to normal operation.

The following table shows some techniques (and the reasons for each) that you may use when writing a routine to handle HP-IB error conditions.

Table 8-1. Error Recovery Techniques

Action	Reason/Result

`FOR I=1 TO 5`
`STATUS 7,I;S(I)`
`NEXT I`
	Obtain the current state of the interface to determine the conditions and possible causes.

`S=SPOLL` (*each device*)	Obtain current status of devices in the system. (A printer may be out of paper, for example.)
`CLEAR` *selected device*	Return the desired device to its particular device-dependent "clear" state. (Like a reset.)
`CLEAR 7`	Return all devices to their device-dependent states.

It is a good practice to preserve a hard copy record of the interface and device status information as it is obtained. You should also record the actions taken to correct the situation. This information is vital for analyzing the cause of a system failure, and can be used in making the necessary adjustments or corrections.

Notes

The HP-IB Interface 9

An HP-IB interface is built into many Hewlett-Packard computers. The HP-IB (Hewlett-Packard Interface Bus) interface system is Hewlett-Packard's implementation of the IEEE-488-1978 interface standard. HP-IB provides mechanical, electrical, timing, and data compatibility between all devices adhering to the standard. A group of HP-IB devices can be connected together to form a *system*, capable of maintaining orderly communication between the various devices. Since HP-IB is a parallel bus system, it is commonly called *the bus*.

You may want to refer to one of the following sources for additional information on the HP-IB interface:

- *IEEE Standard Digital Interface for Programmable Instrumentation*, IEEE Std. 488-1978, Copyright 1978, Institute of Electrical and Electronics Engineers, Inc.
- *Tutorial Description of the Hewlett-Packard Interface Bus*, HP part number 59300-90007, Copyright 1980, Hewlett-Packard Company.

Chapter Contents

This chapter covers the following tasks and topics:

General Structure of the HP-IB

As you would expect of any system, HP-IB has a protocol a precise set of rules that is followed as the system conducts its business. This protocol is defined by the IEEE-488-1978 standard. A complete discussion of HP-IB protocol is beyond the scope of this manual (refer to the IEEE Std. 488-1978 document). However, the general concepts of the HP-IB are covered in this chapter.

Although up to 15 devices can be connected to form an HP-IB system, one device — the *system controller* — has primary control of the bus. Your computer is usually the system controller. The system controller becomes the *active controller* when it is turned on, and assumes the responsibility to maintain order on the bus. It determines which devices may "talk," which devices may "listen," and so forth. However, the system controller can pass its controller duties to another device on the bus (for example, another computer), which then becomes the active controller. There can be only one system controller on the bus, and only one device may be the active controller at any time. (More than one device may be capable of receiving control, however).

The active controller (which may also be the system controller) enables an HP-IB device to transmit messages by a process known as *addressing to talk*. The device that has been so addressed is called the *active talker*. There can be only one active talker on the bus at any time. The active controller can *unaddress* the active talker by sending an *untalk* command.

The active controller can enable one or more devices to "listen" to HP-IB messages by a process known as *addressing to listen*. The devices that have been so addressed are called *active listeners*. The active controller can unaddress the listeners by sending the *unlisten* command.

For many I/O applications using the HP-IB interface you need not concern yourself with messages, listen addressing and so forth. `OUTPUT`, `ENTER`, and other I/O statements automatically perform the necessary functions for you. However, you may have a specialized application that requires a more detailed study of the HP-IB system. The rest of this chapter provides some specific details about HP-IB. You should also refer to the chapter "BASIC Programming With the HP-IB Interface" for some selected programming techniques using HP-IB.

HP-IB Signal Lines

The Hewlett-Packard Interface Bus (HP-IB) consists of 16 parallel *signal lines*. These signal lines are divided into three categories: the *data bus*, the *handshake lines*, and the *control lines*. The figure below shows how these signal lines are used to connect HP-IB devices together:

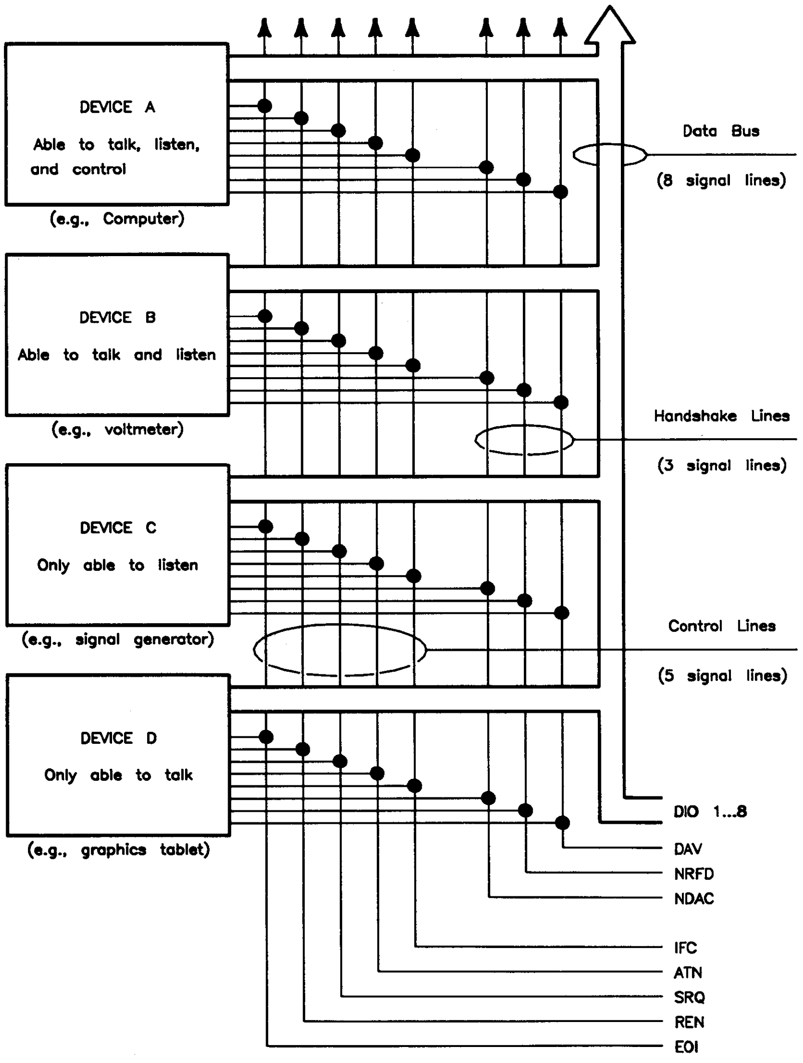


Figure 9-1. HP-IB Signal Lines

Transmitting Data Bytes

Eight of the signal lines are used as a parallel data bus. These lines (`DIO1` through `DIO8`) are used to transmit eight-bit bytes of data from the active talker to any active listeners on the bus. The transmission of each byte of data is accomplished by a handshaking process using the handshake lines. The three handshake lines are:

DAV	Data Valid
NRFD	Not Ready For Data
NDAC	Not Data Accepted

Using these lines, a typical data exchange would proceed as follows. All devices currently designated as active listeners would indicate (via the **NRFD** line) when they are ready for data. A device that is not ready would assert this line true, while a device that is ready would let the line go false. This line will stay true until all active listeners are ready for data. When the talker senses this, it places the next data byte on the data lines, then asserts **DAV** true. This tells the listeners that the information on the data lines is valid, and that they may read it. Each listener (at its own speed) then reads the data and lets the **NDAC** line go false. Again, only when all listeners have let **NDAC** go false will the talker sense that all listeners have read the data. It can then let **DAV** go false and start the entire handshake sequence over again for the next byte of data.

NOTE

All HP-IB data, control, and handshake lines use negative-true logic.

HP-IB Control Lines

In addition to the eight data lines and three handshake lines, there are five HP-IB control lines. The control lines are used by the system controller to send commands to other HP-IB devices, or by external devices to send signals to the controller.

`ATN` (Attention)

Command messages are encoded on the data lines as eight-bit characters, and are distinguished from normal data characters by the setting of the `ATN` (attention) line. That is, when the `ATN` line is false, bytes on the data lines are interpreted as simple data characters. But, when the `ATN` line is true, the data lines become the carriers of command information.

`IFC` (Interface Clear)

Only the system controller can set the `IFC` line true. When the `IFC` line is asserted, all bus activity is unconditionally terminated, the system controller regains active controller status (if control has been passed to another device), and any current talkers and listeners become unaddressed. Normally, this line is only used to abort an unwanted operation, or to allow the system controller to regain control of the bus when something has gone wrong. It overrides any other activity that is currently taking place on the bus.

`REN` (Remote Enable)

This line, when true, allows instruments on the bus to be programmed remotely by another device on the bus, usually (but not necessarily) the active controller. Only the system controller can change the state of this line. Any device that is addressed to listen while `REN` is true is placed in the `REMOTE` mode of operation.

`EOI` (End or Identify)

Normally, data messages sent over the HP-IB are sent using the standard ASCII code and are terminated by the ASCII line-feed character (LF = decimal 10). However, you may want to have a device send blocks of information as eight-bit bytes that represent binary patterns rather than ASCII characters. No specific eight-bit pattern can be designated as a terminating character since it may occur anywhere in the data stream. In this case, the `EOI` line is used to mark the end of the data sequence. When the listeners detect that the `EOI` line is true, they recognize that the byte on the data lines is the last byte of the sequence.

The `EOI` line is also used in conjunction with the `ATN` line to initiate an identify (parallel poll) sequence.

SRQ (Service Request)
The active controller is always in charge of the order of events on the HP-IB. If a device on the bus has some information that the controller should know about, it can use the **SRQ** line to ask for the controller's attention. For example, a printer might request service to inform the controller that it is out of paper. Or a digitizer might assert **SRQ** to tell the controller that its sample button was pressed by the operator and a reading is ready to be taken. This represents a request (not a demand), and it is up to the controller to determine when and how it will service that device. However, the device will continue to assert **SRQ** until it has been satisfied. Exactly what will satisfy a service request depends on the individual device (refer to the owner's manual for each device). If **SRQ** has been asserted, the controller can determine which device has requested service by conducting a serial or parallel poll (refer to the subheading "Polling" later in this chapter).

HP-IB Messages

The term *message*, applied to HP-IB, can refer to a *data byte message* or to a *command message*. A data byte message is an eight-bit byte of data transmitted over the HP-IB data lines. HP-IB command messages may be generally classified as single-line messages and multiple-line messages. A single-line message is sent by asserting just one control line or handshake line. Multiple-line messages are sent by asserting **ATN**, then sending a command byte over the data lines. The tables that follow are complete lists of the single-line and multiple-line command messages. The response to some of these messages is device dependent and/or user defined. Most messages are originated by the controller and cause some specified action to take place on another HP-IB device.

Single-Line Messages

The following single-line command messages are sent by asserting the specified control line or handshake line:

Table 9-1. Single-Line Messages

Menemonic	Message Name	Response
		Control-Line Messages
`ATN`	Attention	The Controller Active device asserts `ATN` true to source commands on the data bus or in conjunction with `EOI`, to do a parallel poll. When `ATN` is false, data may be sent over the data bus by a designated talker.
`IFC`	Interface Clear (Abort)	The system controller uses this to place talkers and listeners in an unaddressed state. If control has been passed, the system controller again becomes Controller Active when it asserts `IFC`.
`REN`	Remote Enable	Removes all devices from Local Lockout mode and causes all devices to revert to manual control. Any device that is addressed to listen while `REN` is true is placed in the REMOTE mode of operation. Only the system controller can change the state of this line.
`SRQ`	Service Request	Indicates a device's need for interaction with the controller.
`EOI`	End or Identify	Terminates a flow of data, and can be used with `ATN` to do a parallel poll.
		Handshake-Line Messages
`DAV`	Data Valid	Allows source to validate data lines.
`NRFD`	Not Ready For Data	Used to inform the source that all listener devices are ready for data.
`NDAC`	Not Data Accepted	Used by devices to inform the source that data has been accepted.

Multiple-Line Messages

Multiple-line messages involve the ATN line and the eight data lines. The ATN line is asserted, then the appropriate command byte is sent over the data lines. The following table gives the mnemonic for each message, plus the decimal value of the command byte to be sent:

Table 9-2. Multiple-Line Messages

Mnemonic	Message Name	Decimal Value	Response
GTL	Go To Local	1	Causes selected device(s) to switch to local (front panel) control.
LAG (LA0-LA30)	Listen Address Group	32-62	A group of 31 listen addresses, one of which corresponds to the listen address of the interface.
UNL	Unlisten	63	Device(s) become unaddressed to listen.
TAG (TA0-TA30)	Talk Address Group	64-94	A group of 31 talk addresses.
UNT	Untalk	95	Device becomes unaddressed to talk.
LLO	Local Lockout	17	Disables remote-mode override switch (the LOCAL button) on peripheral device(s).
DCL	Device Clear	20	Causes all devices to be initialized to a predefined or power-up state.
SDC	Selected Device Clear	4	Causes a specified device to be initialized to a predefined or power-up state.

Table 9-2. Multiple-Line Messages (continued)

Mnemonic	Message Name	Decimal Value	Response
SPD	Serial Poll Disable	25	Devices exit serial poll mode to send their status byte.
SPE	Serial Poll Enable	24	Devices enter serial poll mode and are allowed to send their status byte when addressed to talk.
GET	Group Execute Trigger	8	Signals one or more devices to simultaneously initiate a set of device dependent actions.
TCT	Take Control	9	Passes bus controller responsibilities from the current controller to a device that can assume the bus supervisory role.
SCG	Secondary Command Group	96-127	A group of 32 commands that are only recognized if they immediately follow a talk or listen address.

Bus Sequences

HP-IB messages can be arranged to form bus sequences. A bus sequence is a series of messages constructed to accomplish a particular I/O task. BASIC I/O statements such as OUTPUT and ENTER operate by sending a predefined sequence of HP-IB messages over the bus. The table below illustrates the command bus sequence produced by the following OUTPUT statement:

```
10 OUTPUT 705;"HEWLETT-PACKARD INTERFACE BUS"
```

Table 9-3. Typical OUTPUT Bus Sequence

ASCII	Binary	Decimal	Mnemonic	ATN Line
^	01011110	94	MTA (TA30	True
?	00111111	63	UNL	True
%	00100101	37	LAG (LA5)	True
H	01001000	72	(DATA)	False
E	01000101	69	(DATA)	False
W	01010111	87	(DATA)	False
⋮	⋮	⋮	⋮	⋮
B	01000010	66	(DATA	False
U	01010101	85	(DATA)	False
S	01010011	83	(DATA)	False
(CR)	00001101	13		False
(LF)	00001010	10		False

NOTE

The default address of the interface itself is 30. Thus, MTA ("My Talk Address") is equivalent to TA30, and addresses the interface to talk. In the same way, MLA ("My Listen Address") is equivalent to LA30 if the default address of the interface has not been changed.

An analogous bus sequence is generated by the following ENTER statement (assuming the same data):

```
10 ENTER 705;A$
```

Table 9-4. Typical ENTER Bus Sequence

ASCII	Binary	Decimal	Mnemonic	ATN Line
?	00111111	63	UNL	True
>	00111110	62	MLA (LA30)	True
E	01000101	69	TAG (TA5)	True
H	01001000	72	(DATA)	False
E	01000101	69	(DATA)	False
W	01010111	87	(DATA)	False
⋮	⋮	⋮	⋮	⋮
B	01000010	66	(DATA)	False
U	01010101	85	(DATA)	False
S	01010011	83	(DATA)	False
(CR)	00001101	13		False
(LF)	00001010	10		False

Normally, you do not need to know the exact bus sequence of an OUTPUT, ENTER, or similar statement. However, you may have an application that requires a non-standard sequence of HP-IB messages. You can use the SEND statement to send a *custom bus sequence*. The following SEND statement sends the HP-IB command messages GTL, UNT, and UNL, addresses the device at primary address 01 to listen, addresses the interface itself to talk, then sends the data bytes "ABC" followed by a CR/LF EOL sequence:

```
10 SEND 7; CMD 1 UNT UNL LISTEN 1 MTA DATA "ABC" EOL
```

The bus sequence can be represented as follows:

Table 9-5. Custom Bus Sequence

ASCII	Binary	Decimal	Mnemonic	ATN Line
(SOH)	00000001	1	GTL	True
_	01011111	95	UNT	True
?	00111111	63	UNL	True
!	00100001	33	LA1	True
^	01011110	94	MTA (TA30)	True
A	01000001	65	(DATA)	False
B	01000010	66	(DATA)	False
C	01000011	67	(DATA)	False
(CR)	00001101	13		False
(LF)	00001010	10		False

Note that some commands may be specified by mnemonic in a SEND statement (for example, UNL and UNT). Other commands must be specified by the decimal number representing the command byte (for example, GTL = decimal "1"). The complete syntax of the SEND statement is given in the *HP-UX Technical BASIC Reference Manual.*

Polling

The active controller can determine the operating status of other devices on the bus by conducting a polling operation. Two forms are available: the serial poll and the parallel poll. Polling is used to determine the cause of an SRQ interrupt.

Serial Polling

A serial poll permits the active controller to obtain a status byte from any HP-IB device that supports the serial poll function. When the controller detects a service request (SRQ line is set true), it may serially poll the bus devices expected to have requested service. When a device is polled, it always returns its status byte. Bit 6 of the status byte will be set to a "1" if the device requested service; otherwise, the bit will be equal to "0". The other bits of the status byte can be used to indicate the reason for the service request, and are device-dependent. You may use the following method to detect an SRQ message:

```
100 STATUS 7,2;Reg2
110 IF BIT(Reg2,5) THEN GOTO Spoll_Device
```

There may only be one bus device capable of requesting service, and perhaps for only one purpose (for example, a printer out of paper). In this case, a serial poll may not be needed. However, if a single device can assert SRQ for various reasons, or if more than one bus device is capable of asserting SRQ, a serial poll can be used to determine which device(s) requested service and why.

When your computer is the active controller, it can initiate a serial poll of an HP-IB device by executing the SPOLL statement. The SPOLL statement specifies the device selector of the device to be polled and causes the HP-IB interface to send the following sequence over the bus:

1. ATN is set true.
2. UNL (Unlisten) command is sent to the device.
3. The interface addresses itself to listen and the peripheral device to talk.
4. SPE (Serial Poll Enable) is sent over the bus, followed by ATN going false. The device being polled then sends its status byte. If it is the only device requesting service, SRQ is removed. If it is not the only device asserting SRQ (or if it didn't request service), SRQ remains true, indicating some other device requires service.
5. ATN is set true again followed by the SPD (Serial Poll Disable) command.
6. The UNT (Untalk) command is then sent over the bus causing the peripheral device to cease being a talker.

Note in the above sequence that SRQ may or may not be removed when a device is polled. SRQ is removed if the device being polled is the one and only device requesting service. Therefore, your program must specify the order in which the bus devices are polled and serviced. An example of a serial poll routine is given in the chapter "BASIC Programming With the HP-IB Interface."

If your computer is not the active controller, you may allow it to request service with the REQUEST statement followed by a program dependent status byte. Bit 6 of this status byte maps directly to SRQ on the bus. The computer provides this status byte when it is serially polled by the active controller. If it was requesting service, bit 6 will be set to a "1"; otherwise, it will be "0". The computer removes SRQ when it sends its status byte if it is the one and only bus device requesting service. However, if SRQ is removed, bit 6 remains set until changed by the program.

Parallel Polling

In a parallel poll each bus device is assigned one of the DIO (data) lines for identification purposes. Most Hewlett-Packard bus devices capable of responding to a parallel poll are configured to assert the DIO line that corresponds to its primary address. Primary addresses 00 through 07 correspond to data lines DIO1 through DIO8, respectively. Thus, in a parallel poll: device 0 asserts DIO1, device 1 asserts DIO2, and so forth.

The interface itself can respond to a parallel poll if some other device is the active controller.

You can conduct a parallel poll by executing the PPOLL statement. This statement initiates the following sequence on the bus:

1. The ATN and EOI lines are set true.
2. The bus device(s) that requested service assert the appropriate DIO line(s).
3. After waiting a sufficient time for the polled devices to assert their lines, the resulting parallel poll response byte is read.
4. The ATN and EOI lines go false, causing the polled devices to drop their DIO lines.

Once the parallel poll response byte has been obtained, you can use the BIT function to test the eight bits and determine which device(s) originated the service request. A parallel poll can be much faster than a serial poll because the controller reads DIO1 through DIO8 all at once. However, only eight devices can be differentiated, and you may still need to use a serial poll to determine why a device requested service.

Using the GPIO Interface 10

This chapter presents some *selected programming techniques* in Technical BASIC for the GPIO interface. Refer to the appendix for a description of the status and control registers for this interface. Refer to the owner's manual supplied with your interface for installation procedures and technical information.

Chapter Contents

This chapter describes the use of the GPIO interface. It contains the following tasks and topics:

Programming With the GPIO Interface

This section discusses the `ENTER`, `OUTPUT`, `STATUS`, `CONTROL`, and `RESET` statements as used with the GPIO interface. "GPIO" stands for "General Purpose Input and Output" and reflects the versatile nature of the interface. The GPIO interface is a parallel interface. You should become familiar with the technical details of the GPIO interface before attempting to program it (refer to the GPIO interface owner's manual).

WARNING

TO AVOID POSSIBLE ELECTRICAL SHOCK AND/OR EQUIPMENT DAMAGE, FOLLOW THE INSTALLATION PROCEDURES IN THE GPIO INTERFACE OWNER'S MANUAL CAREFULLY WHEN CONNECTING PERIPHERAL DEVICES TO THE GPIO INTERFACE. TECHNICAL FAMILIARITY WITH THE PERIPHERAL DEVICE AND THE INTERFACE IS NECESSARY TO ACHIEVE SAFE, SUCCESSFUL OPERATION.

NOTE

Before you can perform any I/O operations with an interface, you must assign an interface select code to it (refer to the chapter "An Introduction to I/O"). The examples in this section assume that interface select code 8 has been assigned to the GPIO interface.

ENTER and OUTPUT Statements

You may use the usual methods of transfer — OUTPUT and ENTER — with the GPIO interface. Note that information for formatting using these statements can be found in the chapter in this manual entitled "Formatted I/O Operations."

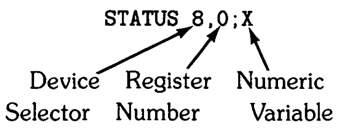
The choice of a proper OUTPUT image is only a matter of definition. You know what data is generated by your program, so you need only choose a desirable form for its output. The main problem is to avoid image overflow conditions.

The choice of an ENTER image can be easy if you know the nature of the incoming data or it can be difficult if you do not. If you **do not** know what it is, 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 decimal equivalent of the binary value of each byte will be entered. You can use the CHR$ function to determine the exact character sequence that is being received. Then, knowing the exact nature of the incoming data, you can choose an appropriate ENTER image. The following program (also found in the chapter "Formatted I/O Operations" in the section "Inspecting Unknown Data") is an example of this technique:

```
100 ! Program to inspect incoming data
110 ASSIGN 8 TO "gpio"
120 S1=8 ! Device selector for the gpio interface
130 ! Establish a terminating condition
140 SET TIMEOUT S1;3000
150 ON TIMEOUT S1 GOTO 240
160 !
170 I=1 ! Initialize counter
180 ! Input 1 byte; display analysis
190 ENTER S1 USING "#,B" ; X
200 DISP "BYTE";I;TAB(11);"VALUE =";X;TAB(24);"CHAR = ";CHR$(X)
210 I=I+1 ! Count bytes
220 GOTO 190
230 !
240 DISP "DATA INPUT HAS STOPPED"
250 RESET S1 ! Stop I/O operation
260 END
```

Checking Interface Status

You can obtain the status of an interface register by specifying the select code of the interface, the number of the desired register, and the name of a numeric variable in a **STATUS** statement. The **STATUS** statement returns the value of the specified register. The following statement obtains the value of the identification register (Status Register 0) for the GPIO interface at device selector 8:



```
STATUS 8,0;X
```

The value of Status Register 0 is always the interface's identification code (4 for the GPIO). For the other status registers, **STATUS** returns a value that depends on the status of the interface (refer to the appendix).

Interface Control

The **CONTROL** statement allows you to program the operation of the interface by writing to its control registers. Two items of general interest are:

- setting and clearing interface control lines,
- selecting an end-of-line (EOL) sequence.

Control lines are interface dependent. In general, you can set or clear interface-specific control lines by writing to Control Register 2 of the GPIO interface. Refer to the appendix for information on Control Register 2 for the GPIO interface.

CAUTION

DO NOT WRITE TO THE INTERFACE CONTROL LINE REGISTER (CONTROL REGISTER 2) UNLESS YOU ARE COMPLETELY FAMILIAR WITH THE PROTOCAL FOR YOUR INTERFACE. AN IMPROPER CONTROL LINE OPERATION MAY CAUSE A LOSS OF DATA, A DEVICE MALFUNCTION, OR DAMAGE TO YOUR INTERFACE.

Suppose that you want to set the CTLA (Control A) control line on the GPIO. You can do this by writing the value 1 to Control Register 2 for the GPIO. You can use either of the following statements (assuming select code 8):

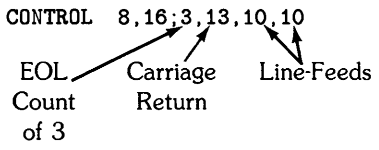
```
CONTROL 8,2;1
ASSERT 8;1
```

Both statements set the CTLA line true and *immediately* set the control line (CTLA in this case) regardless of any I/O operation in progress.

A second item of general interest in control register programming is the end-of-line (EOL) character sequence sent by the interface. This EOL sequence defaults to carriage return/line feed for all interfaces, and is sent after every PRINT or OUTPUT operation (unless you suppress it). The EOL sequence is also sent if you include the / (slash) image specifier in a PRINT or OUTPUT statement, or if you specify EOL in a SEND statement.

You may occasionally want to change the EOL sequence to suit the needs of a particular device. For example, you could use "carriage return/line feed/line feed" to obtain double spacing on a printer. For a CRT terminal that performs an automatic line-feed when a carriage return is received, you could use just a carriage return.

The GPIO interface provides EOL sequence programmability with Control Registers 16 through 23. Up to seven EOL characters can be sent. You specify the number of characters by writing to Control Register 16. You then specify the consecutive characters by writing to CR17 through CR23. For example, you could use the following CONTROL statement to program a double line-feed EOL sequence with EOI (End Or Identify) set on the last line feed.



```
CONTROL  8,16;3,13,10,10
```

NOTE

Control Registers 16 through 23 are used in exactly the same way to select an EOL sequence for the HP-IB interface. The method is similar for the HP-IB interface, but the function of Control Register 16 is slightly different (refer to the appendix).

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 register descriptions in the appendix to determine which capabilities you need and how to implement them. You may need to experiment a bit to obtain the result you desire.

The SEND Statement

In some applications you may want to send an arbitrary data sequence from the computer to one or more peripherals. You can use the SEND statement to send data to any of the interfaces. The following are examples of properly formed SEND data statements:

```
SEND 8; DATA "Welcome to the world of I/O" EOL
SEND 8; DATA "OCT. 9", "NOV. 13"
```

An end-of-line sequence is sent by the interface if you specify EOL.

The *HP-UX Technical BASIC Reference Manual* defines the syntax of the SEND statement. Note that the SEND statement, when used with GPIO interfaces, can **only** use the secondary keyword DATA, as shown above.

The RESET Statement

RESET terminates the current I/O operation and returns the interface to its power-up state. This should only be necessary when first beginning a program or when recovering from an interface failure.

The RESET command is used with the GPIO interface to reset peripherals and the interface. An example of this statement is given below:

```
RESET 8
```

Status and Control Registers A

HP-UX Technical BASIC provides a set of registers for each interface. There are three types of registers:

- *Status registers* are read-only registers. You can read these registers with the **STATUS** statement.
- *Control registers* are write-only registers. You can write to these registers with the **CONTROL** statement.
- *Status/control* registers are read/write registers. Use the **STATUS** statement to read, and the **CONTROL** statement to write to these registers.

Chapter Contents

This appendix covers the status, control, and status/control registers provided by BASIC for the HP-IB and GPIO interfaces. Tasks and topics covered in this chapter are as follows:

Tasks/Topics	Page
The HP-IB Interface	A-2
The GPIO Interface	A-7

Note that some interfaces may not implement all of the register functions documented here. For example, some HP-IB interfaces may not allow you to access all of the control lines.

You can control the operation of your interface by accessing its registers with the **STATUS** and **CONTROL** statements (refer to the chapter "Direct Interface Communication"). Some registers can also be accessed with other statements such as **ASSERT**. For example, **ASSERT 7;32** is equivalent to **CONTROL 7,2;32** for the HP-IB interface (refer to the chapter "Direct Interface Communication").

The HP-IB Interface

This section describes the registers that HP-UX Technical BASIC provides for the HP-IB interface.

Status Register 0: Interface Identification

This register always returns a value of 1 ("00000001" binary), the identification code for an HP-IB interface.

Status Register 2: HP-IB Control Lines

A bit, when set, indicates that the corresponding HP-IB control line is true.

Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Not used	Not used	SRQ	Not used	Not used	Not used	NDAC	Not used
Value=128	**Value=64**	**Value=32**	**Value=16**	**Value=8**	**Value=4**	**Value=2**	**Value=1**

NOTE

The operation of this register is hardware dependent.

Control Register 2: HP-IB Control Lines

This register gives direct access to the HP-IB control lines. When you set a bit in this register, the corresponding control line is asserted.

CAUTION

CONTROL REGISTER 2 PROVIDES DIRECT ACCESS TO THE HP-IB CONTROL LINES. YOU MUST USE THIS REGISTER WITH CARE, AND ONLY IF YOU ARE AWARE OF HP-IB PROTOCOLS! IMPROPER USE OF THIS REGISTER MAY CAUSE A BUS MALFUNCTION, AND MAY RESULT IN DAMAGE TO YOUR INTERFACE OR PERIPHERAL.

Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Not used	REN	SRQ	Not used	Not used	Not used	Not used	Not used
Value=128	**Value=64**	**Value=32**	**Value=16**	**Value=8**	**Value=4**	**Value=2**	**Value=1**

NOTE

The operation of this register is hardware dependent.

An HP-IB control line will remain true as long as the corresponding Control Register 2 bit remains set (equal to "1"). The control line goes false when the bit is cleared (equal to "0"). Follow HP-IB bus protocols carefully when using this register. For example, a non-system controller is not supposed to assert the `REN` line.

Status Register 4: HP-IB Address/System Controller

This register indicates the HP-IB address of the interface itself, and whether the interface is the system controller.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not used	Not used	System Controller	HP-IB Address				
Value=128	Value=64	Value=32	Value=16	Value=8	Value=4	Value=2	Value=1

Bits 0 through 4: HP-IB Address
These bits indicate the currently set HP-IB address of the interface.

Bit 5: System Controller
This bit, if set to a "1", indicates that the interface is the system controller.

Bits 6 and 7
These bits are always 0.

Status Register 5: HP-IB State

This register indicates the current HP-IB status of the interface. The default value is 160 (System Controller and Controller Active).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SC	LA	CA	TA	Not used	Not used	REN	Not used
Value=128	Value=64	Value=32	Value=16	Value=8	Value=4	Value=2	Value=1

Bit 1: Remote
This bit, when set, indicates that the interface is in a remote state (REN).

Bit 4: Talker Active
This bit, when set, indicates that the interface is addressed to talk (TA).

Bit 5: Controller Active
This bit, when set, indicates that the interface is Controller Active (CA).

Bit 6: Listener Active
This bit, when set, indicates that the interface is addressed to listen (LA).

Bit 7: System Controller
This bit, when set, indicates that the interface is the system controller (same as SR4 bit 5).

Control Register 8: PPOLL Value

This register controls the response to parallel poll.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not used	Not used	Not used	PPOLL Disable	PPOLL Sense	PPOLL Response Line		
Value=128	**Value=64**	**Value=32**	**Value=16**	**Value=8**	**Value=4**	**Value=2**	**Value=1**

Bits 0 through 2: PPOLL Response Line
These bits determine the data line on which the Status Bit is to be placed. For instance, if these bits are "000", then the Status Bit is to be placed on DIO1. If these bits are "111", then the response is to be placed on DIO8.

Bit 3: PPOLL Sense
This bit determines the logic sense of the Status Bit. If this bit is "0", then the "I need service" message is a "0"; if this bit is "1", the "I need service" message is "1."

Bit 4: PPOLL Disable
This bit determines whether a response will or will not be configured. A "1" tells this controller not to configure a response, and a "0" tells the controller to configure a response.

Status Register 9: HP-IB Burst Mode

This register indicates whether *burst mode* is enabled for this interface. The default value is 0.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not used	Not used	Not used	Not used	Not used	Not used	Not used	Burst mode
Value=128	**Value=64**	**Value=32**	**Value=16**	**Value=8**	**Value=4**	**Value=2**	**Value=1**

Bit 0: Burst mode
This bit, if set to a "1", indicates that *burst mode* is enabled for this interface.

Bits 1 through 7
Not used.

Control Register 9: HP-IB Burst Mode

This register is used to enable and disable burst mode for this for the HP-IB interface.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not used	Not used	Not used	Not used	Not used	Not used	Not used	Burst mode
Value=128	**Value=64**	**Value=32**	**Value=16**	**Value=8**	**Value=4**	**Value=2**	**Value=1**

Bit 0: Burst mode
Setting this bit to "1" enables *burst mode* for the HP-IB interface. Setting it to "0" disables *burst mode* for the HP-IB interface.

Bits 1 through 7
Not used.

WARNING

ENABLING *BURST MODE* LOCKS THE HP-IB INTERFACE AND **SHOULD NOT** BE USED WITH ANY INTERFACE SUPPORTING A SYSTEM DISC OR SWAP DEVICE.

Control Register 16: EOL Control

This register controls the number of characters in the EOL (end-of-line) sequence that is sent out at the end of a line of output or in response to the "/" output image specifier. You can also specify that EOI (End or Identify) be asserted with the last character of a data transfer. The default value is 2.

Bits 0 through 2: EOL Character Count

These bits specify the number of characters sent as the EOL (end-of-line) sequence. The default count is 2, which causes two characters (normally carriage return/line feed) from Control Registers 17 and 18 to be sent. You can specify up to seven EOL characters. A count of 0 specifies that no EOL sequence is to be sent.

Bits 3 through 6

Not used.

Bit 7: EOI Enable

This bit, when set, causes the EOI control line to be asserted with the last byte of a data transfer. If the EOL count is non-zero, EOI is asserted with the last character of the EOL sequence. If the EOL count is zero, EOI is asserted with the last character of the data list being sent (PRINT and SEND only).

Control Registers 17 through 23: EOL Sequence

Each of these registers specifies a character in the EOL sequence (by its ASCII decimal value). The first character is specified by Control Register 17 (default = 13, "carriage return"), the second by Control Register 18 (default = 10, "line feed"), and so forth. Control Registers 19 through 23 all have the default value "0" (no character).

You can define up to seven characters for an EOL sequence. To set up an EOL sequence for double-spaced printing, for example, set the EOL character count to 3 and the EOL sequence to CR, LF, LF:

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

The GPIO Interface

This section describes the registers that HP-UX Technical BASIC provides for the GPIO interface.

Status Register 0: Interface Identification

Status Register 0 always returns the value 4 ("00000100"), the identification code for a GPIO interface.

Status Register 2: Line Status

The bits of this register indicate the *logical* states (not necessarily the electrical states) of the corresponding FLG lines. Flag lines are either "busy" or "ready." If a FLG line is busy, the corresponding bit is set ("1"). If ready, the bit is clear ("0").

Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Not used	Not used	FLGB "1"=Busy "0"=Ready	FLGA "1"=Busy "0"=Ready	Not used	Not used	Not used	Not used
Value=128	**Value=64**	**Value=32**	**Value=16**	**Value=8**	**Value=4**	**Value=2**	**Value=1**

Control Register 2: Assertion Control

You can use either the CONTROL or ASSERT statement to write to this register. If you set a bit equal to "1", the corresponding line is asserted to the true state. If you set a bit equal to "0", the corresponding line goes false.

Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Not used	Not used	Not used	Not used	Not used	CTLB	Not used	CTLA
Value=128	**Value=64**	**Value=32**	**Value=16**	**Value=8**	**Value=4**	**Value=2**	**Value=1**

Bit 0: CTLA Line
Setting this bit asserts the CTLA line true.

Bit 1
Not used.

Bit 2: CTLB Line
Setting this bit asserts the CTLB line true.

Bits 3 through 7
Not used.

Control Register 16: EOL Control

Bits 0 through 2 of this register specify the number of characters in the EOL (end-of-line) sequence. Up to seven EOL characters may be specified. The default value is 2.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not used	Not used	Not used	Not used	Not used	Number of Characters in EOL Sequence		
Value=128	**Value=64**	**Value=32**	**Value=16**	**Value=8**	**Value=4**	**Value=2**	**Value=1**

Control Registers 17 through 23: EOL Sequence

Each of these registers specifies a character in the EOL sequence (by its ASCII decimal value). The first character is specified by Control Register 17 (default = 13, "carriage return"), the second by Control Register 18 (default = 10, "line feed"), and so forth. Control Registers 19 through 23 all have the default value "0" (no character).

Index

MANUAL COMMENT CARD

HP-UX Technical BASIC

I/O Programming Techniques

for HP 9000 Computers

Manual Reorder No. 97068-90030

Name: ______________________________

Company: ______________________________

Address: ______________________________

____

Phone No: ______________

Please note the latest printing date from the Printing History (page ii) of this manual and any applicable update(s); so we know which material you are commenting on ______________.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

MANUAL COMMENT CARD

HP-UX Technical BASIC

I/O Programming Techniques

for HP 9000 Computers

Manual Reorder No. 97068-90030

Name: ____________________

Company: ____________________

Address: ____________________

____

Phone No: ____________

Please note the latest printing date from the Printing History (page ii) of this manual and any applicable update(s); so we know which material you are commenting on ____________.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

HEWLETT
PACKARD

HP Part Number
97068-90030
Microfiche No. 97068-99030
Printed in U.S.A. 2/86

97068-90603
For Internal Use Only