# BASIC 5.0 Interfacing Techniques

## Vol. 1: General Topics

### HP 9000 Series 200/300 Computers

HP Part Number 98613-90022


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

## NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

## WARRANTY

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

# Printing History

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

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

January 1987...Edition 1

# Table of Contents

**Chapter 5: Entering Data**



**Chapter 6: Registers**

# Table of Contents

# Manual Overview 1

## Introduction

This manual is intended to present the concepts of computer interfacing that are relevant to programming the HP Series 200/300 computers. However, it is not a text dealing with computer architecture or hardware in general. It is intended to present the topics that will increase your understanding of interfacing devices to these computers. If you would like a more detailed discussion of general hardware interfacing concepts, you may want to consult a text on computer architecture.

## Manual Organization

This manual is organized by topics. The text is arranged to focus your attention on interfacing concepts rather than to present only a serial list of the BASIC-language I/O statements. Once you have read this manual and are familiar with the general and specific concepts involved, you can use either this manual or the *BASIC Language Reference* when searching for a particular detail of how a statement works. Keep in mind that this manual has been designed as a *learning tool*, not as a *reference*.

### Where to Begin

This manual is designed for easy access by both experienced programmers and beginners.

- Less experienced users may want to begin with Chapter 2, "Interfacing Concepts", before reading about general or interface-specific techniques.
- Experienced users may decide to go directly to the chapter that describes the particular interface to be used (such as HP-IB or GPIO). It is also usually helpful to become familiar with display and keyboard I/O operations, since these are helpful in seeing results while testing I/O programs.
- If more background is required, the information in chapters 3 through 8 will provide further explanation.

The brief descriptions in the next section will help you determine which chapters you will need to read for your particular application.

# Chapter Previews

### Chapter 2: Interfacing Concepts

This chapter presents a brief explanation of relevant interfacing concepts and terminology. This discussion is especially useful for beginners as it covers much of the "why" and "how" of interfacing. Experienced programmers may also want to skim this material to better understand the terminology used in this manual.

### Chapter 3: Directing Data Flow

This chapter describes how to specify which computer resource is to send data to or receive data from the computer. The use of device selectors, string variable names, and "I/O path names" in I/O statements are described.

### Chapter 4: Outputting Data

This chapter presents methods of outputting data to devices. All details of this process are discussed, and several examples of free-field output and output using images are given. Since this chapter completely describes outputting data to devices, you may only need to read the sections relevant to your application.

### Chapter 5: Entering Data

This chapter presents methods of entering data from devices. All details of this process are discussed, and several examples of free-field enter and enter using images are given. As with Chapter 4, you may only need to read sections of this chapter relevant to your application.

### Chapter 6: Registers

This chapter describes the use and access of registers. The uses of registers are explained, and programming techniques used to examine and change register contents are presented. Individual interface register definitions are not contained in this chapter, but are discussed in the corresponding interface chapter.

### Chapter 7: Interrupts and Timeouts

This chapter describes event-initiated branching from an interface's point of view. The uses of both interrupts and timeouts are discussed, and several examples are given. Again, the interface-dependent details are not given in this chapter, but are covered in the chapter dedicated to discussing programming techniques for each interface.

### Chapter 8: I/O Path Attributes

This chapter presents several powerful capabilities of the I/O path names provided by the BASIC language system. Interfacing to devices is compared to interfacing to mass storage files, and the benefits of using the same statements to access both types of resources are explained. This chapter is also highly recommended to all readers.

### Chapter 9: Advanced Transfer Techniques

This chapter describes advanced I/O techniques which can be used when communicating with devices. These techniques are generally used with devices which have data-transfer rates either much faster or much slower than the computer's normal transfer rate(s).

### Chapter 10: Display Interfaces

This chapter describes accessing your CRT display through its interface to the computer. Since these devices can be accessed like most other interfaces (via OUTPUT, ENTER, CONTROL, and STATUS), most of the programming techniques presented in Chapters 3 through 9 are applicable to these devices. **If you have no experience in programming interfaces, you will find this chapter very useful; many tools are presented that will help you program and understand the other interfaces.**

### Chapter 11: Keyboard Interfaces

As with the display chapter, this chapter describes several programming techniques applicable to interfacing to the keyboards available with Series 200/300 computers.

### Chapter 12: The HP-IB Interface

This chapter describes programming techniques specific to the HP-IB interface. Details of HP-IB communications processes are also included to promote better overall understanding of how this interface may be used.

### Chapter 13: RS-232 Serial Interface

This chapter describes programming techniques specific to using the asynchronous-protocol capabilities of the HP 98626 and HP 98644 Serial Interfaces, as well as the built-in serial interfaces of some computer models (Models 216, 217, 310, etc).

### Chapter 14: The Datacomm Interface

This chapter describes the HP 98628 Data Communications Interface and presents programming techniques for using the asynchronous or HP Data Link protocols provided by this interface.

### Chapter 15: Powerfail Protection

This chapter describes programming techniques for achieving powerfail protection (Option 050, only available on Models 226 and 236, is required to use these capabilities).

### Chapter 16: The GPIO Interface

This chapter describes programming techniques specific to using the HP 98622 GPIO Interface.

### Chapter 17: The BCD Interface

This chapter describes programming techniques specific to using the HP 98623 BCD Interface. Using this interface requires AP2.0.

### Chapter 18: EPROM Programming

This chapter describes how to program EPROMs (erasable programmable read only memory) using the HP 98255 EPROM Memory Cards and HP 98253 EPROM Programmer Card.

### Chapter 19: The HP-HIL Interface

This chapter describes how to access HP-HIL (Human Interface Link) devices from a low level. The chapter lists the categories of HP-HIL devices, and shows which ones already have BASIC drivers and which ones do not. For the devices that do *not* have BASIC drivers, this chapter describes how to write device drivers.

# Table of Contents

# Interfacing Concepts 2

This chapter describes the functions and requirements of interfaces between the computer and its resources. Concepts in this chapter are presented in an informal manner. **All** levels of programmers can gain useful background information that will increase their understanding of the **why** and **how** of interfacing.

## Terminology

These terms are important to your understanding of the text of this manual. The purpose of this section is to make sure that our terms have the same meanings.

*computer* is herein defined to be the processor, its support hardware, and the BASIC-language operating system; together these system elements **manage** all computer resources.

*hardware* describes both the electrical connections and electronic devices that make up the circuits within the computer; any piece of hardware is an actual physical device.

*software* describes the user-written, BASIC-language programs.

*firmware* refers to the pre-programmed, machine-language programs that are invoked by BASIC-language statements and commands. As the term implies, firmware is not usually modified by BASIC users. The machine-language routines of the operating system are firmware programs.

*computer resource* is herein used to describe all of the "data-handling" elements of the system. Computer resources include: internal memory, CRT display, keyboard, and disc drive, and any external devices that are under computer control.

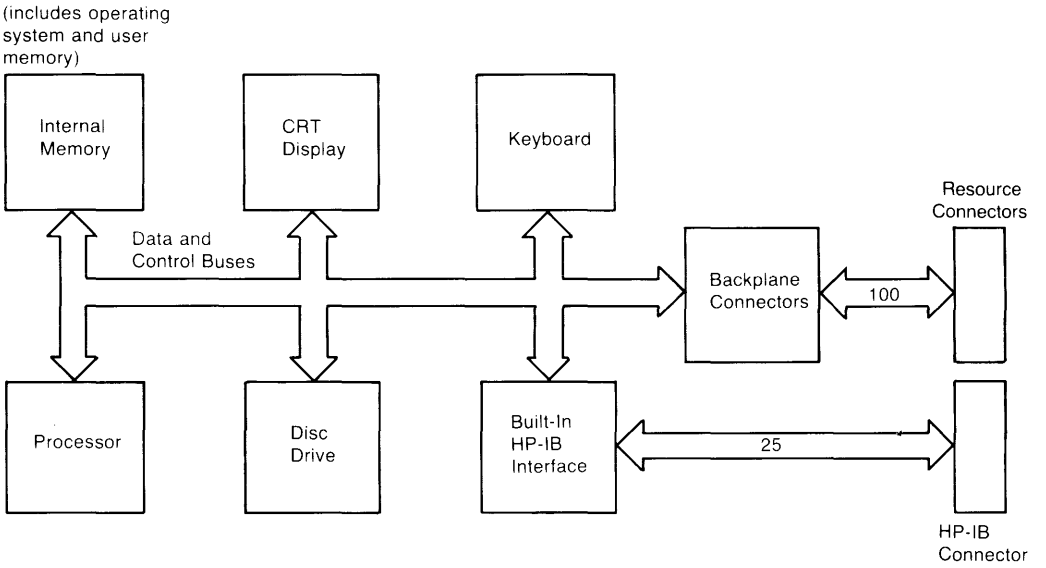



**Figure 2-1. Block Diagram of the Computer**

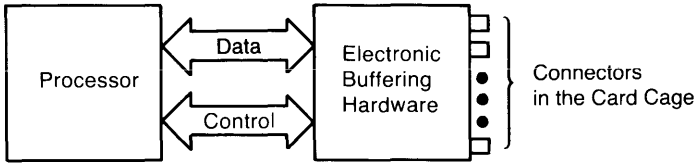
*I/O* is an acronym that comes from "Input and Output"; it refers to the process of copying data to or from computer memory.

*output* involves moving data from computer memory to another resource. During output, the *source* of data is computer memory and the *destination* is any resource, including memory.

*input* is moving data from a resource to computer memory; the source is any resource and the destination is a variable in computer memory. *Inputting data is also referred to as "entering data" in this manual* for the sake of avoiding confusion with the INPUT statement.

*bus* refers to a common group of hardware lines that are used to transmit information between computer resources. The computer communicates directly with the internal resources through the data and control buses.

*computer backplane* is an extension of these internal data and control buses. The computer communicates indirectly with the external devices through interfaces connected to the backplane hardware.




The Processor Communicates with the Interfaces through Backplane Hardware

**Figure 2-2. Backplane Hardware**

## Why Do You Need an Interface?

The primary function of an interface is, obviously, to provide a communication path for data and commands between the computer and its resources. Interfaces act as intermediaries between resources by handling part of the "bookkeeping" work, ensuring that this communication process flows smoothly. The following paragraphs explain the need for interfaces.

First, even though the computer backplane is driven by electronic hardware that generates and receives electrical signals, this hardware was not designed to be connected directly to external devices. The electronic backplane hardware has been designed with specific electrical logic levels and drive capability in mind.

**CAUTION**

EXCEEDING BACKPLANE HARDWARE RATINGS WILL DAMAGE THIS ELECTRONIC HARDWARE.

Second, you cannot be assured that the connectors of the computer and peripheral are compatible. In fact, there is a good probability that the connectors may not even mate properly, let alone that there is a one-to-one correspondence between each signal wire's function.

Third, assuming that the connectors and signals are compatible, you have no guarantee that the data sent will be interpreted properly by the receiving device. Some peripherals expect single-bit serial data while others expect data to be in 8-bit parallel form.

Fourth, there is no reason to believe that the computer and peripheral will be in agreement as to when the data transfer will occur; and when the transfer does begin the transfer rates will probably not match. As you can see, interfaces have a great responsibility to oversee the communication between computer and its resources. The functions of an interface are shown in the following block diagram.

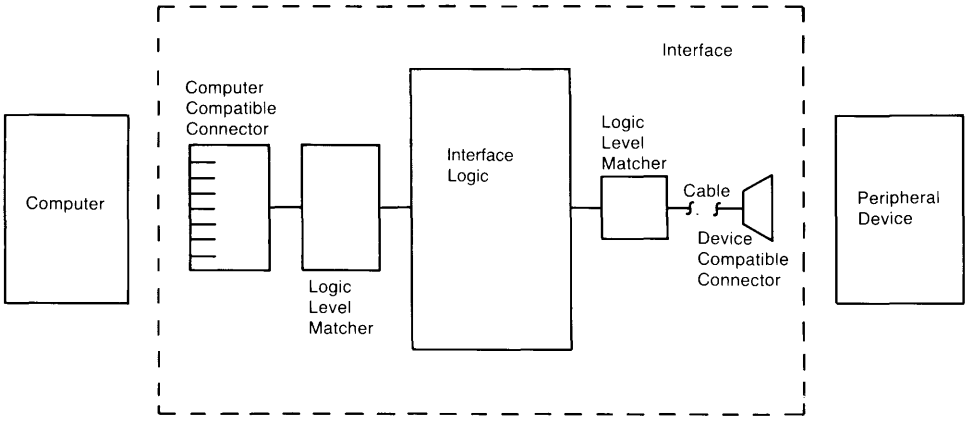



**Figure 2-3. Functional Diagram of an Interface**

## Electrical and Mechanical Compatibility

Electrical compatibility must be ensured before any thought of connecting two devices occurs. Often the two devices have input and output signals that do not match; if so, the interface serves to match the electrical levels of these signals before the physical connections are made.

Mechanical compatibility simply means that the connector plugs must fit together properly. All of the 9826 interfaces have 100-pin connectors that mate with the computer backplane. The peripheral end of the interfaces may have unique configurations due to the fact that several types of peripherals are available that can be operated with the 9826. Most of the interfaces have cables available that can be connected directly to the device so you don't have to wire the connector yourself.

## Data Compatibility

Just as two people must speak a common language, the computer and peripheral must agree upon the form and meaning of data before communicating it. As a programmer, one of the most difficult compatibility requirements to fulfill before exchanging data is that the format and meaning of the data being sent is identical to that anticipated by the receiving device. Even though some interfaces format data, most interfaces have little responsibility for matching data formats; most interfaces merely move agreed-upon quantities of data to or from computer memory. The computer must generally make the necessary changes, if any, so that the receiving device gets meaningful information.

## Timing Compatibility

Since all devices do not have standard data-transfer rates, nor do they always agree as to when the transfer will take place, a consensus between sending and receiving device must be made. If the sender and receiver can agree on both the transfer rate and beginning point (in time), the process can be made readily.

If the data transfer is not begun at an agreed-upon point in time and at a known rate, the transfer must proceed one data item at a time with acknowledgement from the receiving device that it has the data and that the sender can transfer the next data item; this process is known as a "handshake". Both types of transfers are utilized with different interfaces and both will be fully described as necessary.

## Additional Interface Functions

Another powerful feature of some interface cards is to relieve the computer of low-level tasks, such as performing data-transfer handshakes. This distribution of tasks eases some of the computer's burden and also decreases the otherwise-stringent response-time requirements of external devices. The actual tasks performed by each type of interface card vary widely and are described in the next section of this chapter.

# Interface Overview

Now that you see the need for interfaces, you should see what kinds of interfaces are available for the computer. Each of these interfaces is specifically designed for specific methods of data transfer; each interface's hardware configuration reflects its function.

## The HP-IB Interface

This interface is Hewlett-Packard's implementation of the IEEE-488 1978 Standard Digital Interface for Programmable Instrumentation. The acronym "HP-IB" comes from Hewlett-Packard Interface Bus, often called the "bus".

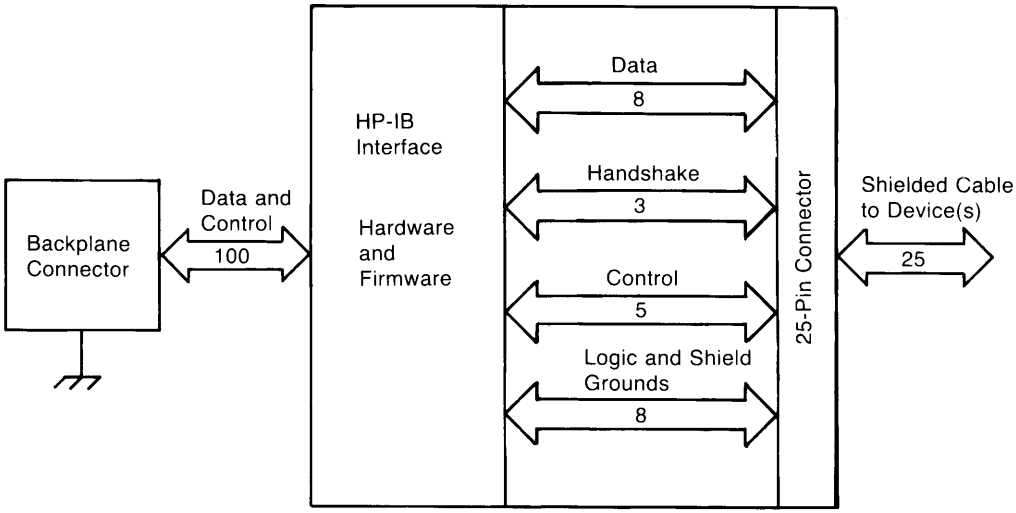



**Figure 2-4. Block Diagram of the HP-IB Interface**

The HP-IB interface fulfills all four compatibility requirements (hardware, electrical, data, and timing) with no additional modification. Just about all you need to do is connect the interface cable to the desired HP-IB device and begin programming. All resources connected to the computer through the HP-IB interface must adhere to this IEEE standard.

The "bus" is somewhat of an independent entity; it is a communication arbitrator that provides an organized protocol for communications between several devices. The bus can be configured in several ways. The devices on the bus can be configured to act as senders or receivers of data and control messages, depending on their capabilities.

## The RS-232C Serial Interface

The serial interface changes 8-bit parallel data into bit-serial information and transmits the data through a two-wire (usually shielded) cable; data is received in this serial format and is converted back to parallel data. This use of two wires makes it more economical to transmit data over long distances than to use 8 individual lines.

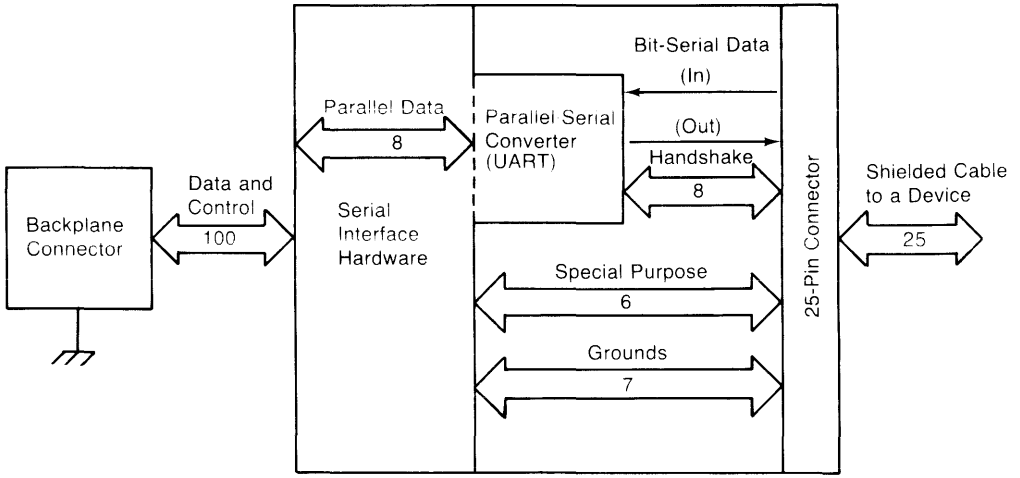



**Figure 2-5. Block Diagram of the Serial Interface**

Data is transmitted at several programmable rates using either a simple data handshake or no handshake at all. The main use of this interface is in communicating with simple devices.

## The Datacomm Interface

This interface also changes 8-bit parallel data into bit-serial data (and vice versa) in a manner similar to the serial interface described above. However, the datacomm interface is controlled by a Z-80A microprocessor resident or the interface board, which implements high-level features such as inbound and outbound data buffers and the use of control blocks. The datacomm interface is intended for general data communications applications, most of which cannot be adequately handled by the serial interface.

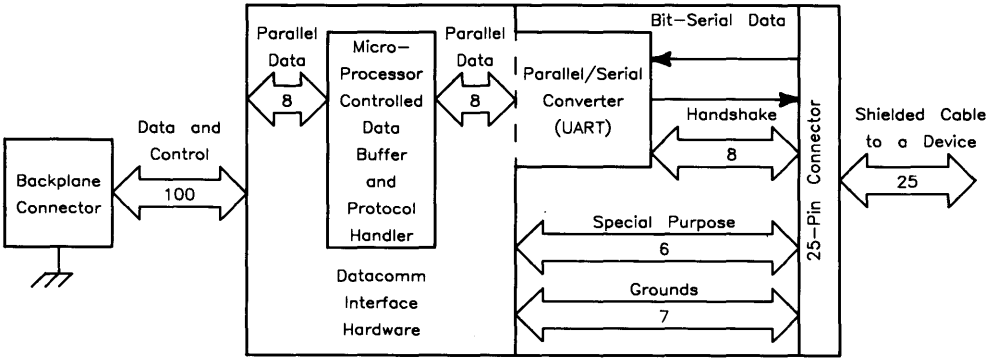



**Figure 2-6. Block Diagram of the Datacomm Interface**

## The GPIO Interface

This interface provides the most flexibility of all the interfaces. It consists of 16 output-data lines, 16 input-data lines, two handshake lines, and other assorted control lines. Data is transmitted using programmable handshake conventions and logic senses.

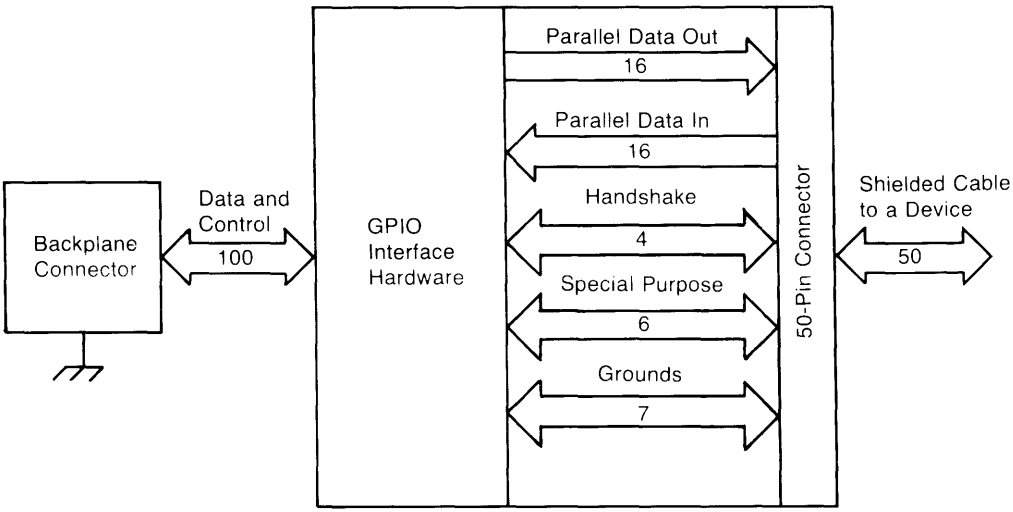



**Figure 2-7. Block Diagram of the GPIO Interface**

## The BCD Interface

This interface is designed to be used with peripheral devices that implement a binary-coded decimal (BCD) data representation. Forty input lines allow up to ten BCD characters to be entered with one handshake cycle. Eight lines are available for data output. The interface provides great flexibility by allowing two peripheral devices to be connected and by featuring a binary-data operating mode.

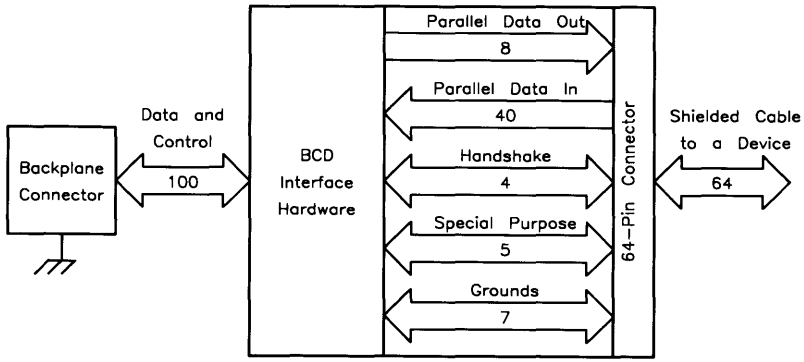



**Figure 2-8. Block Diagram of the BCD Interface**

# Data Representations

As long as data is only being used internally, it really makes little difference how it is represented: the computer always understands its own representations. However, when data is to be moved to or from an external resource, the data representation is of paramount importance.

## Bits and Bytes

Computer memory is no more than a large collection of individual bits (**bi**nary digi**ts**), each of which can take on one of two logic levels (high or low). Depending on how the computer interprets these bits, they may mean on or not on (off), true or not true (false), one or zero, busy or not busy, or any other bi-state condition. These logic levels are actually voltage levels of hardware locations within the computer. The following diagram shows the voltage of a point versus time and relates the voltage levels to logic levels.

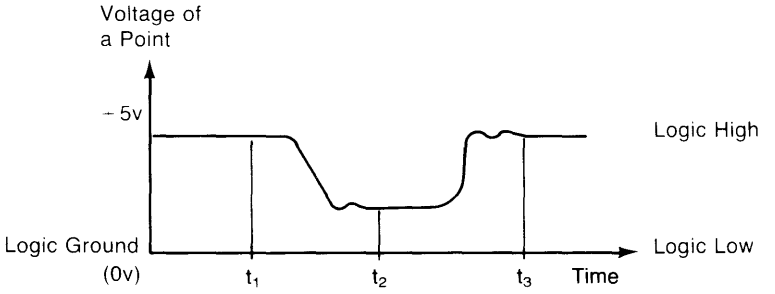



**Figure 2-9. Voltage and Positive-True Logic**

In some cases, you want to determine the state of an individual bit (of a variable in computer memory, for instance). The logical binary functions (BIT, BINCMP, BINIOR, BINEOR, BINAND, ROTATE, and SHIFT) provide access to the individual bits of data.

In most cases, these individual bits are not very useful by themselves, so the computer groups them into multiple-bit entities for the purpose of representing more complex data. Thus, all data in computer memory are somehow represented with binary numbers.

The computer's hardware accesses groups of sixteen bits at one time through the internal data bus; this size group is known as a *word*. With this size of bit group, 65 536 (=2↑16) different bit patterns can be produced. The computer can also use groups of eight bits at a time; this size group is known as a *byte*. With this smaller size of bit group, 256 (=2↑8) different patterns can be produced. How the computer and its resources interpret these combinations of ones and zeros is very important and gives the computer all of its utility.

## Representing Numbers

The following binary weighting scheme is often used to represent numbers with a single data byte. Only the non-negative integers 0 through 255 can be represented with this particular scheme.

Most-Significant Bit Least-Significant Bit

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| Value=128 | Value=64 | Value=32 | Value=16 | Value=8 | Value=4 | Value=2 | Value=1 |

Notice that the value of a 1 in each bit position is equal to the power of two of that position. For example, a 1 in the 0th bit position has a value of 1 (=2↑0), a 1 in the 1st position has a value of 2 (=2↑1), and so forth. The number that the byte represents is then the total of all the individual bit's values.

### Determining the Number Represented

$0 \times 2^0 = 0$
$1 \times 2^1 = 2$
$1 \times 2^2 = 4$
$0 \times 2^3 = 0$
$1 \times 2^4 = 16$
$0 \times 2^5 = 0$
$0 \times 2^6 = 0$
$1 \times 2^7 = 128$

Number represented =

$2 + 4 + 16 + 128 = 150$

The preceding representation is used by the "NUM" function when it interprets a byte of data. The next section explains why the character "A" can be represented by a single byte.

```
100  Number=NUM("A")
110  PRINT " Number = ";Number
120  END
```

**Printed Result**

```
Number = 65
```

## Representing Characters

Data stored for humans is often alphanumeric-type data. Since less than 256 characters are commonly used for general communication, a single data byte can be used to represent a character. The most widely used character set is defined by the ASCII standard[1]. This standard defines the correspondence between characters and bit patterns of individual bytes. Since this standard only defines 128 patterns (bit 7 = 0), 128 additional characters are defined by the computer (bit 7 = 1). The entire set of the 256 characters on the computer is hereafter called the "extended ASCII" character set.

When the CHR$ function is used to interpret a byte of data, its argument must be specified by its binary-weighted value. The single (extended ASCII) character returned corresponds to the bit pattern of the function's argument.

```
100   Number=65              !  Bit pattern is "01000001"
110   PRINT " Character is ";
120   PRINT CHR$(Number)
130   END
```

**Printed Result**

```
Character is A
```

## Representing Signed Integers

There are two ways that the computer represents signed integers. The first uses a binary weighting scheme similar to that used by the NUM function. The second uses ASCII characters to represent the integer in its decimal form.

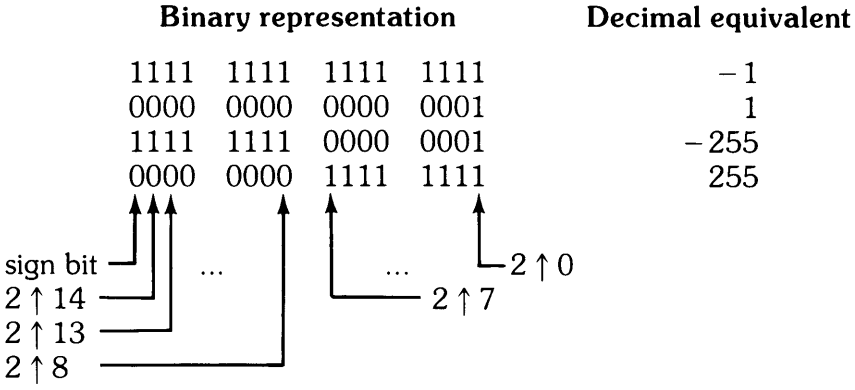
### Internal Representation of Integers

Bits of computer memory are also used to represent signed (positive and negative) integers. Since the range allowed by eight bits is only 256 integers, a word (two bytes) is used to represent integers. With this size of bit group, 65536 (=2↑16) unique integers can be represented.

The range of integers that can be represented by 16 bits can arbitrarily begin at any point on the number line. In the computer, this range of integers has been chosen for maximum utility; it has been divided as symmetrically as possible about zero, with one of the bits used to indicate the sign of the integer.

[1] ASCII stands for "American Standard Code for Information Interchange". See the Useful Tables appendix in the *BASIC Language Reference* for the complete table.

With this "2's-complement" notation, the most significant bit (bit 15) is used as a sign bit. A sign bit of 0 indicates positive numbers and a sign bit of 1 indicates negatives. You still have the full range of numbers to work with, but the range of absolute magnitudes is divided in half (−32768 through 32767). The following 16-bit integers are represented using this 2's-complement format.



The representation of a positive integer is generated according to place value, just as when bytes are interpreted as numbers. To generate a negative number's representation, first derive the positive number's representation. Complement (change the ones to zeros and the zeros to ones) all bits, and then to this result add 1. The final result is the two's-complement representation of the negative integer. This notation is very convenient to use when performing math operations. Let's look at a simple addition of 2 two's-complement integers.

**Example: 3 + (−3) = ?**

| | | | | | |
|---|---|---|---|---|---|
| First, +3 is represented as: | 0000 | 0000 | 0000 | 0011 | |
| Now generate −3's representation: | | | | | |
| first complement +3, | 1111 | 1111 | 1111 | 1100 | |
| then add 1 | + 0000 | 0000 | 0000 | 0001 | |
| −3's representation: | 1111 | 1111 | 1111 | 1101 | |
| Now add the two numbers: | 1111 | 1111 | 1111 | 1101 | |
| | + 0000 | 0000 | 0000 | 0011 | |
| | 1← | | | 1← | carry on |
| final carry not used | 0000 | 0000 | 0000 | 0000 | all places |

### ASCII Representation of Integers

ASCII digits are often used to represent integers. In this representation scheme, the decimal (rather than binary) value of the integer is formed by using the ASCII digits 0 through 9 {CHR$(48) through CHR$(57), respectively}. An example is shown below.

**Example**

The decimal representation of the binary value "1000 0000" is 128. The ASCII-decimal representation consists of the following three characters.

| Character | Decimal Code | Binary Code |
|---|---|---|
| 1 | 49 | 00110001 |
| 2 | 50 | 00110010 |
| 8 | 56 | 00111000 |

## Representing Real Numbers

Real numbers, like signed integers, can be represented in one of two ways with the computers. They are represented in a special binary mantissa-exponent notation within the computers for numerical calculations. During output and enter operations, they can also be represented with ASCII-decimal digits.

### Internal Representation of Real Numbers

Real numbers are represented internally by using a special binary notation[1]. With this method, all numbers of the REAL data type are represented by eight bytes: 52 bits of mantissa magnitude, 1 bit for mantissa sign, and 11 bits of exponent. The following equation and diagram illustrate the notation; the number represented is 1/3.

| Byte | 1 | 2 | 3 | 4 | ... | 8 |
|---|---|---|---|---|---|---|
| Decimal value of character | 63 | 213 | 85 | 85 | ... | 85 |
| Binary value of characters | 00111111 | 11010101 | 01010101 | 01010101 | ... | 01010101 |

mantissa sign  exponent  mantissa

$$\text{Real number} = (-1)^{\text{mantissa sign}} * 2^{\text{exponent} - 1023} * (1.\ \text{mantissa})$$

[1] The internal representation used for real numbers is the IEEE standard 64-bit floating-point notation. For further details, consult the "Numeric Computation" chapter of the *BASIC Programming Techniques* manual.

Even though this notation is an international standard, most external devices don't use it; most use an ASCII-digit format to represent decimal numbers. The computer provides a means so that both types of representations can be used during I/O operations.

### ASCII Representation of Real Numbers

The ASCII representation of real numbers is very similar to the ASCII representation of integers. Sign, radix, and exponent information are included with ASCII-decimal digits to form these number representations. The following example shows the ASCII representation of 1/3. Even though, in this case, 18 characters are required to get the same accuracy as the eight-byte internal representation shown above, not all real numbers represented with this method require this many characters.

| ASCII characters | 0 | . | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Decimal value of characters | 48 | 46 | 51 | 51 | 51 | 51 | 51 | 51 | 51 | 51 | 51 | 51 | 51 | 51 | 51 | 51 | 51 | 51 |

# The I/O Process

When using statements that move data between memory and internal computer resources, you do not usually need to be concerned with the details of the operations. However, you may have wondered how the computer moves the data. This section takes you "behind the scenes" of I/O operations to give you a better intuitive feel for how the computer outputs and enters data.

## I/O Statements and Parameters

The I/O process begins when an I/O statement is encountered in a program. The computer first determines the type of I/O statement to be executed (such as, OUTPUT, ENTER USING, etc.). Once the type of statement is determined, the computer evaluates the statement's parameters.

### Specifying a Resource

Each resource must have a unique specifier that allows it to be accessed to the exclusion of all other resources connected to the computer. The methods of uniquely specifying resources (output destinations and enter sources) are device selectors, string variable names, and I/O path names. These specifiers are further described in the next chapter.

For instance, before executing an OUTPUT statement, the computer first evaluates the parameter which specifies the destination resource. The source parameter of an ENTER statement is evaluated similarly.

```
OUTPUT Dest_parameter;Source_item

ENTER Sourc_parameter;Dest_item
```

### Firmware

After the computer has determined the resource with which it is to communicate, it "sets up" the moving process. The computer chooses the method of moving the specified data according to the type of resource specified and the type of I/O statement. The actual machine-language routine that executes the moving procedure is in firmware. Since there are differences in how each resource represents and transfers data, a dedicated firmware routine must be used for each type of resource. After the appropriate firmware routine has been selected, the next parameter(s) must be evaluated (i.e., source items for OUTPUT statements and destination items for ENTER statements).

### Registers

The computer must often read certain memory locations to determine which firmware routines will be called to execute the I/O procedure. The content of these locations, known as registers, store parameters such as the type of data representation to be used and type of interface involved in the I/O operation.

An example of register usage by firmware is during output to the CRT. Characters output to this device are displayed beginning at the current screen coordinates. After the computer has evaluated the first expression in the source-item list, it must determine where to begin displaying the data on the screen. Two memory locations are dedicated to storing the "X" and "Y" screen coordinates. The firmware determines these coordinates and begins copying the data to the corresponding locations in display memory.

The program can also determine the contents of these registers. The statements that provide access to the registers are described in Chapter 6. The contents of all registers accessible by the program are described in the interface programming chapters.

## Data Handshake

Each byte (or word) of data is transferred with a procedure known as a data-transfer handshake (or simply "handshake"). It is the means of moving one byte of data at a time when the two devices are not in agreement as to the rate of data transfer or as to what point in time the transfer will begin. The steps of the handshake are as follows.

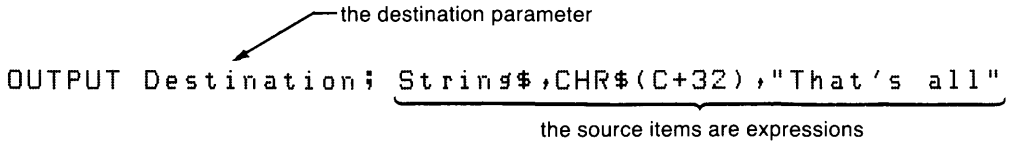
1. The sender signals to get the receiver's attention.
2. The receiver acknowledges that it is ready.
3. A data byte (or word) is placed on the data bus.
4. The receiver acknowledges that it has gotten the data item and is now busy. No further data may be sent until the receiver is ready.
5. Repeat these steps if more data items are to be moved.

# I/O Examples

Now that you have seen all of the steps taken by the computer when executing an I/O statement, let's look at how two typical I/O statements are executed by the computer.

## Example Output Statement

Data can be output to only one resource at a time with the OUTPUT statement (with the exception of the HP-IB Interface). This destination can be any computer resource, which is specified by the destination parameter as shown below.




The source of data for output operations is always memory. Either string or numeric expressions can specify the actual data to be output. The flow of data during output operations is shown below. Notice that all data copied from memory to the destination resource by the OUTPUT statement passes through the processor under the control of operating-system firmware.

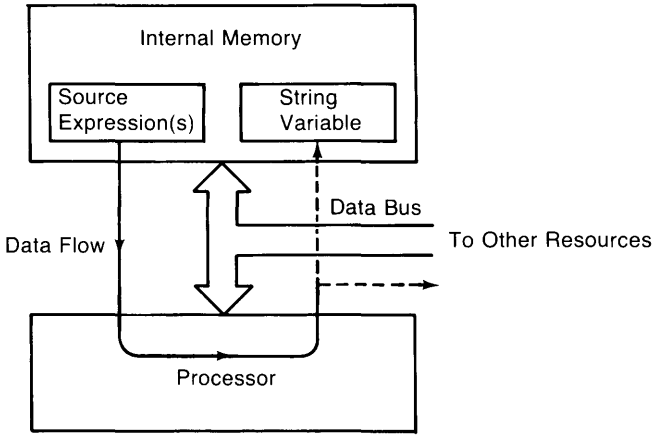



**Figure 2-10. Data Flow During Output Operations**

### Source-Item Evaluation

The source items, listed after the semicolon and separated by commas, can be any valid numeric or string expression. As the statement is being executed, these expressions must be individually evaluated and the resultant data representation sent to the specified destination. The results of the evaluation depend on the type of expression (numeric or string) and on which data representation (ASCII or internal) is to be used during the I/O operation.

If the expression is a variable **and** the internal data representation is to be used, the data is ready to be copied byte-serially (or word-serially) to the destination; otherwise, the expression must be completely evaluated. The representation generated during the evaluation is stored in a temporary variable within memory. In both cases, once the beginning memory location and length of the data are known, the copying process can be initiated.

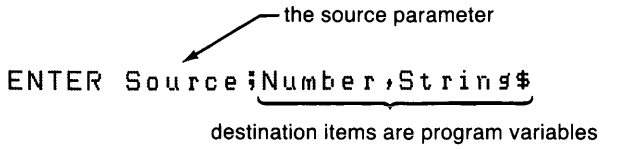
### Copying Data to the Destination

The computer employs "memory-mapped" I/O operations; all devices are addressable as memory locations. All output operations involve a series of two-step processes. The first step is to copy one byte (or word) from memory into the processor. The second step is then to copy this byte (or word) into the destination location (a memory address). Each item in the list is output in this serial fashion. The appropriate handshake firmware routine is executed for each byte (or word) to be copied.

Since there may be several data items in the source list, it may be necessary to output an item-terminator character after each item to communicate the end of the item to the receiver. If the item is the last item in the source list, the computer may signal the receiver that the output operation is complete. Either an item terminator or end-of-line sequence of characters can be sent to the receiver to signal the end of this data transmission. The OUTPUT statement is described in full detail in Chapter 4.

## Example Enter Statement

Data can be entered from only one resource at a time. This source can be any resource and is specified by the source parameter as shown in the following statement.




The destinations of enter operations are always variables in memory. Both string and numeric variables can be specified as the destinations. The flow of data during enter operations is shown below.

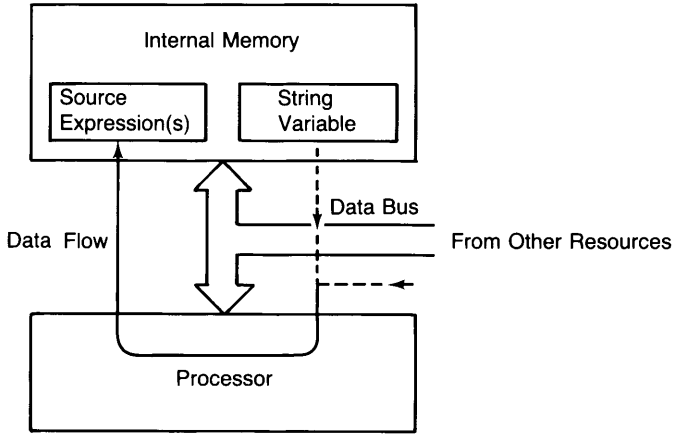



**Figure 2-11. Data Flow During Enter Operations**

### Destination-Item Evaluation

The destination(s) of data to be entered is (are) specified in the destination list. Either string or numeric variables can be specified, depending on the type of data to be entered. In general, as each destination item is evaluated, the computer finds its actual memory location so that data can be copied directly into the variable as the enter operation is executed. However, if the ASCII representation is in use, numeric data entered is stored in a temporary variable during entry.

### Copying Data into the Destinations

As with output operations, entering data is a series of two-step processes. Each data byte (or word) received from the sender is entered into the processor by the appropriate handshake firmware. It is then copied into either a temporary variable or a program variable. If more than one variable is to receive data, each incoming data item must be properly terminated. If the internal representation is in use, the computer knows how many characters are to be entered for each variable. If the ASCII representation is in use, a terminator character (or signal) must be sent to locate the end of each data item. When all data for the item has been received, it is evaluated, and the resultant internal representation of the number is placed into the appropriate program variable. Further details concerning the ENTER statement are contained in Chapter 5.

# Table of Contents

# Directing Data Flow 3

As described in the previous chapter, data can be moved between computer memory and several resources, including:

- Computer memory (BASIC string variables)
- Internal devices (such as the display and keyboard)
- Mass storage files
- External devices (such as instruments and printers)
- Buffers (variables in memory with special capabilities for high-speed, background-process transfers)

This chapter describes how string variables and devices are specified in I/O statements. Specifying mass storage files in I/O statements is briefly described in the "I/O Path Attributes" chapter of this manual, and in the "Data Storage and Retrieval" chapter of *BASIC Programming Techniques.* Buffers are described in the "Advanced Transfer Techniques" chapter of this manual.

# Specifying a Resource

Each resource must have a specifier that allows it to be accessed to the exclusion of all other computer resources. String variables are specified with their names, while devices can be specified with either their device selector or with a new data type known as an I/O path name. This section describes how to specify these resources in OUTPUT and ENTER statements.

## String-Variable Names

Data is moved to and from string variables by specifying the string variable's name in an OUTPUT or ENTER statement. Examples of each are shown in the following program.

```
100  DIM To_dest$[80],From_source$[80]
110  DIM Data_out$[80]
120  !
130  From_source$="Source data"
140  Data_out$="OUTPUT data"
150  !
160  PRINTER IS CRT
170  PRINT "To_dest$ before OUTPUT = ";To_dest$
180  PRINT
190  !
200  OUTPUT To_dest$;Data_out$; ! ";" suppresses CR/LF.
210  PRINT "To_dest$ after OUTPUT = ";To_dest$
220  PRINT
230  !
240  ENTER From_source$;To_dest$
250  PRINT "To_dest$ after ENTER = ";To_dest$
260  PRINT
270  !
280  END
```

**Printed Results**

```
To_dest$ before OUTPUT= (null string)

To_dest$ after OUTPUT= OUTPUT data

To_dest$ after ENTER= Source data
```

As with I/O operations between the computer and other resources, the source and destination of data are specified in software (in an I/O statement within a BASIC program). The data is then moved through a hardware path under operating-system firmware control. An overview of this process is illustrated in the following diagram.

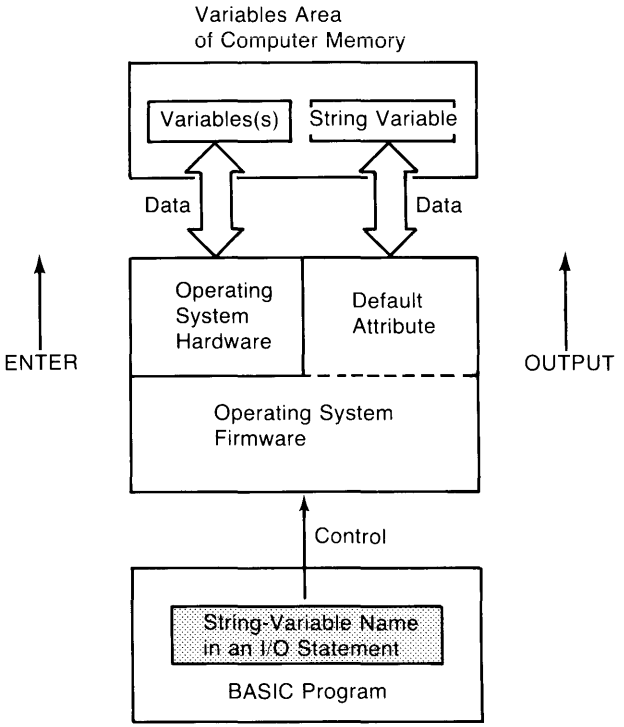



**Figure 3-1. Diagram of the Default I/O Path Used for String-Variable I/O Operations**

Data is always copied to the destination string (or from the source string) beginning at the first position of the variable; subscripts cannot be used to specify any other beginning position within the variable.

The use of outputting to and entering from string variables is a very powerful method of buffering data to be output to other resources. With OUTPUT and ENTER statements that use images, the data sent to the string variables can be explicitly formatted before being sent to (or while being received from) the variable. Further uses of string variables are described in the section of the "I/O Path Attributes" chapter called "Applications of Unified I/O".

## Device Selectors

Devices include the built-in CRT and keyboard, external printers and instruments, and all other physical entities that can be connected to the computer through an interface. Each interface has a unique number by which it is identified, known as its *interface select code*.

### Select Codes of Built-In Interfaces

The internal devices are accessed with the following, permanently assigned interface select codes.

**Table 3-1. Internal Device Select Codes**

| Built-In Interface/Device | Select Code |
|---|---|
| Alpha Display | 1 |
| Keyboard | 2 |
| Graphics Display<br>(*non*-bit-mapped alpha/graphics displays) | 3 |
| Flexible Disc Drive<br>(Models 226 and 236 only) | 4 |
| Powerfail Protection<br>(optional with Models 226 and 236 only) | 5 |
| Graphics Display<br>(*bit-mapped* alpha/graphics displays) | 6 |
| Built-in HP IB[1] | 7 |
| Built-in serial[1] | 9 |
| Parity-checking (memory),<br>cache memory,<br>and floating-point math hardware | 32 (pseudo) |

[1] Not all computer models have built-in HP IB and serial interfaces.

### Select Codes of Optional Interfaces

Optional interfaces all have switch-setable select codes. The valid range of select codes is 8 through 31 (they *cannot* use select codes 1 through 7, since these may be used by built-in devices). The following settings on optional interfaces have been made at the factory but can be reset to any unique select code between 8 and 31. See the interface's installation manual for further instructions.

**Table 3-2. Factory Settings for Interface Select Codes**

| Built-In Interface/Device | Select Code |
|---|---|
| HP-IB (HP 98624) | 8 |
| Serial (HP 98626, HP 98644) | 9[1] |
| BCD (HP 98623) | 11 |
| GPIO (HP 98622) | 12 |
| High-Speed (HP-IB) Disc (HP 98625) | 14 |
| Data Communications (HP 98628) | 20 |
| Shared Resource Manager (HP 98629) | 21 |
| EPROM Programmer (HP 98253) | 27 |
| Color Output (HP 98627) | 28 |
| Bubble Memory (HP 98259) | 30 |

Examples of using interface select codes to access devices are shown below.

```
OUTPUT 1;"Data to CRT"
ENTER CRT;Crt_line$

Int_sel_code=12
OUTPUT Int_sel_code;String$&"Expression",Num_expression
ENTER Int_sel_code;Str_variable$,Num_variable

Number=2
ENTER 7+Number;Serial_data$
OUTPUT 11-Number;"Data to serial card"
```

The device selector can be any numeric expression which rounds to an integer in the range 1 through 31. If the interface select code specifies an HP-IB interface, additional information must be specified to access a particular HP-IB device, since more than one device can be connected to the computer through HP-IB interfaces.

[1] Use another select code if there is already a built-in serial interface at this select code.

## HP-IB Device Selectors

Each device on the HP-IB interface has a **primary address** by which it is uniquely identified; each address must be unique so that only one device is accessed when one address is specified. The device selector is then a combination of the interface select code and the device's address[1]. Some examples are shown below.

**Table 3-3. HP-IB Device Selector Examples**

| Device Location | Device Selector |
|---|---|
| interface select code 7<br>at primary address 22 | 722 |
| interface select code 10<br>at primary address 13 | 1013 |
| interface select code 10<br>at primary address 01 | 1001 |

[1] The HP-IB also has additional capabilities that add to this definition of device selectors. See the chapter called "The HP-IB Interface" for further details

Accessing devices with device selectors in BASIC statements is described in the following diagram.

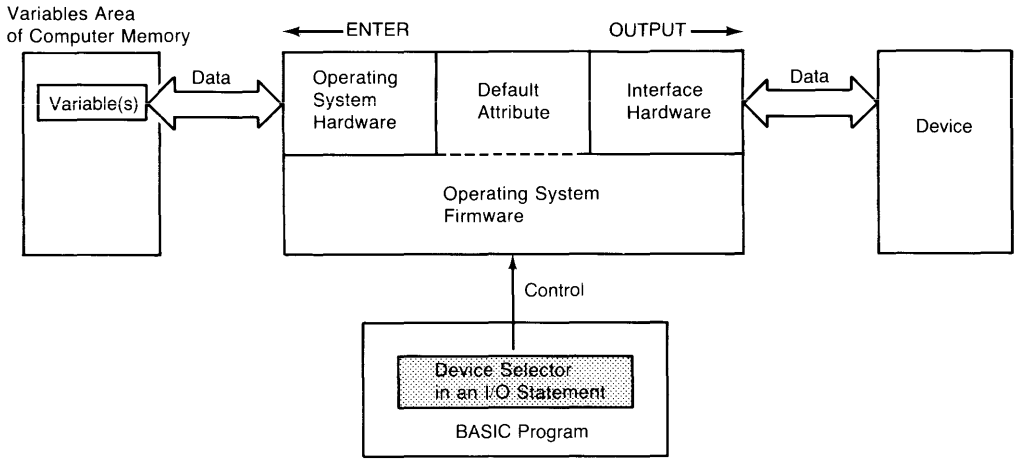



**Figure 3-2. Diagram of the Default I/O Path Used when a Device Selector is Specified**

Disc drives are also considered to be devices and are connected to the computer through interfaces. However, files on the disc media cannot be uniquely accessed with only the select code of its interface; additional information specifying which file is to be accessed must be included. Accessing mass storage files is fully described in the "Data Storage and Retrieval" chapter of the *BASIC Programming Techniques* manual; these tasks are compared to accessing devices in the "I/O Path Attributes" chapter of this manual.

## I/O Path Names

As shown in the previous diagrams, all data entered into and output from the computer is moved through an "I/O path". An I/O path consists of the hardware and operating-system firmware used to carry out this moving process. When a string variable or device selector is specified in an ENTER or OUTPUT statement, the operating system first evaluates the expression that specifies a resource and then chooses the corresponding **default** I/O path through which data will be moved.

With the I/O language of the computer, the I/O paths to devices and mass storage files can be assigned special names; I/O paths to string variables can only be assigned names if the variable is declared as a buffer. Assigning names to I/O paths provides many improvements in performance and additional capabilities over using device selectors, described in "Benefits of Using I/O Path Names" at the end of this chapter.

The concept of using I/O path names is shown in the following diagram: by comparing it to the previous diagram, you can see several major differences between using I/O path names and device selectors in I/O operations. These differences are described in the section of this chapter called "Benefits of Using I/O Path Names".

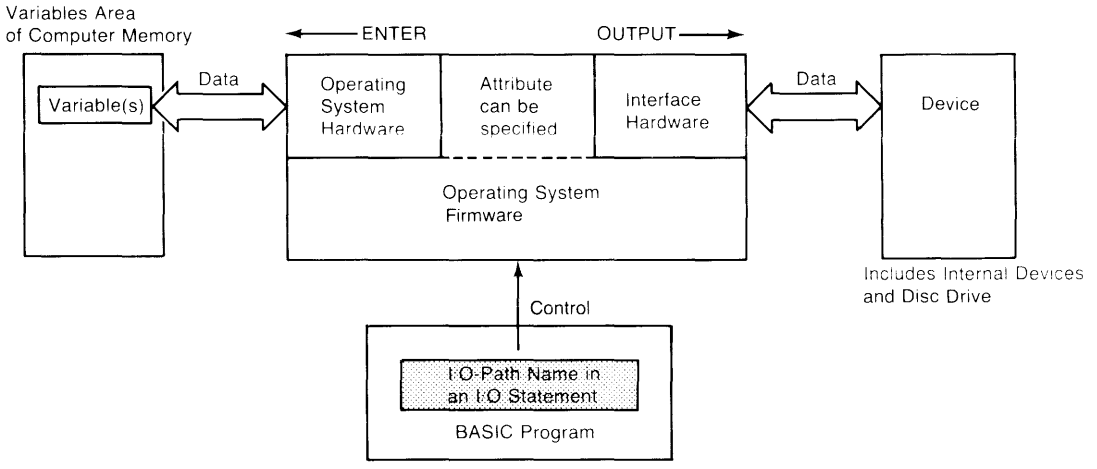



**Figure 3-3. I/O Paths to Devices and Mass-Storage Files**

# Assigning I/O Path Names

An I/O path name is a new data type that can be assigned to either a device or a data file on a mass storage device. Any valid name[1] preceded by the "@" character can be used. Examples of the statement that makes this assignment are as follows.

**Examples**

```
ASSIGN @Display TO 1

ASSIGN @Printer TO 701

ASSIGN @Serial TO 9

ASSIGN @Gpio TO 12
```

Now you can use the I/O path names instead of the device selectors to specify the resource with which communication is to take place.

```
OUTPUT @Display;"Display message"

OUTPUT @Printer;"Message to the Printer"

ENTER @Serial;Variable,Variable$

ENTER @Gpio;Word1,Word2
```

[1] A "name" is a combination of 1 to 15 characters, beginning with an uppercase alphabetical character or one of the characters CHR$(161) through CHR$(254) and followed by up to 14 lowercase alphanumeric characters, the underbar character (_), or the characters CHR$(161) through CHR$(254). Numeric-variable names are examples of valid names.

Since an I/O path name is a data type, a fixed amount of memory is allocated, or "reserved", for the variable similar to the manner in which memory is allocated for other program variables (INTEGER, REAL, and string variables). Since the variable does not initially contain usable information, the validity flag, shown below, is set to false. When the ASSIGN statement is actually executed, the allocated memory space is then filled with information describing the I/O path between the computer and the specified resource, and the validity flag is set to true.

**Table 3-4. I/O Path Variable Contents**

| | |
|---|---|
| validity flag | |
| type of resource | |
| device selector of resource | |
| additional information, if any, depends on the type of resource | |

Attempting to use an I/O path name that **does not** appear in **any** program line results in error 910 (`Identifier not found in this context`). This error message indicates that memory space has not been allocated for the variable. However, attempting to use an I/O path name that **does** appear in an ASSIGN statement in the program **but which has not yet been executed** results in error 177 (`Undefined I/O path name`). This error indicates that the memory space was allocated but the validity flag is still false; no valid information has been placed into the variable since the I/O path name has not yet been assigned to a resource.

This I/O path information is only accessible to the context in which it was allocated, unless it is passed as a parameter or appears in the proper COM statements[1]. Thus, an I/O path name cannot be initially assigned from the keyboard, and it cannot be accessed from the keyboard unless it is presently assigned within the current context. However, an I/O path name can be re-assigned from the keyboard, as described in the next section.

This information describing the I/O path is accessed by the operating system whenever the I/O path name is specified in subsequent I/O statements. A portion of this information can also be accessed with the STATUS and CONTROL statements described in the "Registers" chapter. For now, the important point is that it contains a description of the resource sufficient to allow its access.

[1] See the *BASIC Language Reference* or the "Subprograms" chapter of *BASIC Programming Techniques* for details.

## Re-Assigning I/O Path Names

If an I/O path name already assigned to a resource is to be re-assigned to another resource, the preceding form of the ASSIGN statement is also used. The resultant action is that the validity flag is first set false, implicitly "closing" the I/O path name to the device. A "new assignment" is then made just as if the first assignment never existed. Making this new assignment places information describing the specified device into the variable and sets the validity flag true. An example is shown below.

```
100   ASSIGN @Printer TO 1     ! Initial assignment.
110   OUTPUT @Printer;"Data1"
120   !
130   ASSIGN @Printer TO 701   ! 2nd ASSIGN closes 1st
140   OUTPUT @Printer;"Data2" ! and makes a new assignment.
150   PAUSE
160   END
```

The result of running the program is that "Data1" is sent to the CRT, and "Data2" is sent to HP-IB device 701. Since the program was paused (which maintains the program context), the I/O path name @Printer can be used in an I/O statement or re-assigned to another resource **from the keyboard**.

## Closing I/O Path Names

A second use of the ASSIGN statement is to **explicitly close** the name assigned to an I/O path. When the name is closed, the validity flag is set false, labeling the information as invalid[1]. Attempting to use the closed name results in error 177 (`Undefined I/O path name`). Examples of statements that close path names are as follows.

### Examples

```
ASSIGN @Printer TO *

ASSIGN @Serial_card TO *

ASSIGN @Gpio TO *
```

After executing this statement for a particular I/O path name, the name cannot be used in subsequent I/O statements until it is re-assigned. This same name can be assigned either to the same or to a different resource with a subsequent ASSIGN statement. However, if it is used prior to being re-assigned, error 177 occurs.

[1] Additional action may also be taken when the I/O path name assigned to a mass storage file is closed.

# I/O Path Names in Subprograms

When a subprogram (either a SUB subprogram or a user-defined function) is called, the "context" is changed to that of the called subprogram[1]. The statements in the subprogram only have access to the data of the new context. Thus, in order to use an I/O path name in any statement within a subprogram, **one** of the following conditions must be true.

- The I/O path name must already be assigned within the context (i.e., the same instance of the subprogram).
- The I/O path name must be assigned in another context and passed to this context by reference (i.e., specified in both the formal-parameter and pass-parameter lists).
- The I/O path name must be declared in a variable common (with COM statements) and already be assigned within a context that has access to that common block.

The following paragraphs and examples further describe using I/O path names in subprograms.

## Assigning I/O Path Names Locally Within Subprograms

Any I/O path name can be used in a subprogram if it has first been assigned to an I/O path within the same context of the subprogram. A typical example is shown below.

```
10   CALL Subprogram_x
20   END
30   !
40   SUB Subprogram_x
50   ASSIGN @Log_device TO 1 ! CRT.
60   OUTPUT @Log_device;"Subprogram"
70   SUBEND
```

[1] Subprograms and user-defined functions are fully discussed in the "Subprograms" chapter of *BASIC Programming Techniques*.

When the subprogram is exited, all I/O path names assigned locally within the subprogram are automatically closed. If the program (or subprogram) that called the exited subprogram attempts to use the I/O path name, an error results. An example of this closing local I/O path names upon return from a subprogram is shown below.

```
10 CALL Subprogram_x
11 OUTPUT @Log_device;"Main" <---------- Insert into previous example.
20 END
30 !
40 SUB Subprogram_x
50 ASSIGN @Log_device TO 1 ! CRT.
60 OUTPUT @Log_device;"Subprogram"
70 SUBEND
```

When the above program is run, error 177, `Undefined I/O path name`, occurs in line 11.

Each context has its own set of local variables, which are not automatically accessible to any other context. Consequently, if the same I/O path name is assigned to I/O paths in separate contexts, the assignment local to the context is used while in that context. Upon return to the calling context, any I/O path names accessible to this context remain assigned as before the context was changed.

```
1  ASSIGN @Log_device TO 701 <----------+ Insert into previous example.
2  OUTPUT @Log_device;"First Main" <----+
10 CALL Subprogram_x
11 OUTPUT @Log_device;"Second Main" <---- Change this line.
20 END
30 !
40 SUB Subprogram_x
50 ASSIGN @Log_device TO 1    ! CRT.
60 OUTPUT @Log_device;"Subprogram"
70 SUBEND
```

The results of the above program are that the outputs "First Main" and "Second Main" are directed to device 701, while the output "Subprogram" is directed to the CRT. Notice that the original assignment of @Log_device to device selector 701 is "restored" when the subprogram's context is exited, since the assignment of @Log_device made to interface select code 1 was local to the subprogram.

## Passing I/O Path Names as Parameters

I/O path names can be used in subprograms if they are assigned and have been passed to the called subprogram by reference; they cannot be passed by value. The I/O path name(s) to be used must appear in both the pass-parameter and formal-parameter lists.

```
1  ASSIGN @Log_device TO 701
2  OUTPUT @Log_device;"First Main"
10 CALL Subprogram_x(@Log_device)  <-- Add pass parameter.
11 OUTPUT @Log_device;"Second Main"
20 END
30 !
40 SUB Subprogram_x(@Log)  <-- Add formal parameter.
50 ASSIGN @Log TO 1 ! CRT.
60 OUTPUT @Log;"Subprogram"
70 SUBEND
```

Upon returning to the calling routine, any changes made to the assignment of the I/O path name passed by reference are maintained; the assignment local to the calling context is **not** restored as in the preceding example, since the I/O path name is **accessible to both contexts**. In this example, @Log_device remains assigned to interface select code 1; thus, "Subprogram" and "Second Main" are both directed to the CRT.

## Declaring I/O Path Names in Common

An I/O path name can also be accessed by a subprogram if it has been declared in a COM statement (labeled or unlabeled) common to calling and called contexts, as shown in the following example.

```
1  COM @Log_device  <-- Insert COM statement.
3  ASSIGN @Log_device TO 701
4  OUTPUT @Log_device;"First Main"
10 CALL Subprogram_x  <-- Parameters not necessary.
11 OUTPUT @Log_device;"Second Main"
20 END
30 !
40 SUB Subprogram_x  <-- Parameters not necessary.
41 COM @Log_device  <-- Insert COM statement.
50 ASSIGN @Log_device TO 1 ! CRT.
60 OUTPUT @Log_device;"Subprogram"
70 SUBEND
```

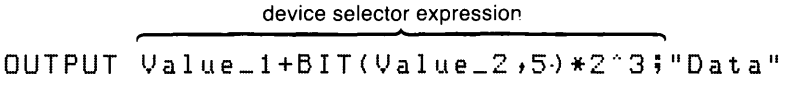
If an I/O path name in common is modified in any way, the assignment is changed for all subsequent contexts; the original assignment is not "restored" upon exiting the subprogram. In this example, "First Main" is sent to HP-IB device 701, but "Subprogram" and "Second Main" are both directed to the CRT. This is identical to the preceding action when the I/O path name was passed by reference.

# Benefits of Using I/O Path Names

Devices can be accessed with both device selectors and I/O path names, as shown in the previous discussions. With the information presented thus far, you may not see much difference between using these two methods of accessing devices. This section describes these differences in order to help you decide which method may be better for your application.

## Execution Speed

When a device selector is used in an I/O statement to specify the I/O path to a device, the numeric expression must be evaluated by the computer every time the statement is executed. If the expression is complex, this evaluation might take several milliseconds.




If a numeric variable is used to specify the device selector, this expression-evaluation time is reduced; this is the fastest execution possible when using device selectors. However, more information about the I/O process must be determined before it can be executed.

In addition to evaluating the numeric expression, the computer must determine which type of interface (HP-IB, GPIO, etc.) is present at the specified select code. Once the type of interface has been determined, the corresponding attributes of the I/O path must then be determined before the computer can use the I/O path. Only after all of this information is known can the process of actually copying the data be executed.

If an I/O path name is specified in an OUTPUT or ENTER statement, all of this information has already been determined at the time the name was assigned to the I/O path. Thus, an I/O statement containing an I/O path name executes slightly faster than using the corresponding I/O statement containing a device selector (for the same set of source-list expressions).

## Re-Directing Data

Using numeric-variable device selectors, as with I/O path names, allows a single statement to be used to move data between the computer and several devices. Simple examples of re-directing data in this manner are shown in the following programs.

**Example of Re-Directing with Device Selectors**

```
100   Device=1
110   GOSUB Data_out
        .
        .
        .
200   Device=9
210   GOSUB Data_out
        .
        .
        .
410   Data_out: OUTPUT Device;Data$
420             RETURN
```

**Example of Re-Directing with I/O Path Names**

```
100   ASSIGN @Device TO 1
110   GOSUB Data_out
        .
        .
        .
200   ASSIGN @Device TO 9
210   GOSUB Data_out
        .
        .
        .
410   Data_out: OUTPUT @Device;Data$
420             RETURN
```

The preceding two methods of re-directing data execute in approximately the same amount of time. As a comparison of the two methods, executing the "Device=" statement takes less time than executing the "ASSIGN @Device" statement. Conversely, executing the "OUTPUT Device" statement takes more time than executing the "OUTPUT @Device". However, the overall time for each method is approximately equal.

There are two additional factors to be considered. First, device selectors cannot be used to direct data to mass storage files; I/O path names are the only access to files. If the data is ever to be directed to a file, you should use I/O path names. A good example of re-directing data to mass storage files is given in the "I/O Path Attributes" chapter. The second additional factor is described below.

## Attribute Control

I/O paths have certain "attributes" which control how the system handles data sent through the I/O path. For example, the FORMAT attribute possessed by an I/O path determines which data representation will be used by the path during communications. If the path possesses the attribute of FORMAT ON, the ASCII data representation will be used. This is the default attribute automatically assigned by the computer when I/O path names are assigned to device selectors. If the I/O path possesses the attribute of FORMAT OFF, the internal data representation is used; this is the default format for BDAT files. Further details of these and additional attributes are discussed in the "I/O Path Attributes" chapter.

The second additional factor that favors using I/O path names is that you can control which attribute(s) are to be assigned to the I/O path to devices (and also to the I/O paths to files and buffers). If device selectors are used, this control is not possible. The "I/O Path Attributes" chapter describes how to specify the attributes to be assigned to an I/O path and gives several useful techniques for using the available attributes.

# Table of Contents

# Outputting Data 4

## Introduction

The preceding chapter described how to identify a specific device as the destination of data in an OUTPUT statement. Even though a few example statements were shown, the details of how the data are sent were not discussed. This chapter describes the topic of outputting data to devices; outputting data to string variables, buffers, and mass storage files is described in the "I/O Path Attributes" and "Advanced Transfer Techniques" chapters of this manual, in the "Data Storage and Retrieval" chapter of *BASIC Programming Techniques*, and in the *BASIC Language Reference.*

There are two general types of output operations. The first type, known as "free-field outputs", use the computer's default data representations[1]. The second type provides precise control over each character sent to a device by allowing you to specify the exact "image" of the ASCII data to be output.

[1] The ASCII representation described briefly in the preceding chapter is the default data representation used when communicating with with devices; however, the internal representation can also be used. See the "I/O Path Attributes" chapter for further details.

# Free-Field Outputs

Free-field outputs are invoked when the following types of OUTPUT statements are executed.

**Examples**

```
OUTPUT @Device;3.14*Radius^2

OUTPUT Printer;"String data";Num_1

OUTPUT 9;Test,Score,Student$

OUTPUT Escape_code$;CHR$(27)&"&A1S";
```

## The Free-Field Convention

The term "free-field" refers to the number of characters used to represent a data item. During free-field outputs, BASIC does *not* send a *constant* number of ASCII characters for each type of data item, as is done during "fixed-field outputs" which use images (described later in this chapter). Instead, a special set of rules is used that govern the number and type of characters sent for each source item. The rules used for determining the characters output for numeric and string data are described in the following paragraphs.

### Standard Numeric Format

The default data representation for devices is to use ASCII characters to represent numbers. The ASCII representation of each expression in the source list is generated during free-field output operations. Even though all REAL numbers have 15 (and INTEGERs can have up to 5) significant decimal digits of accuracy, not all of these digits are output with free-field OUTPUT statements. Instead, the following rules of the free-field convention are used when generating a number's ASCII representation.

All numbers between 1E−5 and 1E+6 are rounded to 12 significant digits and output in floating-point notation with no leading zeros. If the number is positive, a leading space is output for the sign; if negative, a leading "−" is output.

**Examples**

```
 32767
-32768
 123456.789012
-.000123456789012
```

If the number is less than 1E−5 or greater than 1E+6, it is rounded to 12 significant digits and output in scientific notation. No leading zeros are output, and the sign character is a space for positive and "−" for negative numbers.

**Examples**

```
-1.23456789012E+6
 1.23456789012E-5
```

### Standard String Format

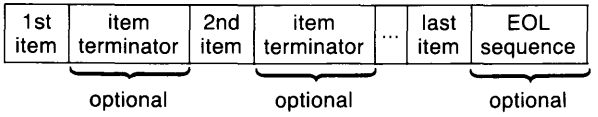
No leading or trailing spaces are output with the string's characters[1].

```
String characters.
No leading or trailing spaces.
```

## Item Separators and Terminators

Data items are output one byte (or word) at a time, beginning with the left-most item in the source list and continuing until all of the source items have been output. Items **in the list** must be **separated** by either a comma or a semicolon. However, items in the data output may or may not be separated by item terminators, depending on the use of item separators in the source lists.

The general sequence of items in the data output is as follows. The end-of-line (EOL) sequence is discussed in the next section.




[1] This statement describes the FORMAT ON attribute (ASCII data representation). When sending data with the FORMAT OFF attribute, however, the internal representation of string data is used; for strings, the data consists of a four-byte length header that contains the number of characters in the string, followed by the string characters. With FORMAT ON, there is *no* length header; only the ASCII string characters are sent.

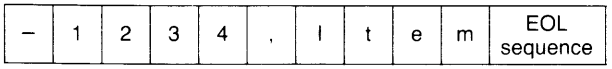
Using a **comma separator** after an item specifies that the **item terminator** (corresponding to the type of item) will be output after the last character of this item. A carriage-return, CHR$(13), and a line-feed, CHR$(10), terminate string items.

```
OUTPUT Device;"Item",-1234
```


The default EOL sequence is a CR/LF.

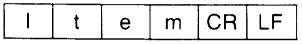
A comma separator specifies that a comma, CHR$(44), terminates numeric items.

```
OUTPUT Device;-1234,"Item"
```

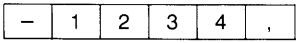



If a separator follows the last item in the list, the proper item terminator will be output **instead** of the EOL sequence.

```
OUTPUT Device;"Item",
```

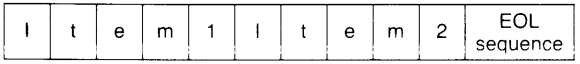



```
OUTPUT Device;-1234,
```

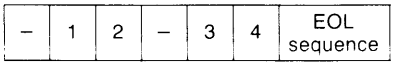



Using a **semicolon separator** suppresses output of the (otherwise automatic) item's terminator.

```
OUTPUT 1;"Item1";"Item2"
```

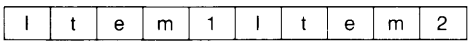



```
OUTPUT 1;-12;-34
```




If a semicolon separator follows the last item in the list, the EOL sequence and item terminators are suppressed.

```
OUTPUT 1;"Item1";"Item2";
```




Neither of the item terminators nor the EOL sequence are output.

If the item is an array, the separator following the array name determines what is output after each array element. (Individual elements are output in row-major order.)

```
100     OPTION BASE 1
110     DIM Array(2,3)
120     FOR Row=1 TO 2
130       FOR Column=1 TO 3
140         Array(Row,Column)=Row*10+Column
150       NEXT Column
160     NEXT Row
170     !
180     OUTPUT CRT;Array(*)   ! No trailing separator.
190     !
200     OUTPUT CRT;Array(*), ! Trailing comma.
210     !
220     OUTPUT CRT;Array(*)   ! Trailing semi-colon.
230     !
240     OUTPUT CRT;"Done"
250     END
```

### Resultant Output

| | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | , | | 1 | 2 | , | | 1 | 3 | , | | 2 | 1 | , | | 2 | 2 | , | | 2 | 3 | EOL sequence |
| | 1 | 1 | , | | 1 | 2 | , | | 1 | 3 | , | | 2 | 1 | , | | 2 | 2 | , | | 2 | 3 | , |
| | 1 | 1 | | 1 | 2 | | 1 | 3 | | 2 | 1 | | 2 | 2 | | 2 | 3 | | | | | | |
| D | O | N | E | EOL sequence | | | | | | | | | | | | | | | | | | | |

Item separators cause similar action for string arrays.

```
100    OPTION BASE 1
110    DIM Array$(2,3)[2]
120    FOR Row=1 TO 2
130      FOR Column=1 TO 3
140        Array$(Row,Column)=VAL$(Row*10+Column)
150      NEXT Column
160    NEXT Row
170    !
180    OUTPUT CRT;Array$(*)   ! No trailing separator.
190    !
200    OUTPUT CRT;Array$(*), ! Trailing comma.
210    !
220    OUTPUT CRT;Array$(*)   ! Trailing semi-colon.
230    !
240    OUTPUT CRT;"Done"
250    END
```

**Resultant Output**

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | CR | LF | 1 | 2 | CR | LF | 1 | 3 | CR | LF | 2 | 1 | CR | LF | 2 | 2 | CR | LF | 2 | 3 | EOL sequence |
| 1 | 1 | CR | LF | 1 | 2 | CR | LF | 1 | 3 | CR | LF | 2 | 1 | CR | LF | 2 | 2 | CR | LF | 2 | 3 | EOL sequence |
| 1 | 1 | 1 | 2 | 1 | 3 | 2 | 1 | 2 | 2 | 2 | 3 | | | | | | | | | | | |
| D | O | N | E | EOL sequence | | | | | | | | | | | | | | | | | | |

A pad byte may be sent following the last character of the EOL sequence when using an I/O path that possesses the WORD attribute. See the "I/O Path Attributes" chapter for further information.

## Changing the EOL Sequence (Requires IO)

An end-of-line (EOL) sequence is normally sent following the last item sent with OUTPUT. The default EOL sequence consists of a carriage-return and line-feed (CR/LF), sent with no device-dependent END indication. When the IO binary is loaded, it is also possible to define your own special EOL sequences that include sending special characters, sending an END indication, and delaying a specified amount of time after sending the EOL sequence.

In order to define non-default EOL sequences to be sent by the OUTPUT statement, an I/O path must be used. The EOL sequence is specified in one of the ASSIGN statements which describe the I/O path. An example is as follows.

```
ASSIGN @Device TO 12;EOL CHR$(10)&CHR$(10)&CHR$(13)
```

The characters following `EOL` are the new EOL-sequence characters. Any character in the range CHR$(0) through CHR$(255) may be included in the string expression that defines the EOL characters; however, the length of the sequence is limited to eight characters or less. The characters are put into the output data before any conversion is performed (if CONVERT OUT is in effect).

If END is included in the EOL attribute, an interface-dependent "END" indication is sent with (or after) the last character of the EOL sequence. However, if no EOL sequence is sent, the END indication is also suppressed. The following statement shows an example of defining the EOL sequence to include an END indication.

```
ASSIGN @Device TO 20;EOL CHR$(13)&CHR$(10) END
```

With the HP-IB Interface, the END indication is an End-or-Identify message (EOI) sent with the last EOL character. The individual chapter that describes programming each interface further describes each interface's END indication (if implemented).

If DELAY is included, the system delays the specified number of seconds (after sending the last EOL character and/or END indication) before executing any subsequent BASIC statement.

```
ASSIGN @Device;EOL CHR$(13)&CHR$(10) DELAY 0.1
```

This parameter is useful when using slower devices which the computer can "overrun" if data are sent as rapidly as the computer can send them. For example, a printer connected to the computer through a serial interface set to operate at 300 baud might require a delay after receiving a CR character to allow the carriage to return before sending further characters.

The default EOL sequence is a CR and LF sent with no END indication and no delay; this default can be restored by assigning EOL OFF to the I/O path.

EOL sequences can also be sent by using the "L" image specifier. See "Outputs that Use Images" for further details.

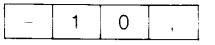
# Using END in Freefield OUTPUT

The secondary keyword END may be optionally specified following the last source-item expression in a freefield OUTPUT statement. The result is to **suppress the End-of-Line (EOL) sequence** that would otherwise be output after the last byte of the last source item. If a comma is used to separate the last item from the END keyword, the corresponding item terminator will be output as before (carriage-return and line-feed for string items and comma for numeric items).

**Examples**

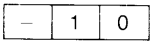
```
ASSIGN @Gpio TO 12

OUTPUT @Gpio;-10,END
```



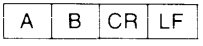

Item terminator, but no EOL sequence, is sent.

```
OUTPUT @Gpio;-10;END
OUTPUT @Gpio;-10 END
```



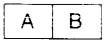

Neither item terminator nor EOL sequence is sent.

```
OUTPUT @Gpio;"AB",END
```



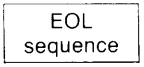

Item terminator, but no EOL sequence, is sent.

```
OUTPUT @Gpio;"AB";END
OUTPUT @Gpio;"AB" END
```




Neither item terminator nor EOL sequence is sent.

```
OUTPUT @Gpio
```




The EOL sequence is sent.

```
OUTPUT @Gpio;END     No EOL sequence is sent.
OUTPUT @Gpio;"" END
```

The END keyword has additional significance when the destination is a mass storage file. See the "Data Storage and Retrieval" chapter of *BASIC Programming Techniques* for further details.

## Additional Definition

BASIC defines additional action when END is specified in a freefield OUTPUT statement directed to either HP-IB or Data Communications interfaces.

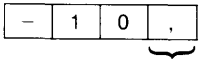
### END with HP-IB Interfaces

With HP-IB interfaces, END has the additional function of sending the End-or-Identify signal (EOI) with the last data byte of the last source item; however, **if no data are sent from the last source item, EOI is not sent**. For further description of the EOI signal, see the "HP-IB Interface" chapter.

#### Examples

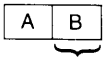
```
ASSIGN @Device TO 701


OUTPUT @Device;-10,END
```




EOI sent with the last character (numeric item terminator).

```
OUTPUT @Device;"AB";END
OUTPUT @Device;"AB" END
```




EOI sent with the last character of the item.

```
OUTPUT @Device;END
OUTPUT @Device;"" END
```

Neither EOL sequence nor EOI is sent, since no data is sent.

### END with the Data Communications Interface

With Data Communication interfaces, END has the additional function of sending an end-of-data indication to the interface. See the "Datacomm Interface" chapter for further details.

# Outputs that Use Images

The free-field form of the OUTPUT statement is very convenient to use. However, there may be times when the data output by the free-field convention is not compatible with the data required by the receiving device.

Several instances for which you might need to format outputs are: special control characters are to be output; the EOL sequence (carriage-return and line-feed) needs to be suppressed; or the exponent of a number must have only one digit. This section shows you how to use image specifiers to create your own, unique data representations for output operations.

## The OUTPUT USING Statement

When this form of the OUTPUT statement is used, the data is output according to the format image referenced by the "USING" secondary keyword. This image consists of one or more individual image specifiers which describe the type and number of data bytes (or words) to be output. The image can be either a string literal, a string variable, or the line label or number of an IMAGE statement. Examples of these four possibilities are listed below.

```
100  OUTPUT 1 USING "6A,SDDD.DDD,3X";" K= ",123.45

100  Image_str$="6A,SDDD.DDD,3X"
110  OUTPUT CRT USING Image_str$;" K= ";123.45

100  OUTPUT CRT USING Image_stmt;" K= ";123.45
110  Image_stmt: IMAGE 6A,SDDD.DDD,3X

100  OUTPUT 1 USING 110;" K= ";123.45
110  IMAGE 6A,SDDD.DDD,3X
```

## Images

Images are used to specify the format of data during I/O operations. Each image consists of groups of individual image (or "field") specifiers, such as 6A, SDDD.DDD, and 3X in the preceding examples. Each of these field specifiers describe one of the following things:

- It describes the desired format of one item in the source list. (For instance, 6A specifies that a string item is to be output in a "6-character Alpha" field. SDDD.DDD specifies that a numeric item is to be output with Sign, 3 Decimal digits preceding the decimal point, followed by 3 Decimal digits following the decimal point.)
- It specifies that special character(s) are to be output. (For instance, 3X specifies that 3 spaces are to be output.) There is no corresponding item in the source list.

Thus, you can think of the image list as either a precise format description or as a procedure. It is convenient to talk about the image list as a procedure for the purpose of explaining how this type of OUTPUT statement is executed.

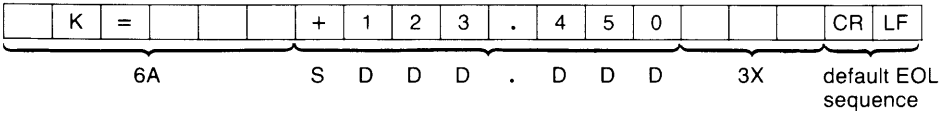
Again, each image list consists of images that each describe the format of data item to be output. The order of images in the list corresponds to the order of data items in the source list. In addition, image specifiers can be added to output (or to suppress the output of) certain characters. The following example steps through exactly how BASIC executes all of the preceding equivalent statements.

## Example of Using an Image

We will use the first of the four, equivalent output statements shown above. Don't worry if you don't understand each of the image specifiers used in the image list; each will be fully described in subsequent sections of this chapter. The main emphasis of this example is that you will see how an image list is used to govern the type and number of characters output.

```
OUTPUT CRT USING "6A,SDDD.DDD,3X";" K= ",123.45
```

The data stream output by the computer is as follows.




Step 1. The computer evaluates the first image in the list. Generally, each group of specifiers separated by commas is an "image"; the commas tell the computer that the image is complete and that it can be "processed". In general, each group of specifiers is processed before going on to the next group. In this case, 6 alphanumeric characters taken from the first item in the source list are to be output.

Step 2. The computer then evaluates the first item in the source list and begins outputting it, one byte (or word) at a time. After the 4th character, the first expression has been "exhausted". In order to satisfy the corresponding specifier, two spaces (alphanumeric "fill" characters) are output.

Step 3. The computer evaluates the next image (note that this image consists of several different image specifiers). The "S" specifier requires that a sign character be output for the number, the "D" specifiers require digits of a number, and the "." specifies where the decimal point will be placed. Thus, the number of digits following the decimal point have been specified. All of these specifiers describe the format of the next item in the source list.

Step 4. The next data item in the source list is evaluated. The resultant number is output one digit at a time, according to its image specifiers. A trailing zero has been added to the number to satisfy the "DDD" specifiers following the decimal point.

Step 5. The next image in the list ("3X")is evaluated. This specifier does not "require" data, so the source list needs no corresponding expression. Three spaces are output by this image.

Step 6. Since the entire image list and source list have been "exhausted", the computer then outputs the current (or default, if none has been specified) "end-of-line" sequence of characters (here we assume that a carriage-return and line-feed are the current EOL sequence).

The execution of the statement is now complete. As you can see, the data specified in the source list must match those specified in the output image in type and in number of items.

## Image Definitions During Outputs

This section describes the definitions of each of the image specifiers when referenced by OUTPUT statements. The specifiers have been categorized by data type. It is suggested that you scan through the description of each specifier and then look over the examples. You are also highly encouraged to experiment with the use of these concepts.

## Numeric Images

These image specifiers are used to describe the format of numbers.

**Table 4-1. Sign, Digit, Radix and Exponent Specifiers**

| Image Specifier | Meaning |
|---|---|
| S | Specifies a "+" for positive and a "−" for negative numbers is to be output. |
| M | Specifies a leading space for positive and a "−" for negative numbers is to be output. |
| D | Specifies one ASCII digit ("0" through "9") is to to be output. Leading spaces and trailing zeros are used as fill characters. The sign character, if any, "floats" to the immediate left of the most-significant digit. If the number is negative and no S or M is used, one digit specifier will be used for the sign. |
| Z | Same as "D" except that leading zeros are output. This specifier cannot appear to the right of a radix specifier (decimal point or R). |
| * | Like D, except that asterisks are output as leading fill characters (instead of spaces). This specifier cannot appear to the right of a radix specifier (decimal point or R). |
| . | Specifies the position of a decimal point radix-indicator (American radix) within a number. There can be only one radix indicator per numeric image item. |
| R | Specifies the position of a comma radix indicator (European radix) within a number. There can be only one radix indicator per numeric image item. |
| E | Specifies that the number is to be output using scientific notation. The "E" must be preceded by at least one digit specifier (D, Z, or *). The default exponent is a four-character sequence consisting of an "E", the exponent sign, and two exponent digits, equivalent to an "ESZZ" image. Since the number is left-justified in the specified digit field, the image for a negative number must contain a sign specifier (see the next section). |
| ESZ | Same as "E" but only 1 exponent digit is output. |
| ESZZZ | Same as "E" but three exponent digits are output. |
| K, −K | Specifies that the number is to be output in a "compact" format, similar to the standard numeric format; however, neither leading spaces (that would otherwise replace a "+" sign) nor item terminators (commas) are output, as would be with the standard numeric format. |
| H, −H | Like K, except that the number is to be output using a comma radix (European radix). |

## Numeric Examples

```
OUTPUT @Device USING "DDDD";-123.769
```

| – | 1 | 2 | 4 | EOL sequence |
|---|---|---|---|---|

```
OUTPUT @Device USING "4D";-1.2
```

| – | 1 | EOL sequence |
|---|---|---|

```
OUTPUT @Device USING "ZZ.DD";1.675
```

| 0 | 1 | . | 6 | 8 | EOL sequence |
|---|---|---|---|---|---|

```
OUTPUT @Device USING "Z.D";.35
```

| 0 | . | 4 | EOL sequence |
|---|---|---|---|

```
OUTPUT @Device USING "DD.E";12345
```

| 1 | 2 | . | E | + | 0 | 3 | EOL sequence |
|---|---|---|---|---|---|---|---|

```
OUTPUT @Device USING "2D.DDE";2E-4
```

| 2 | 0 | . | 0 | 0 | E | – | 0 | 5 | EOL sequence |
|---|---|---|---|---|---|---|---|---|---|

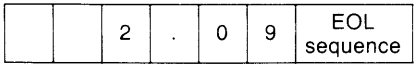
```
OUTPUT @Device USING "K";12.400
```

| 1 | 2 | . | 4 | EOL sequence |
|---|---|---|---|---|

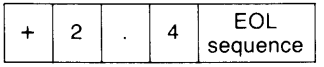
```
OUTPUT CRT USING "MDD.2D";-12.449
```

| – | 1 | 2 | . | 4 | 5 | EOL sequence |
|---|---|---|---|---|---|---|

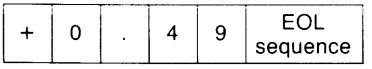
```
OUTPUT CRT USING "MDD.DD";2.09
```




```
OUTPUT 1 USING "SD.D";2.449
```

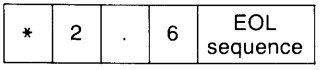



```
OUTPUT 1 USING "SZ.DD";.49
```

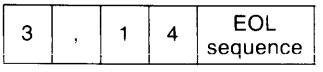



```
OUTPUT CRT USING "SDD.DDE";-2.35
```


```
OUTPUT @Device USING "**.D";2.6
```




```
OUTPUT @Device USING "DRDD";3.1416
```




```
OUTPUT @Device USING "H";3.1416
```

## String Images

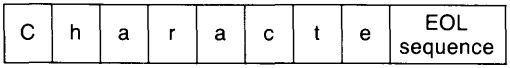
These types of image specifiers are used to specify the format of string data items.

**Table 4-2. Character Specifiers**

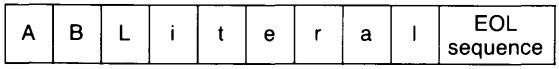
| Image Specifier | Meaning |
|---|---|
| A | Specifies that one character is to be output. Trailing spaces are used as fill characters if the string contains less than the number of characters specified. |
| "literal" | All characters placed in quotes form a string literal, which is output exactly as is. Literals can be placed in output images which are part of OUTPUT statements by enclosing them in double quotes. |
| K, −K, H, −H | Specifies that the string is to be output in "compact" format, similar to the standard string format; however, no item terminators are output as with the standard string format. |

### String Examples

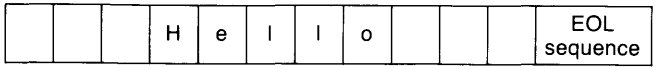
```
OUTPUT @Device USING "8A";"Characters"
```



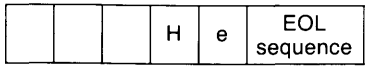

```
OUTPUT @Device USING "K,""Literal""";"AB"
```




```
OUTPUT @Device USING "K";"   Hello   "
```




```
OUTPUT @Device USING "5A";" Hello "
```



## Binary Images

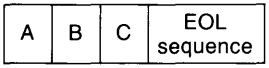
These image specifiers are used to output bytes (8-bit data) and words (16-bit data) to the destination. Typical uses are to output non-ASCII characters or integers in their internal representation.

**Table 4-3. Binary Specifiers**

| Image Specifier | Meaning |
|---|---|
| B | Specifies that one byte (8 bits) of data is to be output. The source expression is evaluated, rounded to an integer, and interpreted MOD 256. If it is less than −32 768, CHR$(0) is output. If is greater than 32 767, CHR$(255) is output. |
| W | Specifies that one word of data (16 bits) are to be sent as a 16-bit, two's-complement integer. The corresponding source expression is evaluated and rounded to an integer. If it is less than −32 768, then −32 768 is sent; if it is greater than 32 767, then 32 767 is sent.<br><br>If either an I/O path name with the BYTE attribute (see the "I/O Path Attributes" chapter) or a device selector is used to access an 8-bit interface, two bytes will be output; the first byte is most significant. If an I/O path name with the BYTE attribute is used to access a 16-bit interface, the BYTE attribute is overridden and one 16-bit word is output in a single handshake operation.<br><br>If an I/O path name with the WORD attribute is used to access a 16-bit interface, a pad byte, CHR$(0), is output whenever necessary to achieve alignment on a word boundary.<br><br>If the destination is a BDAT or HPUX file, string variable, or buffer, the WORD attribute is ignored and all data are sent as bytes; however, pad byte(s) will also be output whenever necessary to achieve alignment on a word boundary. The pad byte may be changed by using the CONVERT attribute (see the "I/O Path Attributes" chapter for details). |
| Y | Like W, except that no pad bytes are output to achieve alignment on a word boundary. If an I/O path with the BYTE attribute is used to access a 16-bit interface, the attribute is not overridden (as with the W specifier). |

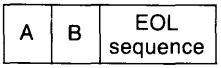
### Binary Examples

```
OUTPUT @Device USING "B,B,B";65,66,67
```



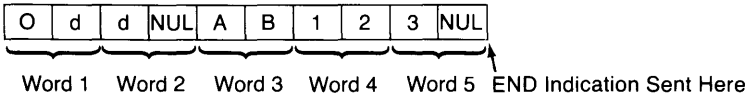

```
OUTPUT @Device USING "B";13
```


```
OUTPUT @Device USING "W";256*65+66
```



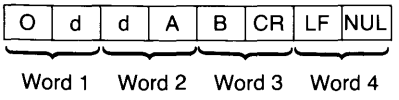

For this example, assume that @Device possesses the WORD attribute and that the EOL sequence consists of the characters "123" with an END indication.

```
OUTPUT @Device USING "K,W";"Odd",256*65+66
```




For this example, assume that @Device possesses the WORD attribute and that the EOL sequence is the default (CR/LF).

```
OUTPUT @Device USING "K,Y";"Odd",256*65+66
```



## Special-Character Images

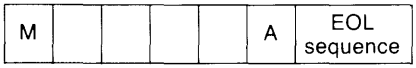
These specifiers require no corresponding data in the source list. They can be used to output spaces, end-of-line sequences, and form-feed characters.

**Table 4-4. Special-Character Specifiers**

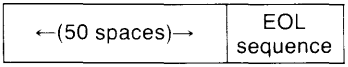
| Image Specifier | Meaning |
|---|---|
| X | Specifies that a space character, CHR$(32), is to be output. |
| / | Specifies that a carriage-return character, CHR$(13), and a line-feed character, CHR$(10), are to be output. |
| @ | Specifies that a form-feed character, CHR$(12), is to be output. |

### Special-Character Examples

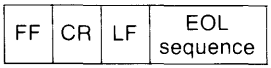
```
OUTPUT @Device USING "A,4X,A";"M","A"
```



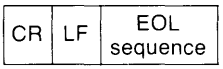

```
OUTPUT @Device USING "50X"
```




```
OUTPUT @Device USING "@,/"
```




```
OUTPUT @Device USING "/"
```



## Termination Images

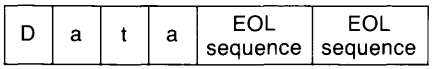
These specifiers are used to output or suppress the end-of-line sequence output after the last data item.

**Table 4-5. Termination Specifiers**

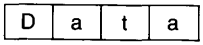
| Image Specifier | Meaning |
|---|---|
| L | Specifies that the current end-of-line sequence is to be output. The default EOL characters are CR and LF; see "Changing the EOL Sequence" for details on how to re-define these characters. If the destination is an I/O path name with the WORD attribute, a pad byte will be output after each EOL sequence when necessary to achieve word alignment. |
| # | Specifies that the EOL sequence that normally follows the last item is to be suppressed. |
| % | Is ignored in output images but is allowed to be compatible with ENTER images. |
| + | Specifies that the EOL sequence that normally follows the last item is to be replaced by a single carriage-return character (CR). |
| – | Specifies that the EOL sequence that normally follows the last item is to be replaced by a single line-feed character (LF). |

### Termination Examples

```
OUTPUT @Device USING "4A,L";"Data"
```

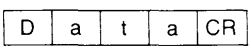



```
OUTPUT @Device USING "#,K";"Data"
```

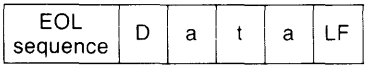



```
OUTPUT @Device USING "#,B";12
```

```
OUTPUT @Device USING "+,K";"Data"
```




```
OUTPUT @Device USING "-,L,K";"Data"
```




# Additional Image Features

Several additional features of outputs which use images are available with the computer. Several of these features, which have already been shown, will be explained here in detail.

## Repeat Factors

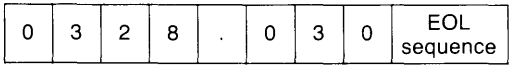
Many of the specifiers can be repeated without having to explicitly list the specifier as many times as it is to be repeated. For instance, to a character field of 15 characters, you do not need to use "AAAAAAAAAAAAAAA"; instead, you merely specify the number of times that the specifier is to be repeated in front of the image ("15A"). The following specifiers can be repeated by specifying an integer repeat factor; the specifiers not listed cannot be repeated in this manner.

### Repeatable Specifiers

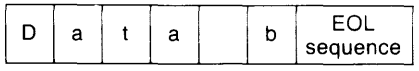
Z, D, A, X, /, @, L

### Examples

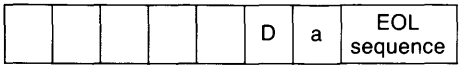
```
OUTPUT @Device USING "4Z.3D";328.03
```



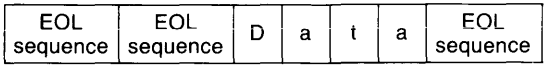

```
OUTPUT @Device USING "6A";"Data bytes"
```



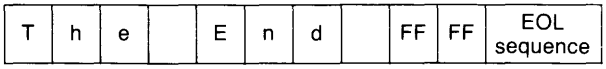
```
OUTPUT @Device USING "5X,2A";"Data"
```




```
OUTPUT @Device USING "2L,4A";"Data"
```




```
OUTPUT @Device USING "8A,2@";"The End"
```




```
OUTPUT @Device USING "2/"
```


## Image Re-Use

If the number of items in the source list exceeds the number of matching specifiers in the image list, the computer attempts to re-use the image(s) beginning with the first image.

```
110   ASSIGN @Device TO CRT
120   Num_1=1
130   Num_2=2
140   !
150   OUTPUT @Device USING "K";Num_1,"Data_1",Num_2,"Data_2"
160   OUTPUT @Device USING "K,/";Num_1,"Data_1",Num_2,"Data_2"
170   END
```

### Resultant Display

```
1Data_12Data_2
1
Data_1
2
Data_2
```

Since the "K" specifier can be used with both numeric and string data, the above OUTPUT statements can re-use the image list for all items in the source list. If any item cannot be output using the corresponding image item, an error results. In the following example, "Error 100 in 150" occurs due to data mismatch.

```
110   ASSIGN @Device TO CRT
120   Num_1=1
130   Num_2=2
140   !
150   OUTPUT @Device USING "DD.DD";Num_1,Num_2,"Data_1"
160   END
```

## Nested Images

Another convenient capability of images is that they can be nested within parentheses. The entire image list within the parentheses will be used the number of times specified by the repeat factor preceding the first parenthesis. The following program is an example of this feature.

```
100   ASSIGN @Device TO 701
110   !
120   OUTPUT @Device USING "3(B),X,DD,X,DD";65,66,67,68,69
130   END
```

**Resultant Output**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | B | C | | 6 | 8 | | 6 | 9 | EOL sequence |

This nesting with parentheses is made with the same hierarchy as with parenthetical nesting within mathematical expressions. Only eight levels of nesting are allowed.

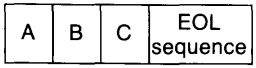
# END with OUTPUTs that Use Images

Using the optional secondary keyword END in an OUTPUT statement that uses an image produces results which differ from those of using END in a freefield OUTPUT statement. Instead of always suppressing the EOL sequence, the END keyword **only suppresses the EOL sequence when no data are output from the last source-list expression.** Thus, the "#" image specifier generally controls the suppression of the otherwise automatic EOL sequence, while the END keyword suppresses it only in less common usages.

**Examples**

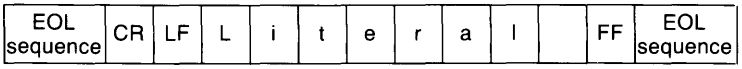
```
Device=12

OUTPUT Device USING "K";"ABC",END
OUTPUT Device USING "K";"ABC";END
OUTPUT Device USING "K";"ABC" END
```



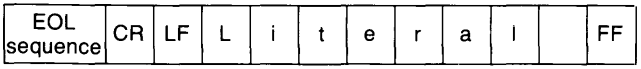

The EOL sequence is not suppressed.

```
OUTPUT Device USING "L,/,""Literal"",X,@"
```




In this case, specifiers that require no source-item expressions are used to generate characters for the output; there are no source expressions. The EOL sequence is output after all specifiers have been used to output their respective characters. Compare this action to that shown in the next example.

```
OUTPUT Device USING "L,/,""Literal"",X,@;END
```




The EOL sequence is suppressed because no source items were included in the statement; all characters output were the result of specifiers which require no corresponding expression in the source list.

## Additional END Definition

The END secondary keyword has been defined to produce additional action when included in an OUTPUT statement directed to HP-IB and Data Communications interfaces.

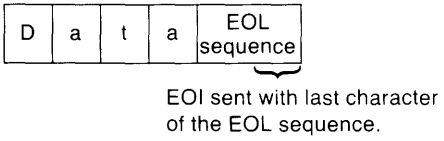
### END with HP-IB Interfaces

With HP-IB interfaces, END has the additional function of sending the End-or-Identify signal (EOI) with the **last character** of either the last source item or the EOL sequence (if sent). As with freefield OUTPUT, **no EOI is sent if no data is sent from the last source item and the EOL sequence is suppressed.**

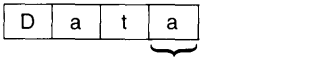
#### Examples

```
ASSIGN @Device TO 701

OUTPUT @Device USING "K";"Data",END
OUTPUT @Device USING "K";"Data","",END
```

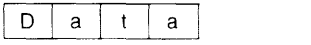



```
OUTPUT @Device USING "#,K";"Data" END
```




EOI is sent with the last character of the last source item when the EOL sequence is suppressed, because the last source item contained data which was used in the output.

```
OUTPUT @Device USING "#,K";"Data","",END
OUTPUT @Device USING """Data""";END
```




The EOI was not sent in either case, since no data were sent from the last source item **and** the EOL sequence was suppressed.

### END with Data Communications Interfaces

With Data Communications interfaces, END has the additional definition of sending an end-of-data indication to the interface in the same instances in which EOI would be sent on HP-IB interfaces. See the "Datacomm Interface" chapter for further details.

# Table of Contents

# Entering Data 5

This chapter discusses the topic of entering data from devices. You may already be familiar with the OUTPUT statement described in the previous chapter; many of those concepts are applicable to the process of entering data. Earlier in this manual, you were told that **the data output from the sender had to match that expected by the receiver.** Because of the many ways that data can be represented in external devices, entering data can sometimes require more programming skill than outputting data. In this chapter, you will see what is involved in being the receiving device. Both free-field enters and enters that use images are described, and several examples are given with each topic.

## Free-Field Enters

Executing the free-field form of the ENTER invokes conventions which are the "converse" of those used with the free-field OUTPUT statement. In other words, data output using the free-field form of the OUTPUT statement can be readily entered using the free-field ENTER statement; no explicit image specifiers are required. The following statements exemplify this form of the ENTER statement.

**Examples**

```
100 ENTER @Voltmeter;Reading

100 ENTER 724;Readings(*)

100 ENTER From_string$;Average,Student_name$

100 ENTER @From_file;Data_code,Str_element$(X,Y)
```

## Item Separators

Destination items in ENTER statements can be separated by **either** a comma or a semicolon. Unlike the OUTPUT statement, it makes *no difference* which is used: data will be entered into each destination item in a manner independent of the punctuation separating the variables in the list. However, *no trailing punctuation is allowed.* The first two of the following statements are equivalent, but an error is reported when the third statement is executed.

**Examples**

```
ENTER @From_a_device;N1,N2,N3
```
*These first two statements are equivalent.*

```
ENTER @From_a_device;N1;N2;N3
```

```
ENTER @From_a_device;N1,N2,N3,
```
*Executing this statement causes an error (because of trailing comma).*

## Item Terminators

Unless the receiver knows exactly how many characters are to be sent, each data item output by the sender must be terminated by special character(s). When entering ASCII data[1] with the free-field form of the ENTER statement, the computer does not know how many characters will be output by the sender.

Item terminators must signal the end of each item so that the computer enters data into the proper destination variable. The terminator of the last item may also terminate the ENTER statement (in some cases). The actual character(s) that terminate entry into each type of variable are described in the next sections.

In addition to the termination characters, each item can be terminated (only with selected interfaces) by a device-dependent END indication. For instance, some interfaces use a signal known as EOI (End-or-Identify). The EOI signal is only available with the HP-IB, CRT, and keyboard interfaces. EOI termination is further described in the next sections.

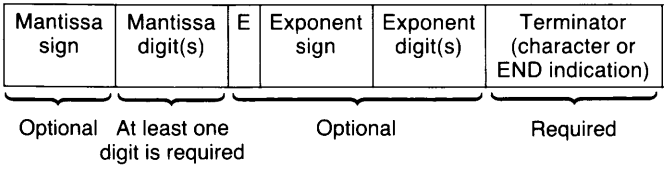
When using an I/O path that possesses the WORD attribute, an additional byte may be entered (but ignored). See the "I/O Path Attributes" chapter for further information.

[1] The ASCII data representation described briefly in Chapter 2 is the default data representation used with devices; however, the internal representation can also be used. See the "I/O Path Attributes" chapter for further details.

## Entering Numeric Data with the Number Builder

When the free-field form of the ENTER statement is used, numbers are entered by a routine known as the "number builder". This firmware routine evaluates the incoming ASCII numeric characters and then "builds" the appropriate internal-representation number. This number builder routine recognizes whether data being entered is to be placed into an INTEGER or REAL variable and then generates the appropriate internal representation.

The number builder is designed to be able to enter several formats of numeric data. However, the general format of numeric data must be as follows to be interpreted properly by the computer.



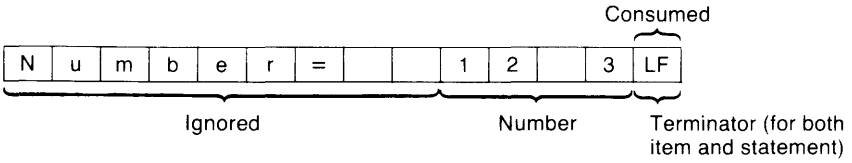

Numeric characters include decimal digits "0" through "9" and the characters ".", "+", "−", "E", and "e". These last five characters must occur in meaningful positions in the data stream to be considered numeric characters; if any of them occurs in a position in which it cannot be considered part of the number, it will be treated as a non-numeric character.

The following **rules** are used by the number builder to construct numbers from incoming streams of ASCII numeric characters.

1. All leading non-numerics are ignored; all leading and imbedded spaces are ignored.

   **Example**

```
100   ASSIGN @Device TO Device_selector
110   ENTER @Device;Number  ! Default is data type REAL.
120   END
```

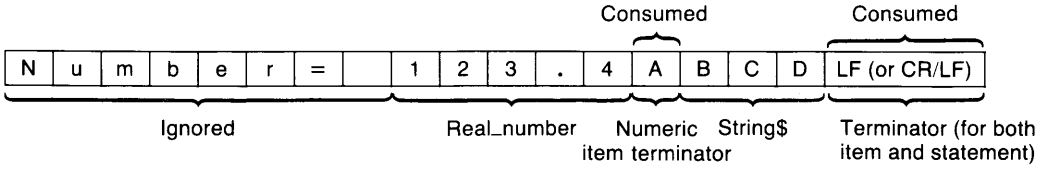



The result of entering the preceding data with the given ENTER statement is that Number receives a value of 123. The line-feed (statement terminator) is **required** since Number is the last item in the destination list.

2. Trailing non-numerics terminate entry into a numeric variable, and the terminating characters (of **both** string and numeric items) are "consumed". In this manual, "consumed" characters refers to characters **used to terminate** an item but not entered into the variable; "ignored" characters are entered but are **not used**.

**Example**

```
ENTER @Device;Real_number,String$
```

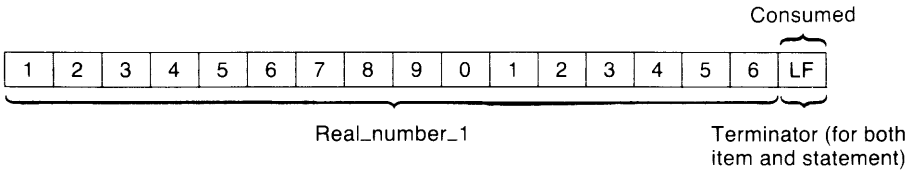



The result of entering the preceding data with the given ENTER statement is that Real_number receives the value 123.4 and String$ receives the characters "BCD". The "A" was lost when it terminated the numeric item; the string-item terminator(s) are also lost. The string-item terminator(s) also terminate the ENTER statement, since String$ is the last item in the destination list.

3. If more than 16 digits are received, only the first 16 are used as significant digits. However, all additional digits are treated as trailing zeros so that the exponent is built correctly.

**Example**

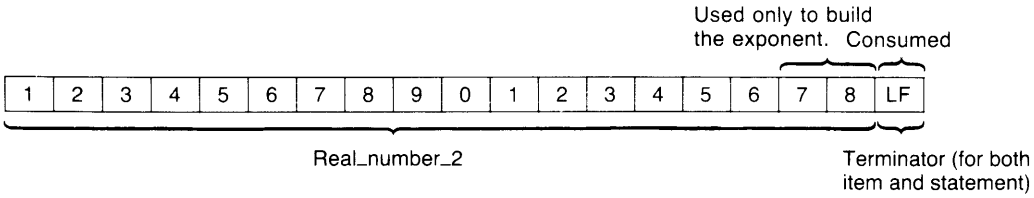
```
ENTER @Device;Real_number_1
```




The result of entering the preceding data with the given ENTER statement is that Real_number_1 receives the value 1.234567890123456 E+15.

**Example**

```
ENTER @Device;Real_number_2
```

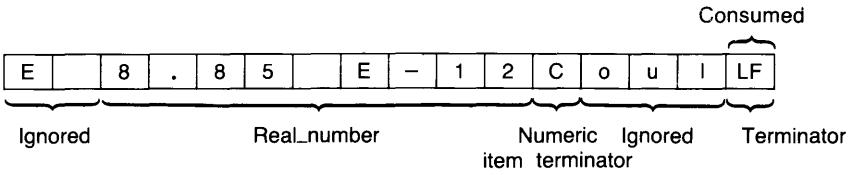



The result of entering the preceding data with the given ENTER statement is that Real_number_2 receives the value 1.234567890123456 E+**17**.

4. Any exponent sent by the source must be preceded by at least one mantissa digit **and** an "E" (or "e") character. If no exponent digits follow the "E" (or "e"), no exponent is recognized, but the number is built accordingly.

   **Example**

```
ENTER @Device;Real_number
```

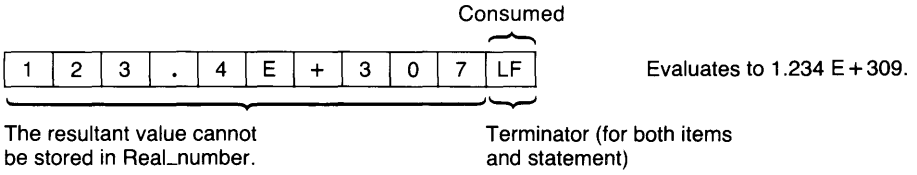



The result of entering the preceding data with the given ENTER statement is that Real_number receives a value of 8.85 E−12. The character "C" terminates entry into Real_number, and the characters "oul" are entered (but ignored) in search of the required line-feed statement terminator. If the character "C" is to be entered but not ignored, you must use an image. Using images with the ENTER statement is described later in this chapter.

5. If a number evaluates to a value outside the range corresponding to the type of the numeric variable, an error is reported. If no type has been declared explicitly for the numeric variable, it is assumed to be REAL.

   **Example**

```
ENTER @Device;Real_number
```




The data is entered but evaluates to a number outside the range of REAL numbers. Consequently, error 19 is reported, and the variable **Real_number retains its former value.**

6. If the item is the **last** one in the list, **both** the **item** and the **statement** need to be properly **terminated**. If the numeric **item** is terminated by a non-numeric character, the **statement** will **not** be terminated until it either receives a **line-feed** character or an END indication (such as EOI signal with a character). The topic of terminating free-field ENTER statements is described later in this chapter in the section of the same name.

## Entering String Data

Strings are groups of ASCII characters of varying lengths. Unlike numbers, almost any character can appear in any position within a string; there is not really any defined structure of string data. The routine used to enter string data is therefore much simpler than the number builder. It only needs to keep track of the dimensioned length of the string variable and look for string-item terminators (such as CR/LF, LF, or EOI sent with a character).

String-item terminator characters are either a line-feed (LF) or a carriage-return followed by a line-feed (CR/LF). As with numeric-item terminators characters, these characters are not entered into the string variable (during free-field enters); they are "lost" when they terminate the entry. The EOI signal also terminates entry into a string variable, but the variable must be the last item in the destination list (during free-field enters).

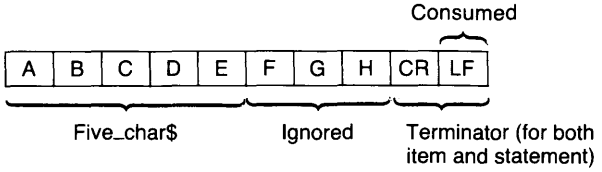
**All** characters received from the source are entered directly into the appropriate string variable until **any** of the following conditions occurs:

- an item terminator character is received.
- the number of characters entered equals the dimensioned length of the string variable.
- the EOI signal is received.

The following statements and resultant variable contents illustrate the first two conditions; the next section describes termination by EOI. Assume that the string variables Five_char$ and Ten_char$ are dimensioned to lengths of 5 and 10 characters, respectively.

**Example**

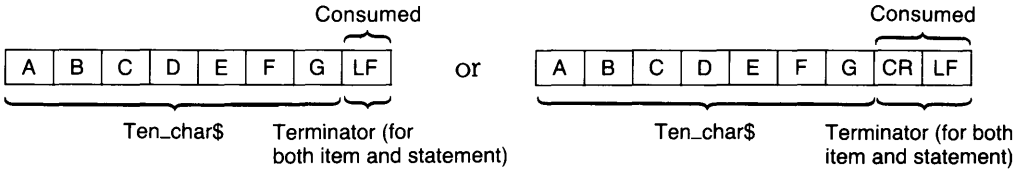
```
ENTER @Device;Five_char$
```




The variable Five_char$ only receives the characters "ABCDE", but the characters "FGH" are entered (and ignored) in search of the terminating carriage-return/line-feed (or line-feed).

**Example**

```
ENTER @Device;Ten_char$
```




The result of entering the preceding data with the given ENTER statement is that Ten_char$ receives the characters "ABCDEFG" and the terminating LF (or CR/LF) is lost.

## Terminating Free-Field ENTER Statements

Terminating conditions for free-field ENTER statements are as follows.

1. If the **last item** is terminated by a line-feed or by a character accompanied by EOI, the **entire statement** is properly terminated.
2. If an **END indication** is received while entering data into the **last item**, the statement is properly terminated. Examples of END indications are encountering the last character of a string variable while entering data from the variable, receiving EOI with a character, and receiving a control block while entering data through the Data Communications interface
3. If one of the preceding **statement-termination** conditions has **not** occurred **but** entry into the **last item** has been terminated,up to 256 **additional** characters are entered in search of a termination condition. If one is not found, an error occurs.

One case in which this termination condition may not be obvious can occur while entering string data. If the last variable in the destination list is a string **and** the dimensioned length of the string has been reached **before** a terminator is received, additional characters are entered (but ignored) until the terminator is found. The reason for this action is that the next characters received are still part of this data item, as far as the data **sender** is concerned. These characters are accepted from the sender so that the next enter operation will not receive these "leftover" characters.

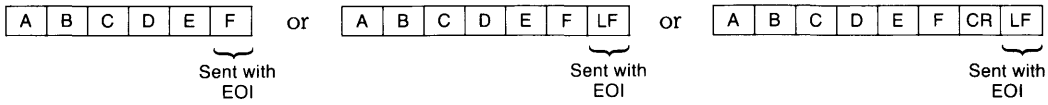
Another case involving numeric data can also occur (see the example given with "rule 4" describing the number builder). If a trailing non-numeric character terminates the last item (which is a numeric variable), additional characters will be entered in search of either a line-feed or a character accompanied by EOI. Unless this terminating condition is found before 256 characters have been entered, an error is reported.

## EOI Termination

A termination condition for the HP-IB Interface is the EOI (End-or-Identify) signal. When this message is sent, it immediately terminates the entire ENTER statement, regardless of whether or not all variables have been satisfied. However, if all variable items in the destination list have not been satisfied, an error is reported.

**Example**

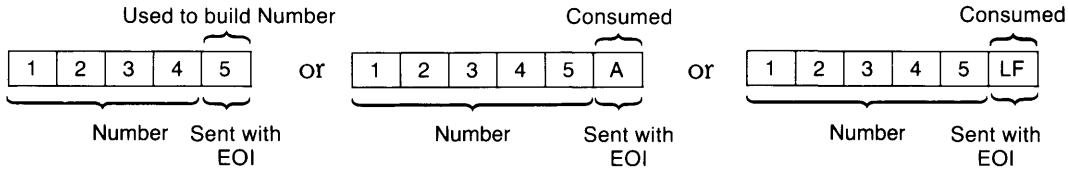
```
ENTER @Device;String$
```




The result of entering the preceding data with the given ENTER statement is that String$ receives the characters "ABCDEF". The EOI signal being received with either the last character or with the terminator character properly terminates the ENTER statement. If the character accompanied by EOI is a string character (not a terminator), it is entered into the variable as usual.

**Example**

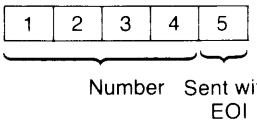
```
ENTER @Device;Number
```




The result of entering any of the above data streams with the given ENTER statement is that Number receives the value 12345. If the EOI signal accompanies a numeric character, it is entered and used to build the number; if the EOI is received with a numeric terminator, the terminator is lost as usual.

**Example**

```
ENTER @Device;Number,String$
```




An error is reported
(Error 153 Insufficient data for ENTER).

The result of entering the preceding data with the given statement is that an **error is reported** when the character "5" accompanied by EOI is received. However, Number receives the value 12345, but String$ retains its previous value. An error is reported because **all** variables in the destination list have **not** been satisfied when the EOI is received. Thus, the EOI signal is an **immediate statement terminator during free-field enters**. The EOI signal has a **different** definition during enters that use images, as described later in this chapter.

The EOI signal is implemented on the HP-IB Interface, described in the "HP-IB Interface" chapter of this manual. Since it is often convenient to to use the keyboard and CRT for external devices, these internal devices have been designed to simulate this signal. Further descriptions of this feature's implementation in the CRT display and keyboard are contained in the "Display Interfaces" and "Keyboard Interfaces" chapters of this manual, respectively.

# Enters that Use Images

The free-field form of the ENTER statement is very convenient to use; the computer automatically takes care of placing each character into the proper destination item. However, there are times when you need to design your own images that match the format of the data output by sources. Several instances for which you may need to use this type of enter operations are: the incoming data does not contain any terminators; the data stream is not followed by an end-of-line sequence; or two consecutive bytes of data are to be entered and interpreted as a two's-complement integer.

## The ENTER USING Statement

The means by which you can specify how the computer will interpret the incoming data is to reference an image in the ENTER statement. The four general ways to reference the image in ENTER statements are as follows.

1. 
```
100  ENTER @Device_x USING "6A,DDD.DD";String_var$,Num_var
```

2. 
```
100  Image_str$="6A,DDD.DD"
110  ENTER @Device_x USING Image_str$;String_var$,Num_var
```

3. 
```
100  ENTER @Device USING Image_stmt;String_var$,Num_var
110  Image_stmt: IMAGE 6A,DDD.DD
```

4. 
```
100  ENTER @Device USING 110;String_var$,Num_var
110  IMAGE 6A,DDD.DD
```

# Images

Images are used to specify how data entered from the source is to be interpreted and placed into variables; each image consists of one or more groups of individual image specifiers that determine how the computer will interpret the incoming data bytes (or words). Thus, image lists can be thought of as a description of **either**:

- the format of the expected data, or
- the procedure that the ENTER statement will use to enter and interpret the incoming data bytes.

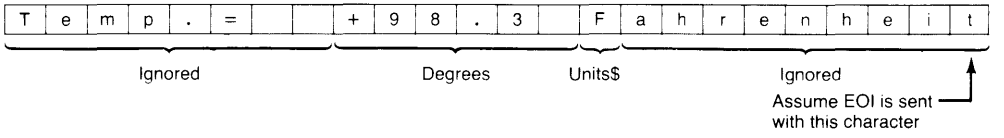
The examples given here treat the image list as a **procedure**.

All of the image specifiers used in image lists are valid for both enters and outputs. However, most of the specifiers have a slightly different meaning for each operation. If you plan to use the same image for output and enter, you must fully understand how both statements will use the image.

## Example of an Enter Using an Image

This example is used to show you exactly how the computer uses the image to enter incoming data into variables. Look through the example to get a general feel for how these enter operations work. Afterwards, you should read the descriptions of the pertinent specifier(s).

Assume that the following stream of data bytes are to be entered into the computer.



Given the preceding conditions, let's look at how the computer executes the following ENTER statement that uses the specified IMAGE statement.

```
300  ENTER @Device USING Image_1;Degrees,Units$
310  Image_1:  IMAGE 8X,SDDD.D,A
```

Step 1. The computer evaluates the first image of the IMAGE statement. It is a special image in that it does not correspond to a variable in the destination list. It specifies that eight characters of the incoming data stream are to be ignored. Eight characters, "Temp.=  ", are entered and are ignored (i.e., are not entered into any variable).

Step 2. The computer evaluates the next image. It specifies that the next six characters are to be used to build a number. Even though the order of the sign, digit, and radix are explicitly stated in the image, the actual order of these characters in the incoming data stream does not have to match this specifier exactly. Only the **number** of numeric specifiers in the image, here six, is all that is used to specify the data format. When all six characters have been entered, the number builder attempts to form a number.

Step 3. After the number is built, it is placed into the variable "Degrees"; the representation of the resultant number depends on the numeric variable's type (INTEGER, REAL, or COMPLEX[1]).

Step 4. The next image in the IMAGE statement is evaluated. It requires that one character be entered for the purpose of filling the variable "Units$". One byte is then entered into Units$.

Step 5. All images have been satisfied; however, the computer has not yet detected a statement-terminating condition. A line-feed or a character accompanied by EOI must be received to terminate the ENTER statement. Characters are then entered, but ignored, in search of one of these conditions. The statement is terminated when the EOI is sent with the "t". For further explanation, see "Terminating Enters that Use Images", near the end of this chapter.

The above example should help you to understand how images are used to determine the interpretation of incoming data. The next section will help you to use each specifier to create your desired images.

# Image Definitions During Enter

This section describes the individual image specifiers in detail. The specifiers have been categorized into data and function type.

## Numeric Images

Sign, digit, radix, and exponent specifiers are all used identically in ENTER images. The number builder can also be used to enter numeric data.

**Table 5-1. Numeric Specifiers**

| Image Specifier | Meaning |
|---|---|
| D | Specifies that one byte is to be entered and interpreted as a numeric character. If the characters is non-numeric (including leading spaces and item terminators), it will still "consume" one digit of the image item. |
| Z, * | Same action as D. Keep in mind that A and * can only appear to the left of the radix indicator (decimal point or R) in a numeric image item. |
| S, M | Same action as D in that one byte is to be entered and interpreted as a numeric character. At least one digit specifier must follow either of these specifiers in an image item. |
| . | Same action as D in that one byte is to be entered and interpreted as a numeric character. At least one digit specifier must accompany this specifier in an image item. |
| R | Same action as D in that one byte is to be entered and interpreted as a numeric character; however, when R is used in a numeric image, it directs the number builder to use the comma as a radix indicator and the period as a terminator to the numeric item. At least one digit specifier must accompany this specifier in the image item. |

**Table 5-1. Numeric Specifiers (Continued)**

| Image Specifier | Meaning |
|---|---|
| | |
| E | Equivalent to 4D, if preceded by at least one digit specifier (Z, *, or D) in the image item.<br>The following specifiers must also be preceded by at least one digit specifier. |
| ESZ | Equivalent to 3D. |
| ESZZ | Equivalent to 4D. |
| ESZZZ | Equivalent to 5D. |
| K, −K | Specifies that a variable number of characters are to be entered and interpreted according to the rules of the number builder (same rules as used in "free-field" ENTER operations). |
| H, −H | Like K, except that a comma is used as the radix indicator, and a period is used as the terminator for the numeric item. |

### Examples of Numeric Images

```
ENTER @Device USING "SDD.D";Number
ENTER @Device USING "3D.D";Number
ENTER @Device USING "5D";Number          These 5 are equivalent.
ENTER @Device USING "DESZZ";Number
ENTER @Device USING "**.DD";Number

ENTER Device USING "K";Number            Use the rules of the number builder.

ENTER @Device USING "DDRDD";Number       Enter five characters,
                                          using comma as radix.

ENTER @Device USING "H";Number           Use the rules of the number
                                         builder, but use the comma as radix
                                         and period as terminator.
```

## String Images

The following specifiers are used to determine the number of and the interpretation of data bytes entered into string variables.

**Table 5-2. String Specifiers**

| Image Specifier | Meaning |
|---|---|
| A | Specifies that one byte is to be entered and interpreted as a string character. Any terminators are entered into the string when this specifier is used. |
| K, H | Specifies that "free-field" ENTER conventions are to be used to enter data into a string variable; characters are entered directly into the variable until a terminating condition is sensed (such as CR/LF, LF, or an END indication). |
| −K, −H | Like K, except that line-feeds (LF's) do not terminate entry into the string; instead, they are treated as string characters and placed in the variable. Receiving an END indication terminates the image item (for instance, receiving EOI with a character on an HP-IB interface, encountering an end-of-data, or reaching the variable's dimensioned length). |
| L, @ | These specifiers are ignored for ENTER operations; however, they are allowed for compatibility with OUTPUT statements (that is, so that one image may be used for **both** ENTER and OUTPUT statements). Note that it may be necessary to skip characters (with specifiers such as X or /) when ENTERing data which has been sent by including these specifiers in an OUTPUT statement. Even greater care must be given to cases in which pad bytes may be sent; see "The BYTE and WORD Attributes" in the "I/O Path Attributes" chapter for further explanation. |

### Examples of String Images

```
ENTER @Device USING "10A";Ten_chars$          Enter 10 characters.

ENTER @Device USING "K";Any_string$           Enter using the free-field rules.

ENTER @Device USING "5A,K";String$,Number$    Enter two strings.

ENTER @Device USING "5A,K";String$,Number     Enter a string and a number.

ENTER @Device USING "-K";All_chars$           Enter characters until string
                                               is "full" or END is received.
```

## Ignoring Characters

These specifiers are used when one or more characters are to be ignored (i.e., entered but not placed into a string variable).

**Table 5-3. Specifiers Used to Ignore Characters**

| Image Specifier | Meaning |
|---|---|
| X | Specifies that a character is to be entered but ignored (not placed into a variable). |
| "literal" | Specifies that the number of characters in the literal are to be entered but ignored (not placed into a variable). |
| / | Specifies that all characters are to be entered but ignored (not placed into a variable) until a line-feed is received. EOI is also ignored until the line-feed is received. |

### Examples of Ignoring Characters

```
ENTER @Device USING "5X,5A";Five_chars$
```
*Ignore first five and use second five characters.*

```
ENTER @Device USING "5A,4X,10A";S_1$,S_2$
```
*Ignore 6th through 9th characters.*

```
ENTER @Device USING "/,K";String2$
```
*Ignore 1st item of unknown length.*

```
ENTER @Device USING """zz"",AA";S_2$
```
*Ignore two characters.*

## Binary Images

These specifiers are used to enter one byte (or word) that will be interpreted as a number.

**Table 5-4. Binary Specifiers**

| Image Specifier | Meaning |
|---|---|
| B | Specifies that one byte is to be entered and interpreted as an integer in the range 0 through 255. |
| W | Specifies that one 16-bit word is to be entered and interpreted as a 16-bit, two's complement INTEGER. If either an I/O path name with the BYTE attribute (see the "I/O Path Attributes" chapter) or a device selector is used to access an 8-bit interface, two bytes will be entered; the first byte entered is most significant. If an I/O path name with the BYTE attribute is used to access a 16-bit interface, the BYTE attribute is overwritten and one word is entered in a single operation. If an I/O path name with the WORD attribute is used to access a 16-bit interface, one byte is entered and ignored when necessary to achieve alignment on a word boundary. If the source is a file, string variable, or BUFFER, the WORD attribute is ignored and all data are entered as bytes; however, one byte may still be entered and ignored when necessary to achieve alignment on a word boundary. |
| Y | Like W, except that pad bytes are never entered to achieve word alignment. If an I/O path name with the BYTE attribute is used to access a 16-bit interface, the BYTE attribute is *not* overwritten (as with the W specifier). |

### Examples of Binary Images

```
ENTER @Device USING "B,B,B";N1,N2,N3
```
*Enter three bytes, then look for LF or END indication.*

```
ENTER @Device USING "W,K";N,N$
```
*Enter the first two bytes as an INTEGER, then the rest as string data.*

*Assume that @Device possesses the WORD attribute.*

```
ENTER @Device USING "B,W";Num_1,Num_2
```
*Enter one byte, ignore one (pad) byte, enter one word, then search for terminator.*

*@Device may possess either BYTE or WORD attribute.*

```
ENTER @Device USING "B,Y";Num_1,Num_2
```
*Enter one byte, enter one word, then search for terminator.*

# Terminating Enters that Use Images

This section describes the **default statement-termination conditions** for **enters that use images** (for devices). The effects of numeric-item and string-item terminators and the end-or-identify (EOI) signal during these operations are discussed in this section. After reading this section, you will be able to better understand how enters that use images work and how the default statement-termination conditions are *modified* by the **#**, **%**, **+**, and **-** image specifiers.

## Default Termination Conditions

The default statement-termination conditions for enters that use images are very similar to those required to terminate free-field enters. *Either* of the following conditions will properly terminate an ENTER statement that uses an image.

- An END indication (such as the EOI signal or end-of-data) is received *with* the byte that satisfies the last *image item* or *within 256 bytes after* the byte that satisfied the last image item.
- A line-feed is received *as* the byte that satisfies the last *image item* (exceptions are the "B" and "W" specifiers) or *within 256 bytes after* the byte that satisfied the last image item.

## EOI Re-Definition

It is important to realize that when an enter uses an image (when the secondary keyword "USING" is specified), the definition of the EOI signal is *automatically modified.* If the EOI signal terminates the *last image item,* the entire statement is properly terminated, as with free-field enters. In addition, *multiple EOI signals are now allowed* and act as *item* terminators; however, the EOI must be received *with* the byte that satisfies each image item. If the EOI is received *before* any image is satisfied, it is *ignored.* Thus, all images must be satisfied, and EOI will not cause early termination of the ENTER-USING-image statement.

The following table summarizes the definitions of EOI during several types of ENTER statement. The statement-terminator modifiers are more fully described in the next section.

**Table 5-5. Effects of EOI During ENTER Statements**

| | **Free-Field ENTER Statements** | **ENTER USING** without # or % | **ENTER USING** with # | **ENTER USING** with % |
|---|---|---|---|---|
| **Definition of EOI** | Immediate statement terminator | Item terminator or statement terminator | Item terminator or statement terminator | Immediate statement terminator |
| **Statement Terminator Required?** | Yes | Yes | No | No |
| **Early Termination Allowed?** | No | No | No | Yes |

## Statement-Termination Modifiers

These specifiers modify the conditions that terminate enters that use images. The first one of these specifiers encountered in the image list modifies the termination conditions for the ENTER statement. If another of these specifiers is encountered in the image list, it again modifies the terminating conditions for the statement.

**Table 5-6. Statement-Termination Modifiers**

| Image Specifier | Meaning |
|---|---|
| # | Specifies that a **statement-termination** condition is **not** required; the ENTER statement is automatically terminated as soon as the **last image item** is satisfied. |
| % | Also specifies that a statement-termination condition is not required. **In addition**, EOI is re-defined to be an **immediate** statement terminator, **allowing early termination** of the ENTER **before all** image items have been satisfied. However, the statement can only be terminated on a "legal item boundary". The legal boundaries for different specifiers are as follows.<br>**Specifier** — **Legal Boundary**<br>K,−K — With any character, since this specifies a variable-width field of characters.<br>S,M,D,E Z,.,A,X "literal" B,W — Only with the last character that satisfies the image (e.g., with the 5th character of a "5A" image). If EOI is received with any other character, it is ignored.<br>/ — Only with the last line-feed character that satisfies the image (e.g., with the 3rd line-feed of a "3/" image); otherwise it is ignored. |
| + | Specifies that an END indication is required to terminate the ENTER statement. Line-feeds are ignored as statement terminators; however, they will still terminate items (unless a −K or −H image is used for strings). |
| − | Specifies that a line-feed is required to terminate the statement. EOI is ignored, and other END indications (such as EOF or end-of-data) cause an error if encountered before the line-feed. |

### Examples of Modifying Termination Conditions

```
ENTER @Device USING "#,B";Byte
```
*Enter a single byte.*

```
ENTER @Device USING "#,W";Integer
```
*Enter a single word.*

```
ENTER @Device USING ",K";Array(*)
```
*Enter an array, allowing early termination by EOI.*

```
ENTER @Device USING "+,K";String$
```
*Enter characters into String$ until line-feed received, then continue entering characters until END received.*

```
ENTER @Device USING "-,K";String$
```
*Enter characters until line-feed received; ignore EOI, if received.*

# Additional Image Features

Several additional image features are available with this BASIC language. Some of these features have already been shown in examples, and all of them resemble the additional features of images used with OUTPUT statements.

## Repeat Factors

All of the following specifiers can be preceded by an integer that specifies how many times the specifier is to be used.

**Repeatable Specifiers**

```
D, Z, A, X, /, @, L
```

## Image Re-Use

If there are fewer images than items in the destination list, the list will be re-used, beginning with the first item in the image list. If there are more images than there are items, the additional specifiers will be ignored.

**Examples**

```
ENTER @Device USING "#,B";B1,B2,B3      The "B" is re-used.

ENTER @Device USING "2A,2A,W";A$,B$     The "W" is not used.
```

## Nested Images

Parentheses can be used to nest images within the image list. The hierarchy is the same as with mathematical operations; evaluation is from inner to outer sets of parentheses. The maximum number of levels of nesting is eight.

**Example**

```
ENTER @Source USING "2(B,5A,/),/";N1,N1$,N2,N2$
```

# Table of Contents

# Registers 6

A register is a memory location. Some registers are memory locations on interface cards, while others are memory locations in the computer which are maintained by BASIC to keep track of various conditions related to interfaces. Some registers store parameters that describe the operation of an interface, some store information describing the I/O path to a device, and some are in locations at which interface cards reside (remember that the computer implements "memory-mapped I/O").

Registers are accessed by the computer when executing I/O statements that specify an interface select code, a device selector, or an I/O path name. Thus, each interface and I/O path has its own set of registers. The general programming techniques used to access these registers and the specific definitions of all I/O path registers are given in this chapter; however, the specific definitions of the interface registers are given in the chapter that describes each interface.

There are **three levels of register access.**

- Firmware register(s) are automatically accessed by BASIC when an I/O statement is executed.

```
OUTPUT @File;Data$              Changes file pointer registers.
ENTER @Buffer;Numeric_item      Changes buffer pointer registers.
```

- STATUS and CONTROL (firmware) registers are explicitly accessed by BASIC statements:

```
100  STATUS CRT,13;Crt_height
110  CONTROL CRT,13;Crt_height+3
```

- Interface (hardware) registers are directly read or written.

```
100  READIO 15,0;Card_id
110  WRITEIO 15,3;Intr_mask  ! Write to Breadboard card reg. 3
```

# Interface Registers

A simple example of an interface register being accessed explicitly by the program and then automatically by I/O statements is shown in the following program. Register 0 of interface select code 1 is the "X" screen coordinate at which subsequent characters output to the the CRT will begin being displayed; register 1 is the corresponding "Y" coordinate.

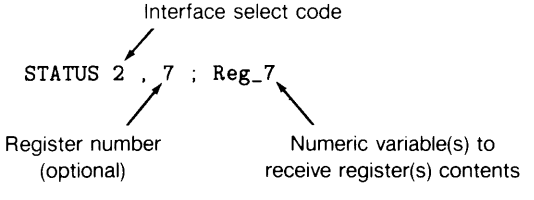
```
100  STATUS CRT;Reg_0,Reg_1 ! Pgrm accessing X & Y coords.
110  OUTPUT CRT;"Print coordinates before 1st OUTPUT:"
120  OUTPUT CRT;"X=";Reg_0,"  Y=";Reg_1
130  OUTPUT CRT
140  !
150  OUTPUT CRT;"1234567";  ! Note ";".
160  STATUS CRT;Reg_0,Reg_1
170  OUTPUT CRT
180  OUTPUT CRT;"Print coordinates after OUTPUTs:"
190  OUTPUT CRT;"X=";Reg_0,"  Y=";Reg_1
200  OUTPUT CRT;" "
210  !
220  END
```

## The STATUS Statement

The contents of a STATUS register can be read with the STATUS statement. Typical examples are shown below. A complete listing of each interface's registers is given in the chapter that describes programming each interface; the definitions of I/O path registers are described later in this chapter.

### Example

STATUS register 7 of the interface at select code 2 is read with the following statement. The first parameter identifies the interface and the optional second parameter identifies which register is to be read. The specified numeric variable receives the register's current contents.



**Example**

I/O path STATUS register 0 is read with the following statement. (Note that this is *not* the same register as keyboard register 0.) Since the second parameter is optional and has been omitted in this instance, register 0 is accessed.

```
100  STATUS @Keyboard;Reg_0
```

**Example**

STATUS registers 4 and 5 of the interface at select code 7 are read with the following statement.

```
100  STATUS 7,4;Reg_4,Reg_5
```

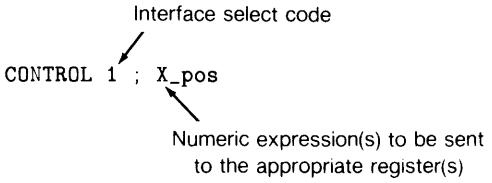
Since two numeric variables are to receive register contents, the next register (5) is accessed. If more than two variables are specified, successive registers are read.

## The CONTROL Statement

When some I/O statements are executed, the contents of some CONTROL registers are automatically changed. For instance, in the above example registers 0 and 1 were changed whenever the OUTPUT statements to the CRT were executed. The program can also change some register's contents with the CONTROL statement, as shown in the following examples. Again, all of the CONTROL register definitions for each interface are given in the chapter that describes programming each interface.

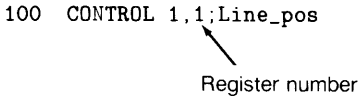
**Example**

Register 0 of interface select code 1 is modified with the following statement. This register determines the "X" screen coordinate at which subsequent characters output to the CRT display will appear.




**Example**

Register 1 of interface select code 1 is modified with the following statement. This register's contents determine the "Y" screen coordinate at which subsequent characters output to the CRT display will appear; changing the contents of this register also allows scrolling the display.



# I/O Path Registers

At this point you know how to access the registers associated with interfaces and I/O path names, but you may not know much about the differences or about the interaction between these two types of registers. Let's first review the definition of an I/O path name.

An I/O path name is a data type that contains a description of an I/O path between the computer and one of its resources sufficient to allow accessing the resource. You learned in the "Directing Data Flow" chapter that the computer uses this information whenever the I/O path name is used in an I/O statement. Much of this information stored in this I/O-path-name table can be accessed with the STATUS and CONTROL statements.

When an I/O path name is used to specify a resource in an I/O statement, BASIC accesses the first table entry (the validity flag) to see if the name is currently assigned.

- If the I/O path name is assigned, the computer reads I/O path register 0 which tells the computer the type of resource involved.
    - If the resource is a device, BASIC must also access the registers of the interface specified by the device selector.
    - If the resource is a file, the table contains additional entries that govern how the I/O process is to be executed.

As you can see, the set of I/O path registers is **not** the same set of registers associated with an interface. The following program is an example of using I/O path register 0 to determine the type of resource to which the I/O path name has been assigned.

```
700 Find_type: STATUS @Resource;Reg_0
710            !
720            IF Reg_0=0 THEN GOTO Not_assigned
730            !
740            IF Reg_0=1 THEN GOTO Device
750            !
760            IF Reg_0=2 THEN GOTO File
770            !
780      PRINT "Resource type unrecognized"
790      PRINT "Program STOPPED."
800      STOP
810            !
820 Not_assigned: PRINT "I/O path name not assigned"
830               GOTO Common_exit
840               !
850 Device: STATUS @Resource,1;Reg_1
860         PRINT "@Resource assigned to device"
870         PRINT "at intf. select code ";Reg_1
880         GOTO Common_exit
890         !
900         !
910 File: STATUS @Resource,1;Reg_1,Reg_2,Reg_3
920       !
930       PRINT "File type         ";Reg_1
940       PRINT "Device selector   ";Reg_2
950       PRINT "Number of sectors ";Reg_3
960       !
970       !
980 Common_exit: !  Exit point of this routine.
```

Once the type of resource has been determined, it can be further accessed with the I/O path registers or the interface registers, depending on the resource type.

- If the I/O path name has been assigned to a **device**, the **interface registers** should be accessed for further information.
- If the name has been assigned to a **mass storage file**, the **I/O path registers** should be accessed.

I/O path names can be assigned to device selectors, files, and buffers. The following program shows an example of determining the interface select code of the resource to which the I/O path name has been assigned.

```
100   ! Example of determining select code
110   ! to which an I/O path name is assigned.
120   !
130 Show_sc: IMAGE "'@Io_path' assigned to ",K,"; Select code = ",D,L
140         !
150   ASSIGN @Io_path TO 701  ! Device selector.
160   Device_selector=FNSc(@Io_path)
170   OUTPUT CRT USING Show_sc;"device 701",Device_selector
180   !
190   ASSIGN @Io_path TO "Data1" ! ASCII file.
200   Device_selector=FNSc(@Io_path)
210   OUTPUT CRT USING Show_sc;"ASCII file",Device_selector
220   !
230   ASSIGN @Io_path TO "Chap1" ! BDAT file.
240   Device_selector=FNSc(@Io_path)
250   OUTPUT CRT USING Show_sc;"BDAT file",Device_selector
260   !
270   ASSIGN @Io_path TO BUFFER [1024] ! Buffer.
280   Device_selector=FNSc(@Io_path)
290   OUTPUT CRT USING Show_sc;"BUFFER",Device_selector
300   !
310   END
320   !
330   DEF FNSc(@Io_path) ! ************************************
340     ! Read I/O path register 0.
350     STATUS @Io_path;Resource_code
360     SELECT Resource_code
370     CASE 0  ! Not assigned.
380       RETURN -1  ! Return a select code out of range.
390       !
400     CASE 1  ! Assigned to a device selector.
410       STATUS @Io_path,1;Select_code
420       RETURN Select_code
430       !
440     CASE 2  ! Assigned to a file specifier.
450       STATUS @Io_path,2;Device_selector
460       RETURN Device_selector MOD 100 ! Remove addressing.
470       !
480     CASE 3 ! Assigned to a buffer.
490       RETURN 0  ! No error, but cannot determine source
500                 ! or destination of transfer to/from buffer.
510     END SELECT
520     !
530     FNEND ! ********************************************
```

The following printout shows a typical example of the program's output.

```
'@Io_path' assigned to device 701; Select code = 7

'@Io_path' assigned to ASCII file; Select code = 7

'@Io_path' assigned to BDAT file; Select code = 7

'@Io_path' assigned to BUFFER; Select code = 0
```

The user-defined function called FNSc interrogates I/O path registers to find the select code. If the I/O path name is currently not assigned, the function returns an arbitrary value of −1 (an invalid value of select code). Since STATUS Register 2 of I/O path names assigned to files contains the entire device selector, which may include addressing information, the function removes any addressing information (Device_selector MOD 100).

Notice that buffers have no select code associated with them, since they are a data type resident in computer memory; thus the function returns a value of 0.

The SC function is a feature of the "Main" BASIC system. The following statements show examples of using this function.

```
Select_code=SC(@Io_path)
IF SC(@File)=4 THEN Device_type$="INTERNAL"
```

The only difference in this language-resident function and the preceding example is that the SC function reports an error if the I/O path specified as its argument is not assigned, rather than returning a select code out of range.

# Summary of I/O Path Registers

The following list describes the information contained in I/O path STATUS and CONTROL registers. Note that only STATUS register 0 is identical for **all** types of I/O paths; the rest of the I/O path registers' contents depend on the **type** of resource to which the name is assigned.

## For All I/O Path Names

**STATUS Register 0**

0 = Invalid I/O path name
1 = I/O path name assigned to a device
2 = I/O path name assigned to a data file
3 = I/O path name assigned to a buffer

## I/O Path Names Assigned to a Device

**STATUS Register 1** Interface select code

**STATUS Register 2** Number of devices

**STATUS Register 3** Address of 1st device

If assigned to more than one device, the addresses of the other devices are available starting in STATUS Register 4.

## I/O Path Names Assigned to an ASCII File

**STATUS Register 1** File type = 3

**STATUS Register 2** Device selector of mass storage device

**STATUS Register 3** Number of records

**STATUS Register 4** Bytes per record = 256

**STATUS Register 5** Current record

**STATUS Register 6** Current byte within record

## I/O Path Names Assigned to a BDAT File

| | |
|---|---|
| **STATUS Register 1** | File type = 2 |
| **STATUS Register 2** | Device selector of mass storage device |
| **STATUS Register 3** | Number of defined records |
| **STATUS Register 4** | Defined record length |
| **STATUS Register 5** | Current record |
| **CONTROL Register 5** | Set record |
| **STATUS Register 6** | Current byte within record |
| **CONTROL Register 6** | Set byte within record |
| **STATUS Register 7** | EOF record |
| **CONTROL Register 7** | Set EOF record |
| **STATUS Register 8** | Byte within EOF record |
| **CONTROL Register 8** | Set byte within EOF record |

## I/O Path Names Assigned to a Buffer

| | |
|---|---|
| **STATUS Register 0** | 0 = Invalid I/O path name<br>1 = I/O path assigned to a device<br>2 = I/O path assigned to a data file<br>3 = I/O path assigned to a buffer |

When the status of register 0 indicates a buffer (3), the status and control registers have the following meanings.

| | |
|---|---|
| **STATUS Register 1** | Buffer type (1=named, 2=unnamed) |
| **STATUS Register 2** | Buffer size in bytes |
| **STATUS Register 3** | Current fill pointer |
| **CONTROL Register 3** | Set fill pointer |
| **STATUS Register 4** | Current number of bytes in buffer |
| **CONTROL Register 4** | Set number of bytes |
| **STATUS Register 5** | Current empty pointer |
| **CONTROL Register 5** | Set empty pointer |

**STATUS Register 6** Interface select code of inbound TRANSFER

**STATUS Register 7** Interface select code of outbound TRANSFER

**STATUS Register 8** If non-zero, inbound TRANSFER is continuous

**CONTROL Register 8** Cancel continuous mode inbound TRANSFER if zero

**STATUS Register 9** If non-zero, outbound TRANSFER is continuous

**CONTROL Register 9** Cancel continuous mode outbound TRANSFER if zero

**STATUS Register 10** Termination status for inbound TRANSFER

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| 0 | TRANS-FER Active | TRANS-FER Aborted | TRANS-FER Error | Device Termi-nation | Byte Count | Record Count | Match Character |
| Value=0 | Value=64 | Value=32 | Value=16 | Value=8 | Value=4 | Value=2 | Value=1 |

**STATUS Register 11** Termination status for outbound TRANSFER

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| 0 | TRANS-FER Active | TRANS-FER Aborted | TRANS-FER Error | Device Termi-nation | Byte Count | Record Count | 0 |
| Value=0 | Value=64 | Value=32 | Value=16 | Value=8 | Value=4 | Value=2 | Value=0 |

**STATUS Register 12** Total number of bytes transferred by last inbound TRANSFER

**STATUS Register 13** Total number of bytes transferred by last outbound TRANSFER

## Direct Interface Access

The third level of register access provides **direct** access to interface hardware; this level of access is identical to that possessed by the operating-system firmware. Consequently, these interface-access techniques should **only** be used if you have a **complete** understanding of both the specified register's definition and of the consequences of reading from or writing to these registers. The READIO and WRITEIO interface register definitions and access methods are listed in the chapter that describes each interface.

# Table of Contents

# Interrupts and Timeouts 7

The computer can sense and respond to the occurrence of several types of interrupt events. This chapter describes programming techniques for handling the interface events called "interrupts" and "timeouts" which can initiate program branches. For more details on event-initiated branches, consult the "Program Structure and Flow" chapter of *BASIC Programming Techniques*, and the *BASIC Language Reference* descriptions of the keywords described in this chapter.

## Overview of Event-Initiated Branching

Event-initiated branches are very powerful programming tools. With them, the computer can execute special routines or subprograms whenever a particular event occurs; the program doesn't have to take time to periodically check for each event's occurrence.

This section describes the general topic of event-initiated branching. Subsequent sections take a closer look at interrupt events.

### Types of Events

The statements that enable events to initiate branches are summarized as follows:

**ON CDIAL**—occurs when one of the nine "knobs" (rotary pulse generators) of an HP 46085 Control Dial Box is turned. (See the "Communicating with the Operator" chapter of *BASIC Programming Techniques* for details.

**ON END**—occurs when the computer encounters the end of a mass storage file while accessing the file. (See the "Data Storage and Retrieval" chapter of *BASIC Programming Techniques* for details.)

**ON ERROR**—occurs when a program-execution error is sensed. (See the "Handling Errors" chapter of *BASIC Programming Techniques* for details.)

**ON KEY**—occurs when a currently defined softkey is pressed. (See the "Program Structure and Flow" chapter of *BASIC Programming Techniques* or the "Keyboard Interfaces" chapter of this manual for details.)

**ON KNOB** occurs when the "knob" (rotary pulse generator) is turned. (See the "Program Structure and Flow" chapter of *BASIC Programming Techniques* and the "Keyboard Interfaces" chapter of this manual for details.)

**ON INTR**— occurs when an interrupt is requested by a device or when an interrupt condition occurs at the interface. (Discussed in this chapter.)

**ON TIMEOUT**—occurs when the computer has not detected a handshake response from a device within a specified amount of time. (Discussed in this chapter.)

## A Simple Example

The following program shows how events are serviced by the computer. Subprograms called "Key_1" and "Key_2" are the service routines for the events of pressing softkeys [f1] and [f2] ([k1] and [k2] on 98203 keyboards) being pressed; the software priorities assigned to these events are 3 and 4, respectively. Run the program and alternately press these softkeys; the branch to each key's service routine is initiated by pressing the key. The system priority is "graphed" on the CRT display.

```
150 ON KEY 1,3 CALL Key_0   ! Set up events and
160 ON KEY 2,4 CALL Key_1   ! assign priorities.
170 !
180 OUTPUT CRT;" System","Priority"
190 V$=CHR$(8)&CHR$(10)     ! BS & LF.
200 OUTPUT CRT;"   4"&V$&"3"&V$&"2"&V$&"1"&V$&"0"
210 !
220 Main: CALL Bar_graph(7,"*") ! Sys. prior. is
230                             ! always >= 0.
240       BEEP 100,.1           ! Low tone.
250       FOR Jiffy=1 TO 5000
260       NEXT Jiffy
270       !
280   GOTO Main             ! Main loop.
290   !
300   END
310   !
320 SUB Key_1
330     CALL Bar_graph(4,"*") ! Plot priority.
340     BEEP 300,.1           ! Middle tone.
350     FOR Iota=1 TO 2000
360     NEXT Iota
370     CALL Bar_graph(4," ") ! Erase.
380   SUBEND
390   !
400 SUB Key_2
410     CALL Bar_graph(3,"*") ! Graph priority.
420     BEEP 400,.1           ! High tone.
430     FOR Twinkle=1 TO 2000
```

```
440     NEXT Twinkle
450     CALL Bar_graph(3," ") ! Erase.
460   SUBEND
470   !
480 SUB Bar_graph(Line,Char$)
490     CONTROL 1,1;Line    ! Locate line.
500     OUTPUT 1;Char$      ! Bar-graph character.
510     SUBEND
```

If [f2] is pressed **after** [f1] is pressed, **but while** the Key_1 routine is being executed, execution of Key_1 is **temporarily interrupted** and the Key_2 routine is executed. When Key_2 is finished, execution of Key_1 is resumed at the point where it was temporarily interrupted. This occurs because [f2] was assigned a **higher software priority** than [f1].

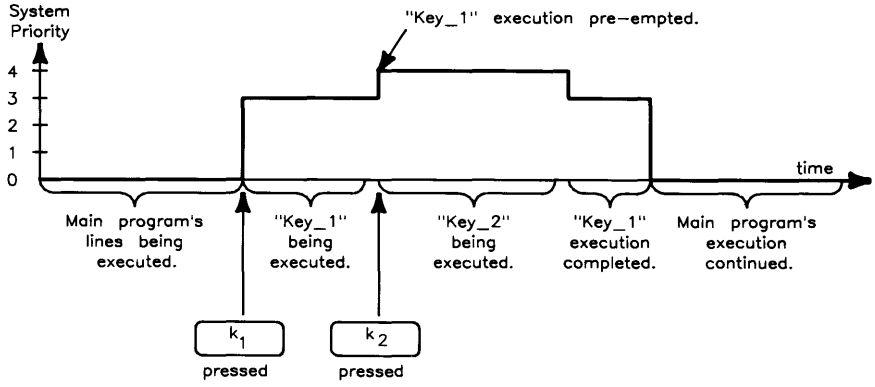



**Figure 7-1. Events with Higher Software Priority Take Precedence**

On the other hand, if [f1] is pressed **while** [f2] is being serviced, the computer finishes executing Key_2 **before** executing Key_1. The event of pressing [f1] was "logged" but **not processed** until **after** the routine having **higher software priority** was completed. This is a very important concept when dealing with event-initiated branching. The action of the computer in logging events and determining assigned software priority is further described in the next section.

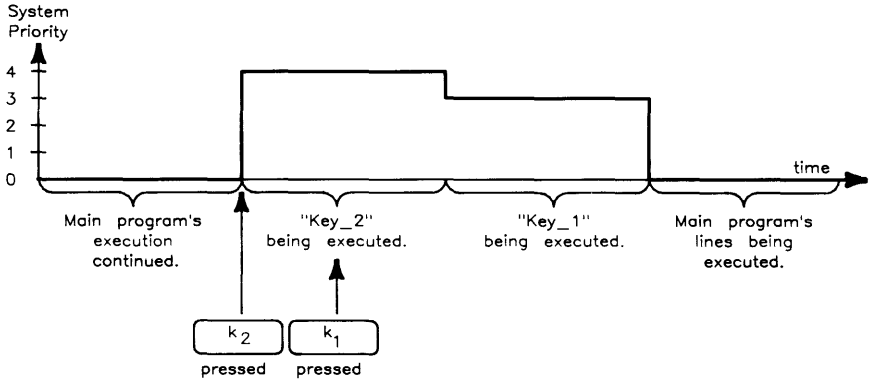



**Figure 7-2. An Event with Lower Software Priority Must Wait**

## Conditions Required for Initiating a Branch

In order for any event to initiate a branch, the following prerequisite conditions must be met. The preceding section showed a simple example of softkey events, which are similar to interface interrupts. This section describes the additional requirements for servicing interface interrupts. Later sections show more details of meeting these requirements.

1. The branch must be **set up** by an ON-event-branch statement, and the *service routine* must exist.

   ```
   100  ON INTR GOSUB Check_device
              :
   920 Check_device: ! Service routine for interface interrupts.
   ```

   The term *service routine* is any legal branch location for the type of branch specified (GOSUB, GOTO, CALL[1], or RECOVER) and current context. The "Program Structure and Flow" chapter of *BASIC Programming Techniques* and the *BASIC Language Reference* fully describe the differences between these types of branches.

2. Before an event (which is set up) can initiate a branch, it must first be enabled to do so. With non-interrupt events (such as ON KEY, and ON KNOB), the event is *automatically* enabled when the ON-event statement is executed. However, with ON INTR, you must explicitly enable the interrupt to initiate its corresponding branch. For example, to enable the interface at select code 7 to initiate an interrupt branch:

   ```
   110  ENABLE INTR 7;Intr_mask
   ```

   Further details of enabling these events are described in the "Interface Interrupts" and "Interface Timeouts" sections of this chapter.

3. The event must **occur** and be **logged** by the BASIC system. (For instance, the HP-IB "Service Request" signal is sent from the device to the computer and is logged by the BASIC operating system.)

4. The **software priority** assigned to the event must be greater than the current SYSTEM PRIORITY[2].

When all of these conditions have been met, the branch is taken.

---

[1] Parameters cannot be passed to the service routine in an ON INTR CALL statement; any variables to be used jointly by the service routine and other contexts must be defined in common. See the "Subprograms" chapter of *BASIC Programming Techniques* or the *BASIC Language Reference* for further details.

[2] Software priority is specified in the event's set-up statement; the range of priorities that can be specified in this statement is 0 through 15. Interfaces also have a "hardware" priority which is different from the software priority. The following sections describe details of hardware and software priority.

## Logging and Servicing Events

The preceding events may occur at any time; however, the computer is only "concerned" if these events have been "set up" to initiate a branch. An example of the computer ignoring an event is seen when an undefined softkey is pressed. Since the event has not been set up, the computer beeps. No service routine is executed, even though the computer was "aware" of the event. Thus, only when an event is first set up and then occurs does the computer "service" its occurrence.

### Software Priority

The computer first "logs" the occurrence of an event which is set up.[1] After recording that the event occurred, the computer then checks the event's software priority against that of the routine currently being executed. The priority of the routine currently being executed is known as **system priority**. If no service routine is being executed, the system priority is 0; otherwise the system priority is equal to the assigned software priority of the routine currently being executed. The following table lists the software priority structure of the BASIC system; priority increases from 0 to 17.

**Table 7-1. Software Priorities of Events**

| Software Priority (SYSTEM PRIORITY) | Explanation |
|---|---|
| 0 | System priority when no service routine is being executed (known as the "quiescent level"). |
| 1 thru 15 | Software-assignable priorities of service routines. |
| 16 | Effective software priority of ON END and ON TIMEOUT; the software priorities of these events **cannot** be changed. |
| 17 | Effective software priority of ON ERROR; the software priorities of these events **cannot** be changed. |

In the above example, system priority was 0 before either of the events occurred. When [f1] was pressed, the system priority became 3. When [f2] was subsequently pressed, the system first logged the event and then checked its priority against the current system priority. Since [f2] had been assigned a priority of 4, it pre-empted [f1]'s service routine because of its higher software priority.

[1] The process of logging event occurrences is described in the section called "Hardware Priority".

It is important to **note that BASIC only services event occurrences when a program line is exited.** This change of lines occurs either:

- at the end of execution of a line, or
- when the line is exited when a user-defined function is called.

When the program line is changed, the computer attempts to service all events that have occurred since the last time a line was exited. The next sections further describe logging and servicing events.

When execution of Key_2 started, the system priority was set to 4. If any event was to interrupt the execution of this service routine, it must have had a software priority of 5 (or greater). When execution of Key_2 completed, the Key_1 service routine had the highest software priority, so its execution was resumed at the point at which it was interrupted.

If [f1] was pressed **again** while its own service routine was being executed, execution of the first service routine was finished before the service routine was executed again. Thus, if an event occurs that has the **same** software priority as the system priority, its service routine will **not** interrupt the current routine. The service routine will **only** be executed if the event's software priority becomes the highest priority of any event which has been logged (i.e., **after all** other events of **higher** software priority have been serviced).

### Changing System Priority

Events are assigned a software priority to allow the computer to respond to occurrences of events with high software priority before those with lower priorities. Occasionally, service routines may contain code segments that should not be interrupted once their execution begins. In such cases, the entire service routine may not require a high software priority, even though a portion of the routine needs a high priority to ensure that it will not be interrupted by most other processes.

The SYSTEM PRIORITY statement can be used in these cases to set the system priority to a level higher than the BASIC system would otherwise set it when the branch to the service routine is taken. The current system priority can also be determined by calling SYSTEM$("SYSTEM PRIORITY"), which returns a string value of the current system priority in the range 0 through 15. Examples are shown in the following program.

```
100   GINIT   ! Use default plotter is CRT.
110   GRAPHICS ON
120   VIEWPORT 0,131,30,100
130   WINDOW 0,2000,0,7
140   !
150   ON KEY 1 LABEL "Prior.1",1 GOSUB Key_1
160   ON KEY 2 LABEL "Prior.2",2 GOSUB Key_2
170   ON KEY 3 LABEL "Prior.2",3 GOSUB Key_3
180   !
190   Sys_prior$="SYSTEM PRIORITY" ! Define string for SYSTEM$.
200   !
210 Main_program: !
220   DISP "Quiescent system priority level = 0."
230   X=X+1
240   Sys_prior=VAL(SYSTEM$(Sys_prior$))
250   GOSUB Plot_priority
260   GOTO Main_program
270   !
280 Key_1:   FOR Iota=1 TO 100
290              DISP "Key 1; priority 1."
300              X=X+1
310              Sys_prior=VAL(SYSTEM$(Sys_prior$))
320              GOSUB Plot_priority
330          NEXT Iota
340          RETURN
350          !
```

```
360 Key_2:   FOR Twinkle=1 TO 100
370              DISP "Key 2; priority 2."
380              X=X+1
390              Sys_prior=VAL(SYSTEM$(Sys_prior$))
400              GOSUB Plot_priority
410          NEXT Twinkle
420          !
430          ! Critical routine raise system priority.
440          SYSTEM PRIORITY 3
450          FOR Split_second=1 TO 100
460              DISP "Subroutine set system priority to 3."
470              X=X+1
480              Sys_prior=VAL(SYSTEM$(Sys_prior$))
490              GOSUB Plot_priority
500          NEXT Split_second
510          !
520          ! System priority lowered when finished.
530          SYSTEM PRIORITY 0
540          RETURN
550          !
560 Key_3:   FOR Jiffy=1 TO 100
570              DISP "Key 3; priority 3."
580              X=X+1
590              Sys_prior=VAL(SYSTEM$(Sys_prior$))
600              GOSUB Plot_priority
610          NEXT Jiffy
620          RETURN
630          !
640 Plot_priority:  !
650                 IF X>2000 THEN ! Draw new plot.
660                     GCLEAR
670                     MOVE 0,0
680                     X=0
690                 END IF
700                 PLOT X,Sys_prior
710                 RETURN
720                 !
730                 !
740    END
```

The subroutine called Key_2 raised the system priority from its current level, 2, to level 3 during the time that the second FOR..NEXT loop was being executed. During this time, pressing [f3] will not interrupt the routine, since a priority of 4 or greater is required to interrupt the Key_2 routine.

By setting the system priority level in this manner, routines can selectively allow and disallow other routines from being executed; routines with higher software priority are allowed to pre-empt the routine, while those with the same or lower priority are not. If no other events are to interrupt the process, system priority can be set to 15. However, keep in mind that END, ERROR, and TIMEOUT events have effective software priorities higher than 15 and can therefore interrupt the service routine (if a branch for one of these events is currently set up).

When the "critical" code has been executed, the program returns the system priority to the value set by the BASIC system when the branch was taken (which was 2 since the Key_2 event was being serviced). Of course, if an event with higher software priority occurs while the code segment is being executed, its service routine will pre-empt the critical code segment.

This technique can also be used within SUB and FN subprograms. Keep in mind that when program control is returned from a context, the system priority is returned to the value it had when the context was called.

### Hardware Priority

There is a second event priority, hardware priority, that also influences the order in which the computer responds to events.

- Hardware priority determines the order in which events are **logged** by the system.
- Software priority determines the order in which events are **serviced**.

The hardware priority of an interface interrupt is determined by the priority-switch setting on the interface card itself[1]. **Hardware priority is independent of the software priority assigned to the event by the ON INTR statement.**

[1] Setting hardware priority on an optional interface is described in the interface's installation manual.

All events have a hardware priority, but not all have hardware priorities that can be changed. The following table lists the hardware-priority structure of Series 200/300 computers. Only the optional interfaces' hardware priorities can be changed.

**Table 7-2. Hardware Priorities of Interfaces**

| Hardware Priority | Interface(s) and Event(s) at This Priority |
|---|---|
| 0 | (Quiescent level; no interface is currently interrupting) |
| 1 | Built-in Keyboard<br>(KEY and KNOB events) |
| 2 | Built-in Disc Drive of 226/236<br>(END event) |
| 3 | Built-in HP-IB or Serial interfaces<br>(INTR and TIMEOUT events) |
| 3–6 | Optional Interface Cards<br>(INTR and TIMEOUT events) |
| 7 | Non-Maskable Interrupts, such as the RESET (Break) key |

In order to fully understand the differences between hardware and software priority, it is helpful to first understand how the computer logs and services events. When any event occurs, the interface (at which the event has occurred) signals it to the computer. The computer responds by temporarily suspending execution of its current task to **poll** (interrogate) the currently enabled interfaces.

When the computer determines which interface is interrupting, it records that it has occurred on this interface (i.e., logs the event) and **disables further interrupts from this interface**. This event is now **logged** and **pending service** by the computer. The computer can then return to its former task (unless other events have occurred which have not been logged).

If other events have occurred but have not yet been logged, they will be **logged in order of descending hardware priority**. This occurs because events with hardware priority lower than that of the event currently being logged are **ignored** until all events with the current hardware priority are logged.

## Servicing Pending Events

If BASIC was interrupted while executing a program line, execution of the line is resumed (after logging all events) and continues until either the line is completely executed or a user-defined function causes the line to be exited. When the line is exited, BASIC begins servicing all pending events.

When servicing pending events, the following rules are used to determine the order in which they are serviced:

1. Highest software priority first, lowest software priority last.
2. If two or more events have the same software priority, the BASIC services the events in order of descending interface select codes.
3. If events have both the same software priority and interface select code (such as softkeys with the same software priority), the events are serviced in the order in which they occurred.

The process of logging of events is still taking place while events are being serviced. This concurrent action has two major effects.

1. Events of higher hardware priority will interrupt the current activity to be logged by the computer.
2. Events which also have higher software priority will interrupt the computer's present activity to be serviced.

Thus, events of high hardware and software priority can potentially occur and be serviced many times between program lines.

For example, suppose that the following events have been set up and enabled to initiate branches. Assume that the events have the hardware priorities shown in the program's comments.

```
100   ON INTR 8,15 CALL Serv_8   ! Hardware priority 6.
110   ON INTR 7,14 CALL Serv_7   ! Hardware priority 3.
120   ON KEY 0,5 CALL Serv_k0    ! Hardware priority 1.
```

The following diagram shows the INTR event on interface select code 8 occurring and being serviced several times after one program line has been exited.

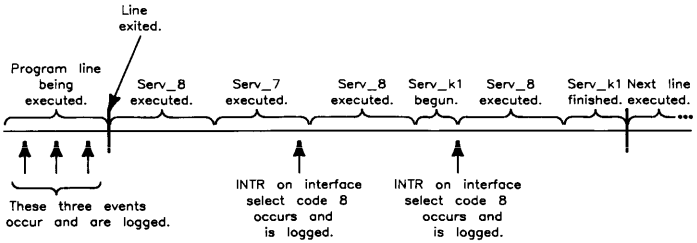



**Figure 7-3. INTR Event Servicing Example**

**Hardware priority's main function** is to keep events of lower hardware priority from being logged so that more "urgent" events can be serviced quickly. Decreasing the system's response time to these urgent events may also *increase overall system throughput.*

# Interface Interrupts

All interfaces have a hardware line dedicated to signal to the computer that an interrupt event has occurred. The source of this signal can be either the device(s) connected to the interface or the interface hardware itself. These possibilities are shown in the following diagram.

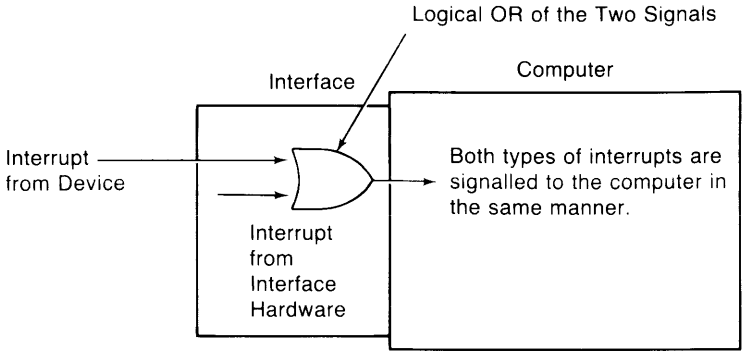



**Figure 7-4. Interface Interrupts**

There are **two general types of interrupt events**.

- One type of event occurs when a **device** determines that it requires the computer to execute a special procedure.
- The second type occurs when the **interface itself** determines that a condition exists or has occurred that requires the computer's attention.

The first type of interrupt event is usually called a **service request**. Service requests **originate at the device**. An example is a voltmeter signaling to the computer that it has a reading; another is a printer generating a service request when it is out of paper. The service routine takes the appropriate action, and the program (usually) resumes execution.

The second type of interrupt event is used to inform the computer of a **specific condition** at the interface. This type of event **originates at the interface**. An example of this interrupt event is the occurrence of a parity error detected by the serial interface. This error usually requires that the erroneous data just received be re-transmitted. The service routine can often correct this error by telling the sender to keep sending the data until the error no longer occurs, after which the computer can resume its former task.

The specific abilities of each interface to detect interrupt conditions and to pass on service requests from devices are described in the interface programming chapters.

## Enabling Interrupt Events

Before the INTR event can initiate its branch, it must be enabled to do so. The following examples show how to enable interrupt events to initiate branches.

### Example

Enable interrupts occurring at interface select code 7 to initiate the branch set up by an ON-event-branch statement.

```
ENABLE INTR 7;Mask
```

The bit pattern of Mask is copied into the "interrupt-enable" register of the specified interface; in this case, register 4 of the built-in HP-IB interface receives Mask's bit pattern. **Individual bits of the mask** are used to enable different types of interrupt events for each interface. Each bit which is **set** (i.e., which has a value of 1) in the mask expression **enables** the corresponding interrupt condition defined for that bit.

For instance, bit 1 of the HP-IB's interrupt-enable register is used to enable and disable service-request interrupts. To enable this event to initiate a branch, bit 1 must be set to a "1". Specifying a mask parameter of "2" causes a value of 2 to be written into this register, thus enabling **only** service requests to initiate branches.

```
ENABLE INTR 7;2
```

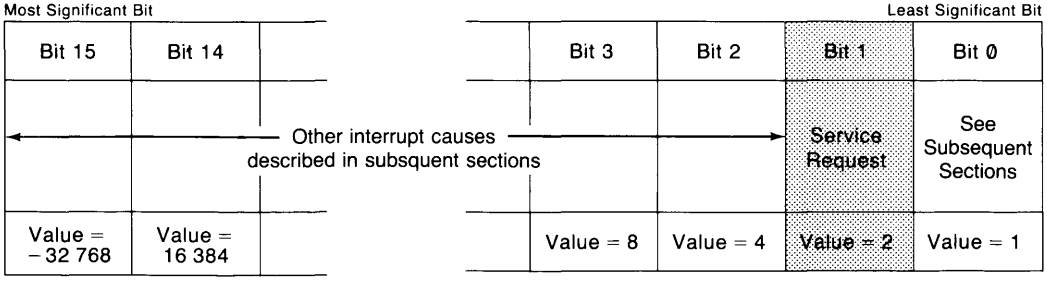



**Figure 7-5. HP-IB Interrupt-Enable Register**

The mask parameter is **optional**.

- If it is included, the specified value is written into the appropriate register of the specified interface.
- If this parameter is omitted, the mask specified in the **last** ENABLE INTR is used. If **no** ENABLE INTR statement has been executed for the specified interface, a value of 0 is used (all interrupt events **disabled**).

**Example**

Re-enable a previously enabled interrupt event.

```
ENABLE INTR 7
```

Since no interrupt-enable mask is specified, the last mask used to enable interrupts on this interface is used.

### Enabling and Disabling Events with WRITEIO

This section shows how to use WRITEIO to perform the same functions as ENABLE INTR and DISABLE INTR statements. The examples are shown for the HP 98630 Breadboard Interface, an interface for which no driver is installed (and therefore will not permit ENABLE INTR and DISABLE INTR to be used).

ON INTR and OFF INTR statements may be executed for *any* I/O card plugged into the computer. However, if there is no driver currently loaded for an interface card, all other I/O statements (CONTROL, STATUS, ENABLE INTR, OUTPUT, ASSIGN, etc.) will generate an `ERROR 163 I/O interface not present` message. Before an interrupt can be generated by a "driverless" interface card, you must emulate the ENABLE INTR statement by using WRITEIO. For example, if an HP 98630 Breadboard card is at select code 17, the following statements set up the service routine "My_card_isr" and enable interrupts for this card:

```
100  ON INTR CALL My_card_isr
110  WRITEIO 17,Mask_reg;Mask_value ! Set the mask.
120  WRITEIO 17,3;128               ! Enable interrupts.
```

The two WRITEIO statements simulate the function of the ENABLE INTR statement.

When the Breadboard card interrupts, BASIC clears bit 7 of WRITEIO register 3 (the interrupt enable bit) and logs the interrupt so that the service routine will be called the next end-of-line (if system priority permits). No other actions are taken during the *hardware* interrupt-logging routine; however, the *software* service routine is free to do whatever you want it to do.

To perform a DISABLE INTR function, execute this statement:

```
300  WRITEIO 17,3;0 ! Disable interrupts.
```

Use this information as required, especially if you wish to use the HP 98630 Breadboard card for customized I/O.

## Service Requests

You can program a service routine to perform any task(s) that is "requested" by the device that initiated the branch. If this event can occur for only one reason, the service routine just performs the specified action. However, with many devices, the service request can occur for several different reasons. In this case, the program must have a means of determining **which** event(s) occurred and then take action.

### Example

The following program shows an example of using a service routine that can be initiated by only one cause — a service request from a device at address 22 on the built-in HP-IB interface.

```
100   ! Example of service routine for HP-IB service requests.
110   !
120   ON INTR 7,5 CALL Intr7    ! Set up interface, priority,
130                             ! branch type, and location.
140                             !
150   ENABLE INTR 7;2           ! Only service requests
160                             ! (bit 1) are enabled.
170                             !
180 Loop: GOTO Loop             ! Idle loop.
190                             !
200   END
210                             !
220  SUB Intr7
230        Z=SPOLL(722)         ! Clear INTR cause first.
240                             !
250        ENTER 722;Reading    ! Take desired action.
260                             !
270        ENABLE INTR 7        ! Re-enable service requests.
280                             !
290      SUBEND
```

The program shows the sequence of steps required to set up and enable interrupt events. These steps are as follows.

1. The interrupt event is **set up** to be logged, as in line 120. This statement also assigns the event's software priority; in this case, the priority is 5.
2. The event must be **enabled** to initiate its branch, as in line 150. The mask value specifies that only service requests (enabled by setting bit 1) can initiate branches.
3. When the event occurs it is **logged**. Any further interrupts from this interface are automatically disabled until this interrupt event is serviced.
4. **Determine the interrupt's cause.** On HP-IB interfaces, a serial poll (line 230) must be performed by the service routine, clearing the interrupt-cause register so that the same event will not cause another branch upon return to the interrupted context. (The serial poll is particular to the HP-IB interface, but analogous actions can be performed to determine interrupt causes on other interfaces.)
5. The actual **requested action is performed** (line 250).
6. If subsequent events are to also initiate branches, they must be **re-enabled** before resuming execution of the previous program segment, as in line 270. Since no interrupt-enable mask is explicitly specified, the previous mask is used.

## Interrupt Conditions

The conditions that can be sensed by each type of interface are different. All interrupt conditions signal to the computer that either its assistance is required to correct an error situation or an operating mode of the interface has changed and must be made known to the computer.

The following service routine demonstrates typical action taken when a receiver-line status ("RLS") interrupt condition is sensed by the serial interface.

```
100   ! Example of interface-condition interrupt event.
110
120   ON INTR 9,4 CALL Intr_9  !  Set up for interface select
130                            !  code 9 and priority of 4.
140   ENABLE INTR 9;4          !  Bit 2 in mask enables
150                            !  "RLS"-type interrupts only.
```

•
• *Main program.*
•

```
600   SUB Intr_9
610         !
620         STATUS 9,10;Intr_cause' ! Clear intr.-cause reg.
630         !
640         ! Check errors and branch to "fix" routines.
650         !
660         IF BIT(Intr_cause,3)=1 THEN GOTO Framing error
670         IF BIT(Intr_cause,2)=1 THEN GOTO Parity_error
680         IF BIT(Intr_cause,1)=1 THEN GOTO Overrun_error
690         IF BIT(Intr_cause,0)=1 THEN GOTO Recv_buf_full
700         ENABLE INTR 9,4      ! Ignore others, re-enable
710         SUBEXIT              ! INTRs, and return.
720         !
730 Framing_error: ! "Fix" and re-enable.
740                SUBEXIT
750                !
760 Parity_error:  ! "Fix" and re-enable.
770                SUBEXIT
780                !
790 Overrun_error: ! "Fix" and re-enable.
800                SUBEXIT
810                !
820 Recv_buf_full: ! "Fix" and re-enable.
830                SUBEXIT
840     SUBEND
```

# Interface Timeouts

A "timeout" occurs when the handshake response from any external device takes longer than the specified amount of time. The time specified for the timeout event is usually the maximum time that a device can be expected to take to respond to a handshake during an I/O statement.

## Setting Up Timeout Events

The following statements set up this event-initiated branch. The software priority of this event **cannot** be assigned by the program; it is permanently assigned priority 15. The maximum time that the computer will wait for a response from the peripheral can be specified in the statement with a resolution of 0.001 seconds.

**Example**

Set up a timeout to occur after the Serial Interface has not detected a response from the peripheral after 0.200 seconds. Branch to a subroutine called "Serial_down".

```
ON TIMEOUT 9,.2 GOSUB Serial_down
```

**Example**

Set up a timeout of 0.060 for the interface at select code 8.

```
ON TIMEOUT 8,.06 GOTO Hp_ib_status
```

## Timeout Limitations

Timeout events cannot be set up for any of the internal interfaces except the built-in HP-IB.

Event-initiated branches are only executed at certain times during program execution, usually after a program line has been executed. Consequently, BASIC may wait up to 25% longer than the specified time to detect a timeout event; however, it will *always* wait *at least* the specified amount of time before generating the interrupt.

There is **no default** timeout time parameter. Thus, if no ON TIMEOUT is executed for a specific interface, the computer will wait **indefinitely** on the device to respond. The only way that the computer can continue executing the program is for the operator to use the CLR I/O ([Break]) key. This key aborts the I/O operation that was left "hanging" by the failure of the device to respond to and complete the handshake.

The times specified for timeouts are passed to subprograms. Thus, unless the time for a timeout event is changed in the subprogram, it remains the same as it was in the calling routine. If the time parameter is changed by the subprogram, it is restored to its former value upon return to the calling context.

# Table of Contents

# I/O Path Attributes 8

This chapter contains two major topics, both of which involve additional features provided by I/O path names.

- The first topic is that I/O path names can be given attributes which control the way that the system handles the data sent and received through the I/O path. Attributes are available for such purposes as controlling data representations, generating and checking parity, and defining special end-of-line (EOL) sequences.
- The second topic is that one set of I/O statements can be used to access most system resources, including the CRT display, the keyboard, mass storage files, and buffers (instead of using a separate set of BASIC statements to access each class of resources). This second topic, herein called "unified I/O", may be considered an implicit attribute of I/O path names.

# The FORMAT Attributes

All I/O paths used as means to move data have certain attributes, which involve both hardware and software characteristics. For instance, some interfaces handle 8-bit data, while others can handle either 8-bit or 16-bit data. Some I/O operations involve sending ASCII data (for "human consumption"), while others may involve sending data in an "internal" form (that is easier for the computer to understand). This second characteristic, data representation, is what the format attributes control.

## Two FORMAT Attributes Are Available

All I/O paths possess one of the two following attributes:

- FORMAT ON— means that the data are sent in ASCII representation[1].
- FORMAT OFF— means that the data are sent in BASIC internal representation[1].

Before getting into how to assign these attributes to I/O paths, let's take a brief look at each one.

### FORMAT ON

With FORMAT ON, internally represented numeric data must be "formatted" into its ASCII representation before being sent to the device. Conversely, numeric data being received from the device must be "unformatted" back into its internal representation. These operations are shown in the diagrams below:

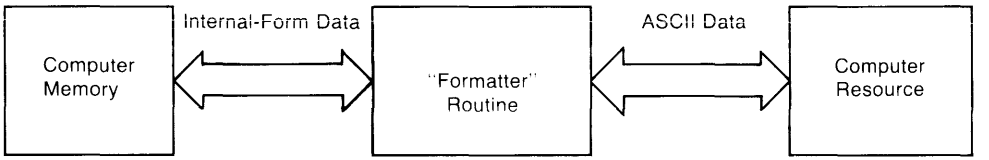



**Figure 8-1. Numeric Data Transformations with FORMAT ON**

For more information about the ASCII data format, see the "Interfacing Concepts" chapter. For details of how items and I/O statements are terminated, see the "Outputting Data" and "Entering Data" chapters.

[1] Complete descriptions of these data representations are given in the "Interfacing Concepts" chapter.

### FORMAT OFF

With FORMAT OFF, however, no formatting is required. The data items are merely copied from the source to the destination. This type of I/O operation requires less time, since fewer steps are involved.

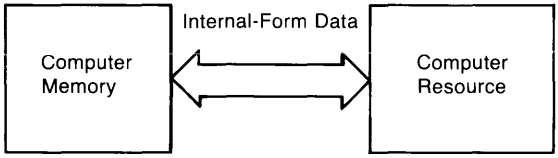



**Figure 8-2. Numeric Data Transfer with FORMAT OFF**

The only requirement is that the resource also use the exact same data representations as the internal BASIC representation.

Here are how each type of data item is represented and sent with FORMAT OFF:

- INTEGER: two-byte (16-bit), two's complement.
- REAL: eight-byte (64-bit) IEEE floating-point standard.
- COMPLEX: same as two REAL values.
- String: four-byte (32-bit) length header, followed by ASCII characters. An additional ASCII space character, CHR$(32), may be sent and received with strings in order to have an even number of bytes.

Here are the FORMAT OFF rules for OUTPUT and ENTER operations:

- No item terminator and no EOL sequence are sent by OUTPUT.
- No item terminator and no statement-termination conditions are required by ENTER.
- No *non-default* CONVERT or PARITY attribute may be assigned to the I/O path (discussed later in this chapter).
- If either OUTPUT or ENTER uses an IMAGE (such as with OUTPUT 701 USING "4D.D"), then the FORMAT ON attribute is *automatically* used.

## Assigning Default FORMAT Attributes

As discussed in the "Directing Data Flow" chapter, names are assigned to I/O paths between the computer and devices with the ASSIGN statement. Here is a typical example:

```
ASSIGN Any_name TO Device_selector
```

This assignment fills a "table" in memory with information that describes the I/O path. This information includes the device selector, the path's FORMAT attribute, and other descriptive information. When the I/O path name is specified in a subsequent I/O statement (such as OUTPUT or ENTER), this information is used by the system in completing the I/O operation.

Different default FORMAT attributes are given to devices and files:

- **Devices**— since most devices use an ASCII data representation, the default attribute assigned to devices is FORMAT ON. (This is also the default for ASCII files and BUFFERs, as discussed later in this chapter and in the next chapter.)
- **BDAT and HPUX files**— the default for BDAT and HPUX files is FORMAT OFF. (This is because for numeric quantities, the FORMAT OFF representation requires no translation time for numeric data; this is possible because humans never see the data patterns written to the file, and therefore the items do not have to be in ASCII, or humanly readable, form.)

One of the most powerful features of this BASIC system is that you can change the attributes of I/O paths programmatically.

## Specifying I/O Path Attributes

There are two ways of specifying attributes for an I/O path:

- Specify the desired attribute(s) when the I/O path name is initially assigned. For example:

```
100  ASSIGN @Device TO Dev_selector; FORMAT ON
 or
100  ASSIGN @Device TO Dev_selector ! Default for devices is FORMAT ON.
```

- Specify only the attribute(s) in a subsequent ASSIGN statement:

```
250  ASSIGN @Device; FORMAT OFF  ! Change only the attribute.
```

The result of executing this last statement is to modify the entry in the I/O path name table that describes which FORMAT attribute is currently assigned to this I/O path. The *implicit* `ASSIGN @Device TO *`, which is automatically performed when the "`TO ...`" portion is included, is *not* performed. Also, the I/O path name must currently be assigned (in this context), or an error is reported.

## Restoring the Default Attributes

If any attribute is specified, the corresponding entry in the I/O path name table is changed (as above); no other attributes are affected. However, if no attribute is assigned (as below), then *all* attributes, except WORD, are restored to their default state (such as FORMAT ON for devices.)

```
340   ASSIGN @Device  !  Restores ALL default attributes.
```

# Additional Attributes

The first section discussed the FORMAT attributes of I/O path names. Several other attributes are available to direct the BASIC system to perform the following operations whenever data are moved through the I/O path possessing the attribute:

- specify that data are to be sent and received on a byte or word basis
- perform conversions on a character-by-character basis on inbound and/or outbound data
- check for parity on inbound data, and generate parity on outbound data
- re-define the end-of-line sequence normally sent after the last data item in output operations

It is also possible to direct the system to return a numeric code to a variable which describes the outcome of an attempted ASSIGN operation. This section describes implementing these functions by using the additional I/O path attributes.

## The BYTE and WORD Attributes

The HP Series 200/300 computers are capable of handling data as either 8-bit bytes or 16-bit words when using 16-bit interfaces. This section describes how to use the BYTE and WORD attributes to determine which way the system will handle data when using these interfaces.

**Unless otherwise specified, the system treats data as bytes during I/O operations.** For instance, when the following I/O statement is executed:

```
OUTPUT Device_selector;Integer_array(*)
```

the 16-bit INTEGER values are normally sent one byte at a time, with the most significant byte of each INTEGER sent first. Executing the following statement:

```
OUTPUT Device_selector USING "W";Integer_array(*)
```

directs the system to send the data as words **if** the interface has the ability to handle data as words. With a 16-bit interface, such as the HP 98622 GPIO Interface, the INTEGER data are sent one word at a time (i.e., one word per handshake cycle). If the interface is not capable of sending one word in a single operation, the word is sent as two bytes with the most significant byte first.

When the BYTE attribute is assigned to an I/O path name, the system sends and receives all data through the I/O path as bytes; one byte is sent (or received) per operation. Thus, **BYTE directs the system to treat a 16-bit interface as if it were an 8-bit interface**. The following statements show examples of assigning the BYTE attribute to an I/O path:

```
ASSIGN @Printer TO 701; BYTE
ASSIGN @Device  TO  12; BYTE
```

In the first statement, the BYTE attribute is redundant, because the WORD attribute cannot be assigned to the HP-IB Interface (since it is an 8-bit interface).

When the I/O path name assigned to an interface possesses the BYTE attribute, the system sends and receives all subsequent data through the interface one byte per handshake operation. As an example, executing either of the following statements (when the I/O path possesses the BYTE attribute):

```
OUTPUT @Device;Integer_array(*)
OUTPUT @Device USING "W";Integer_array(*)
```

directs the system to send the data as bytes, even though the interface is capable of sending the data as words (and in the second example the "W" specifier was used). Stated again, the BYTE attribute directs the system to treat 16-bit interfaces as if they were 8-bit interfaces. With BYTE, only the 8 least significant bits of the interface are used to send and receive data; the most significant bits are always zeros. Keep in mind that the logic sense of the signal lines used to send and receive these bits is determined by switch settings on the interface card.

The **WORD attribute** specifies that all data sent and received through the I/O path are to be moved as words. In other words, this attribute **directs the system to use all 16 data lines of a 16-bit interface for all subsequent I/O operations** that use the I/O path name. This attribute is designed to improve performance in two types of situations (on 16-bit interfaces): when sending and receiving FORMAT OFF data, and when sending and receiving INTEGERs with FORMAT ON. The WORD attribute can also be used under other situations; however, results may show some unexpected "side effects," which are explained later in this section. The interface to which the I/O path name is assigned must be capable of handling data words; if not, an error will be reported when the ASSIGN is executed.

When an I/O path possesses the WORD attribute, an even number of data bytes will always be sent or received by any one I/O statement that uses the I/O path. Consequently, when an operation involves an odd number of data bytes, the system will place pad byte(s) in outbound data or enter (but ignore) additional byte(s) of inbound data. These operations can be thought of as "aligning data on word boundaries." This is the main side effect that can occur with the WORD attribute.

With the FORMAT OFF attribute, all data items are represented by an even number of bytes (see the discussion in "The FORMAT OFF Attributes" earlier in this chapter for details). Since these representations use an even number of bytes, no pad bytes are necessary.

When WORD is used with FORMAT ON, the data will be buffered (automatically by the system) when necessary to allow sending all data as words. Sending INTEGERs does not usually require this type of buffering, because each INTEGER consists of two bytes of data. However, sending strings of odd length often requires that the system perform this automatic buffering. The first byte of each word is placed in a two-character buffer (created by the system); when the second byte is placed in this buffer, the two bytes are sent as one word, with the most significant eight bits representing the first byte. If an odd number of data bytes would otherwise be sent, a Null character, CHR$(0), is placed in the buffer to "flush" the last byte.

The following statements show assigning the WORD attribute and using the I/O path to send data through the GPIO Interface at select code 12. Remember that the default FORMAT attribute assigned to I/O paths to devices is FORMAT ON.

```
110  ASSIGN @Gpio TO 12;WORD
120  OUTPUT @Gpio;"Odd"
130  OUTPUT @Gpio USING "K,L,K";"Odd","Even"
```

The following diagrams show the characters that would be sent by the OUTPUT statements in lines 120 and 130, respectively.

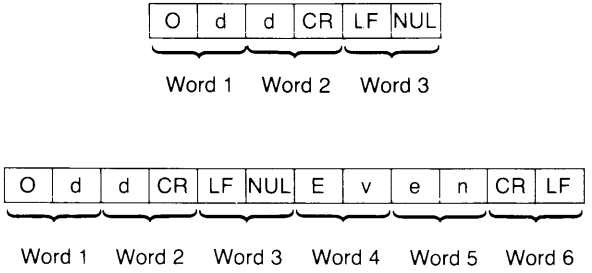



**Figure 8-3. Characters Sent by OUTPUT Statements Shown Above**

In the first statement, a Null was sent after the EOL characters to flush the buffer and force word alignment for a subsequent OUTPUT. The second statement shows that a pad byte will be sent after any EOL sequence when required to achieve word alignment; the Null pad byte was not needed after the second EOL sequence. In addition, if a buffer or file pointer currently has an odd value, a leading pad byte will be output to force word alignment before any data are sent by the OUTPUT statement.

When executing an ENTER statement from an I/O path with the WORD attribute, the system always reads an even number of bytes from the source device, since data are sent as words. In cases where an odd number of data bytes are sent, such as when an odd number of string characters are sent with an even number of statement-terminator characters, the system enters (but ignores) the last byte sent (after the statement-terminator characters). The following statements show an example of entering the data sent by the OUTPUT statements in the preceding example.

```
ASSIGN @Device TO 12;WORD
ENTER @Device;String_var1$
ENTER @Device;String_var2$
ENTER @Device;String_var3$
```

The variables receive the following values:

```
String_var1$="Odd"
String_var2$="Odd"
String_var3$="Even"
```

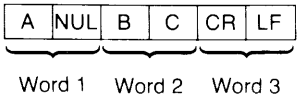
Notice that three ENTER statements were used to enter the data sent by the two preceding OUTPUT statements. This method was used to handle the pad bytes generated by the OUTPUT statement. If two ENTER statements would have been used, the pad byte sent after the second "Odd" and EOL sequence would have to have been skipped by an "X" image specifier. The following ENTER statements show how this could be done.

```
ENTER @Device USING "K,X,K";String_var1$,String_var2$
ENTER @Device USING "K";String_var3$
```

If the "X" specifier would not have been used, a pad byte would have been placed in `String_var2$`. Thus, a **general recommendation** for entering data OUTPUT through an I/O path with the WORD and FORMAT ON attributes is to enter only one item per ENTER statement.

When the WORD attribute is in effect, the "W" image specifier sends data that are always aligned on word boundaries. For instance, the following statement shows how the system defines "W" with the WORD attribute during OUTPUT.

```
OUTPUT @Device USING "B,W";65,256*66+67
```




The Null (NUL) pad byte was sent before the "W" image data to align the INTEGER specified by the "W" on a word boundary.

During ENTER, a pad byte is entered (but ignored) when necessary to align the "W" item on a word boundary. For instance, the following statement would enter the preceding data items in the same manner as they were sent.

```
ENTER @Device USING "B,W";One_byte,One_word
```

Keep in mind that these examples have been provided only to show potential problems that can arise when sending an odd number of data bytes while using the WORD attribute. It would be more appropriate to use only images that send an even number of bytes when using WORD during OUTPUT, and it will simplify matters to send only one item per OUTPUT statement. Similarly, it is generally much simpler if only one item is entered per ENTER statement.

Furthermore, if pad bytes pose a problem when working with INTEGER data (with FORMAT ON), you can also use the "Y" specifier. During OUTPUT, the "Y" does not force word alignment by sending a pad byte; during ENTER, the "Y" does not skip a byte to achieve word alignment.

Note also that the Null character pad byte may be converted to another character by using the CONVERT attribute; see the next section for further details.

The BYTE and WORD attributes affect any ENTER, OUTPUT, or TRANSFER statements that use the I/O path name. However, only the attribute specified on the non-buffer I/O path end of the TRANSFER is used; BYTE or WORD is ignored on the buffer end.

Unlike other attributes, the BYTE and WORD attribute cannot be changed once assigned to an I/O path name. For instance, executing:

```
ASSIGN @Printer TO 12
```

implicitly assigns the BYTE attribute to @Printer, since it is the default attribute. Executing the following statement results in error 600 (**`Attribute cannot be modified`**):

```
ASSIGN @Printer;WORD
```

The converse situation is true for the WORD attribute. Furthermore, if WORD has been assigned to the I/O path, then BYTE is not restored when **`ASSIGN @Device`** is executed; all other default attributes would be restored. For instance, executing:

```
ASSIGN @Device TO 12;WORD,FORMAT OFF
```

assigns the specified non-default attributes to the I/O path name @Device. Executing:

```
ASSIGN @Device
```

restores the default attribute of FORMAT ON (and also other default attributes, if currently non-default), but it **does not** restore the default BYTE attribute.

## Converting Characters

The CONVERT attribute is used to specify a character-conversion table which is to be used for OUTPUT or ENTER operations. If data are to be converted in both directions, a separate conversion table must be defined for each direction. Two conversion methods are available—by index and by pairs. This section shows simple examples of each.

CONVERT...BY INDEX specifies that each original character's code is used to index a replacement character in the specified conversion string. For instance, CHR$(10) is replaced by the 10th character in the conversion string. The only exception is that CHR$(0) will be replaced by the 256th character in the conversion string. If the string contains less than 256 characters, characters with codes that do not index a conversion-string character will not be converted. If the string contains more than 256 characters, error 18 is reported.

The following program shows an example of setting up a conversion by index for OUTPUT operations.

```
100   DIM Conv_string$[256]
110   INTEGER Index_val
120   !
130   ! Generate conversion string.
140   FOR Index_val=1 TO 255
150     SELECT Index_val
160     CASE NUM("a") TO NUM("z")    ! Change to uppercase.
170       Conv_string$[Index_val]=UPC$(CHR$(Index_val))
180     CASE ELSE  ! No conversion.
190       Conv_string$[Index_val]=CHR$(Index_val)
200     END SELECT
210   NEXT Index_val
220   Conv_string$[256]=CHR$(0) ! 256th element has an
230                             ! effective index of 0.
240                             !
250   ! Set up conversions.
260   ASSIGN @Device TO 1;CONVERT OUT BY INDEX Conv_string$
270   !
280   OUTPUT @Device;"UPPERCASE LETTERS ARE NOT CONVERTED."
290   OUTPUT @Device;"Lowercase letters are converted."
300   OUTPUT 1;"Conversions are made only "
310   OUTPUT 1;"when the I/O path is used."
320   !
330   END
```

The program is designed to convert lowercase characters to uppercase characters. In order to make the conversion, the program first computes the characters in the conversion string; the characters are computed one at a time. If the character's original code is not in the range 97 to 122 ("a" to "z"), then no change is made. If it is in the range, an uppercase character is placed in the string at the location indexed by the original (lowercase character's) code.

The example program's output is as follows.

```
UPPERCASE LETTERS ARE NOT CONVERTED.
LOWERCASE LETTERS ARE CONVERTED.
Conversions are made only
when the I/O path is used.
```

To perform the lowercase-to-uppercase conversion, it was not necessary to include characters with codes 123 through 255 in the conversion string, since these characters are not to be converted. They were included to emphasize that the 256th character must be included in the string if CHR$(0) is to be converted with this method. The CONVERT

attribute is then assigned to the I/O path, and all subsequent data sent through the I/O path (while CONVERT is in effect) will be converted.

CONVERT...BY PAIRS specifies that the conversion string contains pairs of characters, each pair consisting of an original character followed by its replacement character. Before each character is moved through the interface, the original characters in the conversion string (the odd characters) are searched for the character's occurrence. If the character is found, it will be replaced by the succeeding character in the conversion string; if it is not found, no conversion takes place. If duplicate original characters exist in the conversion string, only the first occurrence is used. The string variable must contain an even number of characters; if not, error 18 is reported.

The following program shows an example of setting up the same conversion as in the preceding example, except that conversion by pairs is used.

```
100   DIM Conv_string$[512]
110   !
120   ! Define conversion string.
130   Conv_string$="aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpP"
140   Conv_string$=Conv_string$&"qQrRsStTuUvVwWxXyYzZ"
150   !
160   ! Set up conversions.
170   ASSIGN @Device TO 1;CONVERT OUT BY PAIRS Conv_string$
180   !
190   OUTPUT @Device;"UPPERCASE LETTERS ARE NOT CONVERTED."
200   OUTPUT @Device;"Lowercase letters are converted."
210   OUTPUT 1;"Conversions are made only "
220   OUTPUT 1;"when the I/O path is used."
230   !
240   END
```

The pairs method only requires that each character to be replaced (and its replacement) is included in the conversion string. Note that the first character of each pair is the original character and the second is the replacement. If a character does not appear in the conversion string, it will not be converted.

Conversion of inbound characters can also be performed with both of these methods. In the second example, for instance, the conversion is implemented with the following statement.

```
ASSIGN @Device;CONVERT IN BY PAIRS Conv_string$
```

Conversions in both directions will continue until disabled. The following statement could be used to disable conversions of outbound data.

```
ASSIGN @Device;CONVERT OUT OFF
```

It is important to note that the conversion string specified in the ASSIGN statement is used for each OUTPUT or ENTER statement that uses the I/O path while the conversion is enabled. Note that the conversion string's contents are not contained in the I/O path data type; only a pointer to the string variable is maintained. Thus, any changes to the string's value will immediately affect any subsequent OUTPUT or ENTER that uses that I/O path.

It is also important to note that the string must be defined for at least as long as the I/O path which references it; this "lifetime" requirement has several implications. If the I/O path and conversion string are defined in different COM blocks, an error will be reported. If the I/O path is to be used as a formal parameter in a subprogram, the conversion string variable must either appear in the same formal parameter list or be defined in a COM block accessible to that subprogram. If the I/O path name is passed to subprogram(s) by including it as a pass parameter, the string variable must currently be defined in the context which defined the I/O path.

When CONVERT OUT is in effect, the specified conversions are made after any end-of-line (EOL) sequence has been inserted into the data, but before parity generation is performed (with the PARITY attribute). When CONVERT IN is in effect, conversions are made after parity is checked (if enabled), but before the data are checked for any item- or statement-termination characters.

Keep in mind that no non-default CONVERT attribute can be assigned to an I/O path that currently possesses the FORMAT OFF attribute, and vice versa.

## Changing the EOL Sequence

An end-of-line (EOL) sequence is normally sent following the last item sent with free-field OUTPUT statements and when the "L" specifier is used in an OUTPUT that uses an image. The default EOL characters are carriage-return and line-feed (CR/LF), sent with no device-dependent END indication. With AP2.0, it is also possible to define your own special EOL sequences that include sending special characters, sending an END indication, and delaying a specified amount of time after sending the last EOL character.

In order to define non-default EOL sequences to be sent by the OUTPUT statement, an I/O path must be used. The EOL sequence is specified in one of the ASSIGN statements which describe the I/O path. An example is as follows.

```
ASSIGN @Device TO 12;EOL CHR$(10)&CHR$(10)&CHR$(13)
```

The characters following the secondary keyword **EOL** are the EOL characters. Any character in the range CHR$(0) through CHR$(255) may be included in the string expression that defines the EOL characters; however, the length of the sequence is limited to eight characters or less. The characters are put into the output data before any conversion is performed (if CONVERT OUT is in effect).

If END is included in the EOL attribute, an interface-dependent "END" indication is sent with (or after) the last character of the EOL sequence. However, if no EOL sequence is sent, the END indication is also suppressed. The following statement shows an example of defining the EOL sequence to include an END indication.

```
ASSIGN @Device TO 20;EOL CHR$(13)&CHR$(10) END
```

With the HP-IB Interface, the END indication is an End-or-Identify message (EOI) sent with the last EOL character. The individual chapter that describes programming each interface further describes each interface's END indication (if implemented).

If DELAY is included, the system delays the specified number of seconds (after sending the last EOL character and/or END indication) before executing any subsequent BASIC statement.

```
ASSIGN @Device;EOL CHR$(13)&CHR$(10) DELAY 0.1
```

This parameter is useful when using slower devices which the computer can "overrun" if data are sent as rapidly as the computer can send them. For example, a printer connected to the computer through a serial interface set to operate at 300 baud might require a delay after receiving a CR character to allow the carriage to return before sending further characters. Note that the DELAY parameter is not exact: it specifies the minimum amount of delay.

The default EOL sequence is a CR and LF sent with no end indication and no delay: this default can be restored by using the EOL OFF attribute.

## Parity Generation and Checking

Parity is an indication used to help determine whether or not a quantity of data has been communicated without error. The sending device generates the parity indication, which is then checked against the parity expected by the receiving device. If the two indications don't agree, a parity error is reported.

With this system, parity may be indicated by the most significant bit of a data byte. The parity bit is generated (during OUTPUT) or checked (during ENTER) by the system according to the current PARITY attribute in effect for the I/O path through which the data bytes are being sent or received.

Unless otherwise specified, the system will not generate or check parity (the default mode is PARITY OFF). The following optional PARITY attributes are available:

**Table 8-1. Optional PARITY Attributes**

| Option | Effect During ENTER | Effect During OUTPUT |
|---|---|---|
| OFF | No check is performed | No parity is generated |
| EVEN | Check for even parity | Generate even parity |
| ODD | Check for odd parity | Generate odd parity |
| ONE | Check for parity bit set (1) | |
| | Set parity bit (1) | |
| ZERO | Check for parity bit clear (0) | Clear parity bit (0) |

If PARITY EVEN is specified, the parity bit will be a 1 when required to make the total number of 1's in the byte an even number; for instance, a byte with a value of 1 will have the parity bit set to 1 with even parity. Conversely, PARITY ODD specifies that the parity bit will be a 1 when required to make the total number of 1's odd. PARITY ONE specifies that the parity bit will always be 1, while PARITY ZERO specifies that it will always be 0. PARITY OFF disables parity generation and checking, if currently enabled for the I/O path.

To enable parity generation during OUTPUT and ENTER operations, assign a PARITY option to an I/O path. For example:

```
ASSIGN @Serial TO 9;PARITY ODD
```

specifies that all data sent through the I/O path @Serial will use the most significant bit of each byte for parity. However, only 128 different characters will be available, since one bit of the eight is not available for data representation.

If the system detects a parity error while executing an ENTER statement, error 152 (`Parity error`) will be reported. All characters entered up to (but not including) the erroneous byte will be assigned to the appropriate variable, after which the system will report the error.

If the receiving device detects a parity error, it will be responsible for communicating the error to the computer. A typical means would be to enable the interface to signal the error by generating an interrupt. See the chapters that describe interrupts in general and interrupts for the specific interface.

Parity is generated after conversions have been made during OUTPUT and is checked before conversions during ENTER. After parity is checked on inbound data, the parity bit is cleared; however, when PARITY OFF is in effect, bit 7 is not affected.

Disabling parity generation and checking is accomplished by assigning the PARITY OFF attribute to the I/O path.

```
ASSIGN @Serial;PARITY OFF
```

Parity is also disabled when an I/O path name is explicitly closed and then re-assigned, when an I/O path name is re-assigned without being closed, and when the default attributes are restored with statements such as ASSIGN @Serial.

Keep in mind that a non-default PARITY attribute cannot be assigned to an I/O path that currently possesses the FORMAT OFF attribute, and vice versa.

## Determining the Outcome of ASSIGN Statements

Although RETURN is not an attribute, including it in the list of attributes directs the system to place a a numeric code that indicates the outcome of the ASSIGN operation into the specified numeric variable. The following statement shows an example of enabling this error check:

```
ASSIGN @Device TO 12;RETURN Outcome
```

- If the operation is successful, a 0 is returned.
- If a non-zero value is returned, it is the error number which otherwise would have been reported. For instance, if an interface was not present at select code 12, the system would have placed a value of 163 in `Outcome`. This value is the error code for `I/O interface not present`.

The following statement shows a method of determining the Open/Closed status of the I/O path.

```
ASSIGN @Device;RETURN Closed_status
```

If @Device is currently Open, then 0 is returned; if it is Closed, then 177 is returned (`Undefined I/O path name`). When RETURN is used in this manner, the default attributes are not restored.

When RETURN is used in this manner, ON ERROR is normally disabled during the ASSIGN statement; however, there are certain errors which cannot be trapped by using RETURN in the ASSIGN statement.

If more than one error occurred during the ASSIGN, there is no assurance that the error number returned is either the first or the last error.

## Concepts of Unified I/O

This BASIC language system and hardware provide the ability to communicate with the several system resources with the OUTPUT and ENTER statements.

- The "Display Interfaces" and "Keyboard Interfaces" chapters describe how to communicate with the operator (through the CRT and keyboard) by using these I/O statements.
- The next section of this chapter describes how data can be moved to and from string variables with OUTPUT and ENTER statements.
- The "Advanced Transfer Techniques" chapter describes how to use OUTPUT and ENTER with buffers, which can also be used to communicate with several system resources.
- The "HP-IB Interface" chapter describes how these I/O statements are used to communicate with HP-IB peripheral devices.
- And, if you have read about mass storage operations (in the "Data Storage and Retrieval" chapter of *BASIC Programming Techniques*), you know that the ENTER and OUTPUT statements are also used to move data between the computer and mass storage files.

This ability to move data between the computer and all of its resources with the same statements is a very powerful capability of the computer's BASIC language.

Before briefly discussing I/O paths to mass storage files, the following discussion will present some background information that will help you understand the rationale behind implementing the two data representations used by the computer. The remainder of this chapter then presents several uses of this language structure.

## Data-Representation Design Criteria

As you know, the computer supports two general data representations — the ASCII and the internal representations. This discussion presents the rationale of their design.

The data representations used by the computer were chosen according to the following criteria.

- to maximize the rate at which computations can be made
- to maximize the rate at which the computer can move the data between its resources
- to minimize the amount of storage space required to store a given amount of data
- to be compatible with the data representation used by the resources with which the computer is to communicate

The **internal representations** implemented in the computer are designed according to the **first three of the above criteria**. However, the last criterion must always be met if communication is to be achieved. If the resource uses the ASCII representation, this compatibility requirement takes precedence over the other design criteria. The **ASCII representation** fulfills this **last criterion** for most devices and for the computer operator. The first three criteria are further discussed in the following description of data representations used for mass storage files.

## I/O Paths to Files

There are three types of *data files*: ASCII, BDAT, and HPUX.

- Only the ASCII data representation is used with ASCII files.
- But either the ASCII (FORMAT ON) or the internal (FORMAT OFF) representation can be used with BDAT and HPUX files.

I/O paths to files are briefly described in this section to further justify the internal data representations implemented with this system, and to preface the applications presented in the last section of this chapter.

## BDAT Files

BDAT (BASIC Data) and HPUX files[1] have been designed with the first three of the preceding design criteria in mind. *Both numeric and string computations are much faster.* These internal data representations *generally allow much more data to be stored on a disc* because there is no storage overhead (for numeric items); that is, there are no "record headers" for numeric items.

The *transfer rates* for each data type has also been *increased.* Numeric output operations are always much faster because there is no time required for "formatting". Numeric enter operations are also faster because the system does not have to search for item- and statement-termination conditions.

In addition, I/O paths to BDAT and HPUX files can use either the ASCII (FORMAT ON) or the internal (FORMAT OFF) representation.

The following program shows a few of the features of BDAT files. The program first outputs an internal-form string (with FORMAT ON), and then enters the length header and string characters with FORMAT OFF.

```
100   OPTION BASE 1
110   DIM Length$[4],Data$[256],Int_form$[256]
120   !
130   ! Create a BDAT file (1 record; 256 bytes/record.)
140   ON ERROR GOTO Already_created
150   CREATE BDAT "B_file",1
160 Already_created: OFF ERROR
170   !
180   ! Use FORMAT ON during output.
190   ASSIGN @Io_path TO "B_file";FORMAT ON
200   !
210   Length$=CHR$(0)&CHR$(0)  ! Create length header.
220   Length$=Length$&CHR$(0)&CHR$(252)
230   !
```

[1] Examples of HPUX files are shown in the "Porting and Sharing Files" chapter of *BASIC Programming Techniques.*

```
240   ! Generate 256-character string.
250   Data$="01234567"
260   FOR Doubling=1 TO 5
270      Data$=Data$&Data$
280   NEXT Doubling
290   ! Use only 1st 252 characters.
300   Data$=Data$[1,252]
310   !
320   ! Generate internal-form and output.
330   Int_form$=Length$&Data$
340   OUTPUT @Io_path;Int_form$;
350   ASSIGN @Io_path TO *
360   !
370   ! Use FORMAT OFF during enter (default).
380   ASSIGN @Io_path TO "B_file"
390   !
400   ! Enter and print data and # of characters.
410   ENTER Data$
420   PRINT LEN(Data$);"characters entered."
430   PRINT
440   PRINT Data$
450   ASSIGN @Io_path TO * ! Close I/O path.
460   !
470   END
```

### ASCII Files

ASCII files are designed for interchangeability with other HP computer systems. This interchangeability imposes the restriction that the data must be represented with ASCII characters. Each data item sent to these files is a special case of FORMAT ON representation; *each item is preceded by a two-byte length header* (analogous to the internal form of string data). In order to maintain this compatibility, there are two additional restrictions placed on ASCII files:

- The FORMAT OFF attribute *cannot* be assigned to an ASCII file
- You cannot use OUTPUT..USING or ENTER..USING with an ASCII file.

The following program shows the I/O path name @Io_path being assigned to the ASCII file named ASC_FILE. Notice that the file name is in all uppercase letters; this is also a compatibility requirement when using this file with some other systems.

The program creates an ASCII file, and then outputs program lines to the file. The program then gets and runs this newly created program. (If you type in and run this program, be sure to save it on disc, because running the program will load the program it creates, destroying itself in the process.)

```
100   DIM Line$(1:3)[100]  ! Array to store program.
110   !
120   ! Create if not already on disc.
130   ON ERROR GOTO Already_exists
140   CREATE ASCII "ASC_FILE",1  ! 1 record.
150 Already_exists:  OFF ERROR
160   !
170   ASSIGN @Io_path TO "ASC_FILE"
180   STATUS @Io_path,6;Pointer
190   PRINT "Initially:    file pointer=";Pointer
200   PRINT
210   !
220   Line$(1)="100  PRINT ""New program.""  "
230   Line$(2)="110  BEEP"
240   Line$(3)="120  END"
250   !
260   OUTPUT @Io_path;Line$(*)
270   STATUS @Io_path,6;Pointer
280   PRINT "After OUTPUT: file pointer=";Pointer
290   PRINT
300   !
310   GET "ASC_FILE" ! Implicitly closes I/O path.
320   !
330   END
```

## Data Representation Summary

The following table summarizes the control that programs have on the FORMAT attribute assigned to I/O paths.

**Table 8-2. Program Control of the FORMAT Attribute**

| Type of Resource | Default FORMAT Attribute Used | Can Default FORMAT Attribute Be Changed? |
|---|---|---|
| Devices | FORMAT ON | Yes (if an I/O path is used)[1] |
| BDAT files | FORMAT OFF | Yes |
| HPUX files | FORMAT OFF | Yes |
| ASCII files | FORMAT ON[2] | No |
| String variables | FORMAT ON | No |
| Buffers | FORMAT ON | Yes |

[1] FORMAT ON is *always* used whenever an OUTPUT..USING or ENTER..USING statement is used, regardless of the FORMAT attribute assigned to the I/O path.

[2] The data representation used with ASCII files is a special case of the FORMAT ON representation.

# Applications of Unified I/O

This section describes two uses of the powerful unified-I/O scheme of the computer. The first application contains further details and uses of I/O operations with string variables. The second application involves using a disc file to simulate a device.

## I/O Operations with String Variables

Chapter 3 briefly described how string variables may be specified as the source or destination of data in I/O statements, but it described neither the details nor many uses of these operations. This section describes both the details of and several uses of outputting data to and entering data from string variables.

### Outputting Data to String Variables

When a string variable is specified as the destination of data in an OUTPUT statement, source items are evaluated individually and placed into the variable according to the free-field rules or the specified image, depending on which type of OUTPUT statement is used. Thus, item terminators may or may not be placed into the variable. The ASCII data representation is always used during outputs to string variables; in fact, **data output to string variables is exactly like that sent to devices through I/O paths with the FORMAT ON attribute.**

Characters are always placed into the variable beginning at the first position; no other position can be specified as the beginning position at which data will be placed. Thus, **random access of the information in string variables is not allowed** from OUTPUT and ENTER statements; all data must be accessed serially. For instance, if the characters "1234" are output to a string variable by one OUTPUT statement, and a subsequent OUTPUT statement outputs the characters "5678" to the same variable, the second output **does not** begin where the first one left off (i.e., at string position five). The second OUTPUT statement begins placing characters in position one, just as the first OUTPUT statement did, overwriting the data initially output to the variable by the first OUTPUT statement.

The string variable's length header (4 bytes) is updated and compared to the dimensioned length of the string as characters are output to the variable. If the string is filled before all items have been output, an error is reported; however, the string contains the first *n* characters output (where *n* is the dimensioned length of the string).

### Example

The following program outputs string and numeric data items to a string variable and then calls a subprogram which displays each character, its decimal code, and its position within the variable.

```
100   ASSIGN @Crt TO 1  ! CRT is disp. device.
110   !
120   OUTPUT Str_var$;12,"AB",34
130   !
140   CALL Read_string(@Crt,Str_var$)
150   !
160   END
170   !
180   !

190 SUB Read_string(@Disp,Str_var$)
200     !
210     ! Table heading.
220     OUTPUT @Disp;"---------------------"
230     OUTPUT @Disp;"Character  Code  Pos."
240     OUTPUT @Disp;"---------  ----  ----"
250     Dsp_img$="2X,4A,5X,3D,2X,3D"
260     !
270     ! Now read the string's contents.
280   FOR Str_pos=1 TO LEN(Str_var$)
290         Code=NUM(Str_var$[Str_pos;1])
300       IF Code<32 THEN ! Don't disp. CTRL chars.
310           Char$="CTRL"
320       ELSE
330           Char$=Str_var$[Str_pos;1] ! Disp. char.
340       END IF
350       !
360       OUTPUT @Disp USING Dsp_img$;Char$,Code,Str_pos
370   NEXT Str_pos
380   !
390   ! Finish table.
400   OUTPUT @Disp;"---------------------"
410   OUTPUT @Disp ! Blank line.
420   !
430   SUBEND
```

```
--------------------
Character  Code  Pos.
---------  ----  ----
32     1
1          49     2
2          50     3
,          44     4
A          65     5
B          66     6
CTRL       13     7
CTRL       10     8
           32     9
3          51    10
4          52    11
CTRL       13    12
CTRL       10    13
--------------------
```

**Figure 8-4. Final Display**

Outputting data to a string and then examining the string's contents is usually a more convenient method of examining output data streams than using a mass storage file. The preceding subprogram may facilitate the search for control characters, because they are not actually displayed, which could otherwise interfere with examining the data stream.

### Example

The following example program shows how outputs to string variables can be used to reduce the overhead required in ASCII data files. The first method of outputting data to the file requires as much media space for overhead as for data storage, due to the two-byte length header that precedes each item sent to an ASCII file. The second method uses more computer memory, but uses only about half of the storage-media space required by the first method. The second method is also **the only way to format data sent to ASCII data files.**

```
100   PRINTER IS 1
110   !
120   ! Create a file 1 record long (=256 bytes).
130   ON ERROR GOTO File_exists
140   CREATE ASCII "TABLE",1
150 File_exists:   OFF ERROR
160               !
170               !
180   ! First method outputs 64 items individually..
190   ASSIGN @Ascii TO "TABLE"
200   FOR Item=1 TO 64  ! Store 64 2-byte items.
210       OUTPUT @Ascii;CHR$(Item+31)&CHR$(64+RND*32)
220       STATUS @Ascii,5;Rec,Byte
230       DISP USING Image_1;Item,Rec,Byte
240   NEXT Item
```

```
250 Image_1: IMAGE "Item ",DD,"  Record ",D,"  Byte ",3D
260   DISP
270   Bytes_used=256*(Rec-1)+Byte-1
280   PRINT Bytes_used;" bytes used with 1st method."
290   PRINT
300   PRINT
310   !
320   !
330   ! Second method consolidates items.
340   DIM Array$(1:64)[2],String$[128]
350   ASSIGN @Ascii TO "TABLE"
360   !
370   FOR Item=1 TO 64
380       Array$(Item)=CHR$(Item+31)&CHR$(64+RND*32)
390   NEXT Item
400   !
410   OUTPUT String$;Array$(*); ! Consolidate.
420   OUTPUT @Ascii;String$     ! OUTPUT as 1 item.
430   !
440   STATUS @Ascii,5;Rec,Byte
450   Bytes_used=256*(Rec-1)+Byte-1
460   PRINT Bytes_used;" bytes used with 2nd method."
470   !
480   END
```

The program shows many of the features of using ASCII files and string variables. The first method of outputting the data items shows how the file pointer varies as data are sent to the file. Note that the file pointer points to the **next** file position at which a subsequent byte will be placed. In this case, it is incremented by four by every OUTPUT statement (since each item is a two-byte quantity preceded by a two-byte length header).

The program could have used a BDAT file, which would have resulted in using slightly less disc-media space; however, using BDAT files usually saves much more disc space than would be saved in this example. The program does not show that **ASCII files cannot be accessed randomly**; this is one of the major differences between using ASCII and BDAT files.

### Example

Outputs to string variables can also be used to generate the string representation of a number, rather than using the VAL$ function (or a user-defined function subprogram). The **main advantage** is that you can explicitly specify the number's image while still using only a single program line. The following program compares the string generated by the VAL$ function to that generated by outputting the number to a string variable.

```
100   X=12345678
110   !
120   PRINT VAL$(X)
130   !
140   OUTPUT Val$ USING "#,3D.E";X
150   PRINT Val$
160   !
170   END
```

### Printed Results

```
1.2345678E+7
123.E+05
```

## Entering Data From String Variables

Data are entered from string variables in much the same manner as output to the variable. All ENTER statements that use string variables as the data source interpret the data according to the FORMAT ON attribute. Data is read from the variable beginning at the first string position; if subsequent ENTER statements read characters from the variable, the read also begins at the first position. If more data are to be entered from the string than are contained in the string, an error is reported; however, all data entered into the destination variable(s) before the end of the string was encountered remain in the variable(s) after the error occurs.

When entering data from a string variable, the computer keeps track of the number of characters taken from the variable and compares it to the string length. Thus, **statement-termination** conditions are **not** required; the ENTER statement automatically terminates when the last character is read from the variable. However, **item** terminators are still required **if** the items are to be separated **and** the lengths of the items are not known. If the length of each item is known, an image can be used to separate the items.

### Example

The following program shows an example of the need for **either** item terminators **or** length of each item. The first item was not properly terminated and caused the second item to not be recognized.

```
100   OUTPUT String$;"ABC123";  ! OUTPUT w/o CR/LF.
110   !
120   ! Now enter the data.
130   ON ERROR GOTO Try_again
140   !
150 First_try: !
160   ENTER String$;Str$,Num
170   OUTPUT 1;"First try results:"
180   OUTPUT 1;"Str$= ";Str$,"Num=";Num
190   BEEP     ! Report getting this far.
200   STOP
210   !
220 Try_again: OUTPUT 1;"Error";ERRN;" on 1st try"
230            OUTPUT 1;"STR$=";Str$,"Num=";Num
240            OUTPUT 1
250            OFF ERROR  ! The next one will work.
260            !
270   ENTER String$ USING "3A,3D";Str$,Num
280   OUTPUT 1;"Second try results:"
290   OUTPUT 1;"Str$= ";Str$,"Num=";Num
300   !
310   END
```

This technique is convenient when attempting to enter an unknown amount of data or when numeric and string items within incoming data are not terminated. The data can be entered into a string variable and then searched by using images.

### Example

ENTERs from string variables can also be used to generate a number from ASCII numeric characters (a recognizable collection of decimal digits, decimal point, and exponent information), rather than using the VAL function. As with outputs to string variables, images can be used to interpret the data being entered.

```
30    Number$="Value= 43.5879E-13"
40    !
50    ENTER Number$;Value
60    PRINT "VALUE=";Value
70    END
```

## Taking a Top-Down Approach

This application shows how the computer's BASIC-language structure may help simplify using a "top-down" programming approach. In this example, a simple algorithm is first designed and then expanded into a program in a general-to-specific, stepwise manner. The top-down approach shown here begins with the general steps and works toward the specific details of each step in an orderly fashion.

One of the first things you **should** do when programming computers is to **plan the procedure before actually coding any software**. At this point of the design process, you need to have a good understanding of both the problem and the requirements of the program. The general tasks that the program is to accomplish must be described before the order of the steps can be chosen. The following simple example goes through the steps of taking this top-down approach to solving the problem.

**Problem:** write a program to monitor the temperature of an experimental oven for one hour.

*Step 1.* *Verbally describe what the program must do in the* ***most general*** *terms. You may want to make a chart or draw a picture to help visualize what is required of the program.*

Initialize the monitoring equipment. Start the timer and turn the oven on. Begin monitoring oven temperature and measure it every minute thereafter for one hour. Display the current oven temperature, and plot the temperatures vs. time on the CRT.

*Step 2.* *Verbally describe the algorithm. Again, try to keep the steps as general as possible.*

This process is often termed writing the "pseudo code". Pseudo code is merely a written description of the procedure that the computer will execute. The pseudo code can later be translated into BASIC-language code.

Setup the equipment.

Set the oven temperature and turn it on.

Initialize the timer.

Perform the following tasks every minute for one hour.

Read the oven temperature.

Display the current temperature and elapsed time.

Plot the temperature on the CRT.

Turn the oven and equipment off.

Signal that the experiment is done.

*Step 3.* *Begin translating the algorithm into a BASIC-language program.*

The following program follows the general flow of the algorithm. As you become more fluent in a computer language, you may be able to write pseudo code that will translate more directly into the language. However, avoid the temptation to write the initial algorithm in the computer language, because writing the pseudo code is a **very important** step of this design approach!

```
100   ! This program: sets up measuring equipment,
110   ! turns an oven on, and initializes a timer.
120   ! The oven's temperature is measured every
130   ! minute thereafter for one hour. The temp.
140   ! readings are displayed and plotted on the
150   ! CRT.
160   !
170   Rdgs_interval=60  ! 60 seconds between readings.
180   Test_length=60    ! Run test for 60 minutes.
190   !
200   CALL Equip_setup
210   CALL Set_temp
220   GOSUB Start_timer
230   !
240 Keep_monitoring: ! Main loop.
250                  !
260   GOSUB Timer
270   !
280   IF Seconds<=Rdgs_interval THEN
290       GOTO Keep_monitoring
300   ELSE
310       Minutes=Minutes+1
320       CALL Read_temp
330       CALL Plot_temp
340   END IF
350   !
```

```
360   !
370   IF Minutes<Test_length THEN
380       GOTO Keep_monitoring
390   ELSE
400       CALL Off_equip
410       PRINT "End of experiment"
420   END IF
430   !
440   STOP
450   !
460   !
470   ! First the subroutines.
480   !
490 Start_timer: Init_time=TIMEDATE
500              PRINT "Timer initialized."
510              PRINT
520              PRINT
530              RETURN
540              !
550 Timer: !
560        Seconds=TIMEDATE-Minutes*60-Init_time
570        DISP USING Time_image;Minutes,Seconds
580 Time_image: IMAGE "Time: ",DD," min  ",DD.D," sec"
590        RETURN
600        !
610   END
620   !
```

```
630   !
640   ! Now the subprograms.
650   !
660  SUB Equip_setup
670      PRINT "Equipment setup."
680      SUBEND
690      !
700  SUB Set_temp
710      PRINT "Oven temperature set."
720      SUBEND
730      !
740  SUB Read_temp
750      PRINT "Temp.= xx degrees F  ";
760      SUBEND
770      !
780  SUB Plot_temp
790      PRINT "(plotted)."
800      PRINT
810      SUBEND
820      !
830  SUB Off_equip
840      PRINT
850      PRINT "Equipment shut down."
860      PRINT
870      SUBEND
```

At this point, you should run the program to verify that the general program steps are being executed in the desired sequence. If not, keep refining the program flow until all steps are executed in the proper sequence. This is also a very important step of your design process: the sooner you can verify the flow of the main program the better. This approach also relieves you of having to set up and perform the actual experiment as the first test of the program.

Notice also that some of the program steps use CALLs while others use GOSUBs. The general convention used in this example is that subprograms are used only when a program step is to be expanded later. GOSUBs are used when the routine called will probably not need further refinement. As the subprograms are expanded and refined, each can be separately stored and loaded from disc files, as shown in the next step.

*Step 4.* *After the correct order of the steps has been verified, you can begin programming and verifying the details of each step (known as stepwise refinement).*

The computer features a mechanism by which the process of expanding each step can be simplified. With it, each subprogram can be expanded and refined individually and then stored separately in a disc file. This facilitates the use of the top-down approach. Each subprogram can also be tested separately, if desired.

In order to use this mechanism, first save or store the main program; for instance, execute:

```
SAVE "MAIN1"
```

Then, isolate the subprogram by deleting all other program lines in memory. In this case, executing:

```
DEL 10,650
```
and
```
DEL 700,900
```

would delete the lines which are not part of the "Equip_setup" subprogram currently in memory.

```
660   SUB Equip_setup
670       PRINT "Equipment setup."
680   SUBEND
690   !
```

At this point, two steps can be taken:

- Write the temperature-measuring device's initialization routine.
- Write a test routine that simulates the device by returning a known set of data.

The "Equip_setup" subprogram might be expanded as follows to create a disc file and fill it with a known set of temperature readings so that the program can be tested without having to write, verify, and refine the routine that will set up the temperature-measuring device. In fact, you don't even need the device at this point.

```
100  CALL Equip_setup(@Temp_meter,Temp)
110  END
120  !
130  SUB Equip_setup(@Temp_meter,Temp)
140    !
150    ! This subroutine will set up a BDAT file as
160    ! be used to simulate a temperature-measuring
170    ! device. Refine to set up the actual
180    ! equipment later.
190    !
200    ON ERROR GOTO Already
210    CREATE BDAT "Temp_rdgs",1
220    !

230    ! Output fictitious readings.
240    ASSIGN @Temp_meter TO "Temp_rdgs"
250    FOR Reading=1 TO 60
260        OUTPUT @Temp_meter;Reading+70
270    NEXT Reading
280    ASSIGN @Temp_meter TO * ! Reset pointer.
290    !
300 Already: OFF ERROR
310    !
320    ASSIGN @Temp_meter TO "Temp_rdgs"
330    !
340    PRINT "Equipment setup."
350  SUBEND
```

Notice that two pass parameters have been added to the formal parameter list. These parameters allow the main program (and and subprograms to which these parameters are passed) to access this I/O path and variable. The CALL statements in the main program must be changed accordingly before the main program can be run with these subprograms. These parameters can also be passed to the subprograms by declaring them in variable common (that is, by including the appropriate COM statements).

After the subprogram has been expanded, tested, and refined, you should store it in a file with the STORE statement (not SAVE). For instance, execute:

```
STORE "SETUP1"
```

When the main program is to be tested again, the "Equip_setup" subprogram can be loaded back into memory by executing:

```
LOADSUB ALL FROM "SETUP1"
```

Since this subprogram names an I/O path which is to be used to simulate the temperature-measuring device, the "Read_remp" subprogram can also be expanded at this point. The "Read_temp" subprogram only needs to enter a reading from the measuring device (in this case, the disc file which has been set up to simulate the temperature-measuring device.) The following program shows how this subprogram might be expanded.

```
740   SUB Read_temp (@Temp_meter,Temp)
741       ENTER @Temp_meter;Temp
750       PRINT "Temp. =";Temp;" degrees F. "
760   SUBEND
```

This subprogram can also be stored in a disc file by executing:

```
STORE "READ_T1"
```

Now that both of the expanded subprograms have been stored, the main program can be retrieved and modified as necessary. Execute:

```
LOAD "MAIN1"
```
or
```
GET  "MAIN1"
```

Add the pass parameters to the appropriate CALL statements (lines 200 and 320). Since the main program still contains the initial versions of the expanded subprograms, these two subprograms should be deleted. Executing these two statements:

```
DELSUB "Equip_setup"
```
and
```
DELSUB "Read_temp
```

will delete only these two subprograms and leave the rest of the program intact.

Now that the main program has been modified to CALL the expanded/refined subprograms, you may want to store (or save) a copy of the program on the disc. This will relieve you of the effort of deleting the old subprograms from the main program every time it is retrieved. Execute:

```
STORE "MAIN2
```
or
```
SAVE "MAIN2"
```

Now load the subprograms into memory by executing:

```
LOADSUB ALL FROM "SETUP1"
```
and
```
LOADSUB ALL FROM "READ_T1"
```

Running the program first "sets up" the device simulation and then accesses the file as it would access the actual temperature-measuring device.

## Conclusion

As you can see, this approach can be used very easily with Series 200/300 BASIC. In addition, the "Read_temp" subprogram *does not have to be revised* to access the real device. Only "Equip_setup" needs to be changed to assign the I/O path name "@Temp_meter" to the real device. This unified I/O scheme makes this system very powerful and reduces "throw away" code when using this "top down" approach.

# Table of Contents

# Advanced Transfer Techniques 9

This chapter discusses data transfer techniques available with the TRANS binary. While many applications will not need the specialized techniques presented here, these techniques aid in communicating with very slow and very fast devices.

## The Purpose of Transfers

When using OUTPUT and ENTER to communicate with peripheral devices, special problems can arise. Normally, program execution does not leave the statement until all data items are satisfied; therefore, a very slow device will keep the computer waiting between each byte or word. A great amount of time may be wasted while the computer waits for the device to be ready for the next item. Another problem exists when communicating with a very fast device. The device may attempt to send data faster than the computer can accept it. To overcome both problems, an alternate method of communication has been implemented—the TRANSFER statement.

The TRANSFER statement allows you to exchange information with a device or file through I/O paths. The most important difference between using TRANSFER and the regular methods of communication (OUTPUT and ENTER) is that a transfer can take place *concurrently* with continued program execution. Thus a transfer can be thought of as a "background" process or an "overlapped" operation. This has far-reaching consequences that affect the behavior of the BASIC system.

# Overview of Buffers and Transfers

Before any transfer takes place, an area of memory is reserved to hold the data being transferred (examples are shown on the following pages). This area of memory is called a buffer. Defining a buffer is somewhat analogous to creating a high-speed device inside the computer. Two advantages are gained by simulating a device in memory:

- The buffer is *fast* enough to accept incoming data from almost any device.
- The actual transfer operation can be handled *concurrently* with continued program execution (that is, it is a "background process" which can be "overlapped" with concurrent processing of other BASIC program lines).

## Inbound and Outbound Transfers

Every transfer will use a buffer as either its source or its destination. From the buffer's point of view, there are two types of transfers.

An *inbound* transfer moves data from a device or file into the buffer.

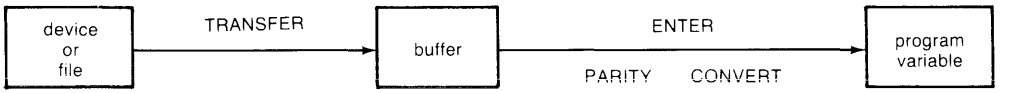



**Figure 9-1. Inbound Transfer**

An *outbound* transfer moves data from the buffer to a device or file.

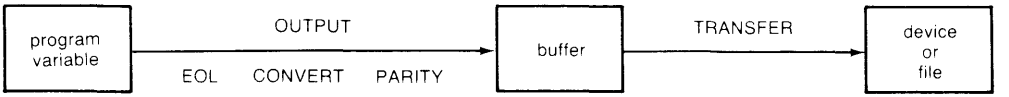



**Figure 9-2. Outbound Transfer**

Data logging is the process of combining inbound and outbound transfers.

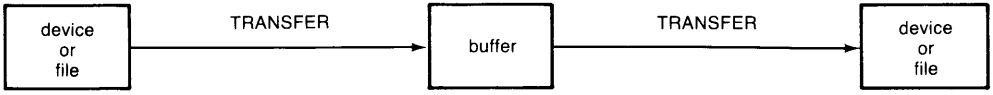



**Figure 9-3. Data Logging**

## Supported Transfer Sources and Destinations

TRANSFER operations are allowed for the following types of interfaces and files:

| Interfaces | Files |
|---|---|
| HP-IB (98624, built-in) | BDAT |
| GPIO (98622) | HPUX |
| Serial (98626, 98644, built-in) | |
| Datacomm (98628) | |

### Restrictions

A transfer **cannot** involve a CRT display, a keyboard, a BCD interface, or a DC600 cartridge-tape drive.

One and only one buffer can be specified in a TRANSFER statement. Transfers from buffer to buffer or from device to device are **not** allowed.

Transfers to and from files on volumes with 512-byte sectors (formatting option 2) is **not** allowed.

Further restrictions are listed in the "Restrictions" section of this chapters.

## Examples of Transfer

Here are two complete programs that show the steps in creating and using buffers. The following paragraphs describe the individual steps of the programs.

```
10    DIM Text$[1025] BUFFER
20    ASSIGN @Buff TO BUFFER Text$
30    ASSIGN @Print TO PRT          ! 'PRT' returns 701 for printer
40    !
50    FOR I=1 TO 25
60      OUTPUT @Buff;"How many times do I need to print this?"
70    NEXT I
80    !
90    TRANSFER @Buff TO @Print      ! Start the transfer
100   !                               Transfer continues as
110   FOR I=1 TO 450                  a "background" process.
120     PRINT TABXY(I MOD 15,0);"As many times as it takes."
130   NEXT I
140   END
```

Lines 10 and 20 create a named buffer. Line 30 assigns a printer that will be used as the destination for the transfer. The OUTPUT statement in line 60 fills the buffer with data (25 lines of 41 characters, including the CR/LF EOL sequence). Line 90 contains the TRANSFER statement that sends the data in the buffer to the printer. Running the program shows the overlapped operation of transfers. Buffered data is being printed on the printer while the program prints on the CRT.

A similar technique can be used for inbound transfers, as shown in the following example program.

```
10    DIM Text$[256] BUFFER,A$(100)[80]
20    ASSIGN @Buff TO BUFFER Text$
30    ASSIGN @Device TO 12          ! Some device at select code 12
40    !
50    TRANSFER @Device TO @Buff;CONT  ! Start the transfer
60    !
70    FOR I=1 TO 100
80      ENTER @Buff;A$(I)           ! Enter the items
90    NEXT I
100   ABORTIO @Device               ! Terminate TRANSFER
110   !
120   END
```

A named buffer is created in lines 10 and 20. A device is assigned in line 30 that will be used as the source for the transfer. The buffer is filled by the TRANSFER in line 50 and the ENTER statement in line 80 empties the buffer.

# A Closer Look at Buffers

A buffer is a section of computer memory reserved to hold the data being transferred.

## Types of Buffers

Two types of buffers can be created and assigned to I/O path names.

- A **named** buffer is a string scalar, or an INTEGER, COMPLEX, or REAL array.

```
100  DIM Num_array(1:512) BUFFER      ! Named buffer.
110  ASSIGN @Buff TO BUFFER Num_array
```

- An **unnamed** buffer is a section of memory which has no associated variable name.

```
100  ASSIGN @Buff TO BUFFER [1024]    ! Unnamed buffer.
```

A named buffer can also be accessed by its variable name (for instance, by using OUTPUT or assigning the variable). However, an unnamed buffer can only be accessed by its I/O path name.

## Creating Named Buffers

Named buffers are buffers which use variables declared in DIM, COM, COMPLEX, REAL, or INTEGER statements. Note that a buffer cannot be allocated by an ALLOCATE statement. Named buffers are declared by placing the keyword BUFFER after the variable name. For instance:

```
100  DIM A$[256],B$[256] BUFFER,C$

110  COM Block(1000),Temp(100) BUFFER,INTEGER X(10,10) BUFFER,Y,Z

120  REAL Fools_buff(1000), Real_buff(10) BUFFER, No_buff(10)
```

Only the variable name immediately preceding the keyword BUFFER becomes a buffer. In the first example statement, B$ is a buffer while A$ and C$ are not buffers. Declaring a variable as a buffer does not prevent it from being used in its normal manner, but care must be taken not to corrupt the information in the buffer.

## Assigning I/O Path Names to Named Buffers

Once a named buffer has been declared, an I/O path name can be assigned to it by an ASSIGN statement. For instance:

```
ASSIGN @Path TO BUFFER B$

ASSIGN @Buff TO BUFFER X(*)

ASSIGN @Buffer TO BUFFER Real_buff(*)
```

The I/O path name can now be used to access the buffer. The keyword BUFFER must appear in both the variable declaration statement and the ASSIGN statement for named buffers.

## Assigning I/O Path Names to Unnamed Buffers

Unnamed buffers are created in ASSIGN statements and can only be accessed by their I/O path names. The following statement shows a typical unnamed buffer assignment.

```
ASSIGN @Buff to BUFFER [65536]
```

The value in brackets indicates the number of bytes of memory to be reserved for the buffer. This allows a buffer to be larger than the maximum length of 32 767 bytes for a string variable. Named buffers using REAL, COMPLEX, and INTEGER arrays can also be larger than 32 767 bytes.

Using unnamed buffers ensures data integrity since the buffer cannot be accessed by a variable name. Closing an I/O path assigned to an unnamed buffer (`ASSIGN @Path TO *`) releases the memory reserved for the buffer. This is similar to the behavior of allocated variables.

## Buffer-Type Registers

Assigning an I/O path name to a buffer creates a control table. This control table defines STATUS and CONTROL registers which can monitor and interact with the operation of the buffer.

All I/O path names, including I/O path names assigned to buffers, use register 0 to indicate the path type.

**STATUS Register 0**
0 = Invalid I/O path name
1 = I/O path assigned to a device
2 = I/O path assigned to a data file
3 = I/O path assigned to a buffer

Register 0 returns a 3 when the I/O path is associated with a buffer. Register 1 indicates whether the buffer is named or unnamed.

**STATUS Register 1** Buffer type (1=named, 2=unnamed)

## Buffer Size Register

Once a buffer has been assigned an I/O path name, Status register 2 returns the buffer's capacity (maximum size, in bytes).

**STATUS Register 2** Buffer size in bytes

When I/O path names are assigned to buffers, the buffer must exist as long as the I/O path name is valid. Consider the example of a buffer created locally in a context and then assigned an I/O path name declared in COM. When execution leaves the local context, the I/O path name would still be valid but the buffer would no longer exist. If this happens, an error is reported:

```
ERROR 602 Improper BUFFER lifetime.
```

This error also occurs if the buffer and the I/O path name being assigned are in different COM areas.

## Buffer Pointers

In order to understand I/O involving buffers, it is essential to understand how a buffer is set up and maintained.

When an ASSIGN statement associates an I/O path name with a buffer, it also creates and initializes a buffer control table. Among the entries in the **control table** are two pointers and a counter which are used to monitor and control all data transfer to and from the buffer through the I/O path.

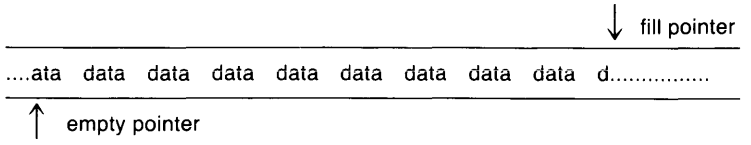
- The buffer **fill pointer** points to the next byte of the buffer which can accept data.
- The **empty pointer** points to the next byte of data which can be read from the buffer.
- The **byte count** shows the number of bytes currently in the buffer (usually equal to fill pointer – empty pointer).

The current values of the pointers can be checked by using the STATUS statement with the following registers.

| | |
|---|---|
| **STATUS Register 3** | Current fill pointer |
| **STATUS Register 4** | Current number of bytes in buffer |
| **STATUS Register 5** | Current empty pointer |

As data is written into the buffer (OUTPUT or TRANSFER), the fill pointer is advanced as necessary to point to the next available byte of buffer storage, and the counter is incremented by the number of bytes added to the buffer.

*(inbound)* `TRANSFER @Device TO @Buffer`




*(outbound)* `TRANSFER @Buffer to @File`

Similarly, when data is read from the buffer (ENTER or TRANSFER), the empty pointer is advanced to point to the first unread byte, and the counter is decremented by the number of bytes which have been read.

It is also important to realize that the buffers used with the TRANSFER statement are **circular**. This means that when the last byte of buffer storage has been accessed, the system will wrap around and access the first byte of buffer storage. The only thing which prevents writing more data into the buffer is the byte count (Register 4) to become equal to the buffer capacity (Register 2). Similarly, once the system has read the data from the last byte of buffer storage, it will next read from the first byte, but reading must cease when the byte count reaches zero.

Both full and empty buffers have the fill pointer and the empty pointer referencing the same byte of buffer storage. The system distinguishes between full and empty by examining the byte count. If it is zero, the buffer is empty. If it is equal to the buffer's capacity, the buffer is full.

It is impossible to perform any operation which would cause the byte count to take on a value less than zero or greater than the buffer capacity. Attempting to OUTPUT more data into a full buffer or ENTER data from an empty buffer produces:

```
ERROR 59 End of file or buffer found
```

Since fill and empty pointers are updated independently of each other and a TRANSFER can execute concurrently with other statements, it is possible for one TRANSFER to be putting data into the buffer while another TRANSFER is removing data.

The amount of data which can be moved by a single transfer operation is not limited by the buffer's capacity. When two TRANSFER statements involving the same buffer are of comparable speed and execute concurrently, the buffer's fill and empty pointers may never reach the empty or full state. If the two TRANSFER statements execute at different speeds because of the transfer mode which must be used or because of the throughput capacity of the devices involved, it is still possible to keep two TRANSFER statements running concurrently by specifying the CONT parameter on both (discussed in subsequent sections). CONT directs a transfer not to terminate when the buffer becomes full or empty. Instead, the transfer "goes to sleep" until the buffer is again ready for the transfer process to continue.

### Accessing Named Buffers via Variable Names

If you plan to transfer data through a buffer without using the I/O path name (such as by using the string varible's name or numeric array variable's name), it will be necessary to change the values of the pointers. CONTROL registers 3, 4, and 5 control the positioning of the pointers.

If either the fill or empty pointer is changed the appropriate pointer is modified and no other action is taken. Assuming no active transfer, if the byte count is changed, the empty pointer is set to zero and the fill pointer is set to correspond to the length specified. If a transfer is active in both directions, you cannot change the byte count or either pointer. If an inbound transfer is active, the empty pointer will be adjusted to set the byte count as specified. Similarly, if an outbound transfer is active, the fill pointer will be adjusted to match the byte count specified.

When the byte count is set along with either the fill or empty pointer, the pointer is moved to the position specified and the remaining pointer is adjusted to correspond to the specified length.

If all three pointers are changed, they must be a consistent set to prevent the following error:

```
ERROR 19 Improper value or out of range
```

If both fill and empty pointers are set to the same value, the length must be either zero (buffer empty) or the maximum buffer length (buffer full).

Attempting to change a pointer used by an active TRANSFER will result in the error:

```
ERROR 612 Buffer pointer(s) in use
```

The fill pointer can be changed during an outbound transfer, but not during an inbound transfer. Similarly, the empty pointer can be changed during an inbound transfer, but not during an outbound transfer.

---

**Note**

When string variables are used as buffers, the length of the string should **not** be changed. Although this does not affect the operation of the buffer, it can prevent access to the contents of the buffer by the variable name.

---

# A Closer Look at Transfers

Once a buffer has been created and an I/O path name assigned to it, data can be transferred into or out of the buffer by a TRANSFER statement. Every TRANSFER will need a buffer as either its source or destination. For example:

```
TRANSFER @Source TO @Buffer
```
or
```
TRANSFER @Buffer TO @Destination
```

From the buffer's point of view, there are two types of transfers: inbound and outbound.

- An inbound transfer will move data from a device or file into the buffer, updating a fill pointer and byte count as it proceeds.
- An outbound transfer will remove data from the buffer, updating an empty pointer and byte count as necessary.

For a complete explanation, see the "Closer Look at Buffer Pointers" section near the end of this chapter.

## Transfer Methods

The actual method of transfer is device dependent and is chosen *automatically* by the BASIC system (you cannot explicitly choose a method). The three possible transfer methods are:

- DMA (direct memory access)
- FHS (fast handshake)
- INT (interrupt)

Descriptions of each method and how the system chooses one for each TRANSFER are covered in the section called "Transfer Methods and Rates".

## OUTPUT and ENTER and Buffers

The OUTPUT and ENTER statements may be used to interact with the data sent through the buffer. If the I/O path name of the buffer is used as the source for an ENTER or the destination for an OUTPUT, the control table (pointers, size, etc.) will be updated automatically.

Accessing the data in a named buffer by using the variable name will not update the buffer pointers. This could easily lead to corruption of the data in the buffer.

## Transfer Formatting

OUTPUT and ENTER statements can format data according to a given IMAGE list and transform the data according to the attributes specified in the ASSIGN statement. **No data formatting or transformation** occurs, however, when data are transferred by a TRANSFER statement.

## Transfer Termination

The ON EOT (End Of Transfer) statement allows you to define a branch to be taken upon the completion of a transfer. When the data being transferred has been divided into records, the ON EOR (End Of Recordr) statement can be used to define a branch to be taken after each record is transferred.

---

**Note**

An active TRANSFER will not be terminated by stopping or pausing a program. You may use [Reset] ([RESET]) or ABORTIO to terminate a TRANSFER prematurely. The [Break] ([CLR I/O]) key will not terminate a TRANSFER.

---

### Visually Determining Transfer Status

If a TRANSFER is active while a program is paused, the "I/O" indicator (`I\O` or `Transfer`) is displayed in the lower-right corner of the CRT instead of the "Pause" indicator (- or `Paused`). See the chapter entitled "Introduction to the System" in *Installing, Using, and Maintaining the BASIC System* for details of this status indicator.

# Choosing Transfer Parameters

For a standard inbound transfer, data from the device (or file) is placed in the buffer and the TRANSFER is deactivated when the buffer is full. For an outbound transfer, all data is removed from the buffer and the TRANSFER is deactivated when the buffer is empty.

## Continuing Transfers Indefinitely

To allow a TRANSFER to continue indefinitely, the CONT parameter can be specified.

```
TRANSFER @Source TO @Buffer;CONT
```

Several interesting things happen when a continuous TRANSFER is specified. Execution cannot leave the current program context unless the buffer and I/O path name are in COM (or passed as parameters), and you will not be able to LOAD, GET, or EDIT a program. During program development, you can terminate a transfer by [RESET] ([Reset]) or ABORTIO @Non_buff (use the I/O path name assigned to either the device or file). ABORTIO can be used in a program or executed from the keyboard.

A continuous TRANSFER can also be canceled by writing to a CONTROL register (use the I/O path name assigned to the buffer). Note that the CONTROL register only cancels the continuous mode. The TRANSFER is still active until the buffer is full or empty.

```
CONTROL @Buff,8;0    for inbound transfers

CONTROL @Buff,9;0    for outbound transfers
```

When the CONT parameter is specified for an inbound transfer, the transfer fills the buffer and is then suspended while program execution continues. The suspended transfer "sleeps" until another operation removes some data from the buffer. The transfer then "wakes up" and continues the transfer operation. When the CONT parameter is specified for an outbound transfer, the transfer empties the buffer and is then suspended. As soon as more data are available, the transfer "wakes up" and continues the transfer operation. This process proceeds until the transfer is completed or the CONT mode is canceled.

## Waiting for a Transfer to End (Non-Overlapped Transfers)

By default, transfers take place concurrently with continued program execution. To defer program execution until a transfer is complete, use the WAIT parameter. This allows transfers to take place serially (non-overlapped).

```
TRANSFER @Source TO @Buffer;WAIT
```

## Continuous Non-Overlapped Transfers

When the WAIT parameter is specified, the program statement following the TRANSFER will not be executed until the transfer has completed. By combining both the CONT and WAIT parameters, a continuous non-overlapped TRANSFER can be defined. However, this is only legal if you already have an active TRANSFER for the buffer in the opposite direction.

```
TRANSFER @Source TO @Buffer;WAIT,CONT
```

## Transferring a Specified Number of Bytes

The COUNT parameter tells a transfer how many bytes are to be transferred. The following TRANSFER specifies 32 bytes to be transferred. The transfer will terminate after 32 bytes have been transferred (or when the buffer becomes full for non-continuous transfers).

```
TRANSFER @Source TO @Buffer;COUNT 32
```

## Delimiter Characters

The DELIM parameter can be used to terminate an inbound transfer when a specified character is received. The following TRANSFER will terminate when the delimiter (comma) is sent or when the buffer is full (unless the CONT parameter is specified). The DELIM parameter is not allowed on outbound transfers or WORD transfers. If the DELIM string is the null string, the DELIM clause is ignored. This allows programmatic disabling of DELIM checking. An error results if the DELIM string contains more than one character.

```
TRANSFER @Source TO @Buffer;DELIM ","
```

## Using the END Indication with Transfers

The END parameter can also be used to terminate a TRANSFER. On an outbound transfer on an HP-IB interface, for example, specifying END causes an End-or-Identify (EOI) signal to be sent with the last character of the transfer.

```
TRANSFER @Buffer TO @Device;END
```

The END parameter is discussed in detail following the introduction of the RECORDS parameter.

## Transferring Records

It is often desirable to divide the data into records. The RECORDS parameter is then used to indicate the size of each record.

Whenever RECORDS is used, there must be a parameter which signals the end of a record. The EOR (End-Of-Record) parameter can use COUNT, DELIM, or END (discussed later) to signify the end of a record. For example, the following statement specifies 4 records of 15 bytes per record are to be transferred.

```
TRANSFER @Source TO @Buffer;RECORDS 4,EOR(COUNT 15)
```

## Multiple Termination Conditions

When multiple termination conditions are specified, the transfer will terminate when any one of the conditions occurs.

```
TRANSFER @Source TO @Buffer;COUNT 128,DELIM ";",END
TRANSFER @Source TO @Buffer;RECORDS 100,EOR(COUNT 15,END)
```

As in all transfer operations, unless the CONT parameter is specified, the TRANSFER will also terminate when the buffer is full or empty.

The END parameter specifies an inbound transfer will be terminated by receiving an interface-dependent signal (for devices) or by encountering the current end-of-file (for files). Some devices on the HP-IB send an EOI concurrently with the last byte of data. Unless the END parameter is specified, receiving an EOI will generate an error. For files, encountering the end-of-file will generate an error unless the END parameter is specified.

Using the END parameter with an outbound transfer on the HP-IB will result in the EOI signal being sent concurrently with the last byte of the transfer. If EOR(END) is specified, EOI will be sent with the last byte of each record. For files, END will cause the end-of-file pointer to be updated at the end of the transfer. Using EOR(END) will cause the pointer to be updated at the end of each record.

## TRANSFER Records and Termination

The following tables show the different system responses to the END and EOR(END) parameters.

**Table 9-1. Inbound TRANSFER**

| Parameter | File | Device |
|---|---|---|
| No END | Terminate prematurely. Bit 3 of Register 10 is set. Error 59 waiting. | Terminate prematurely. Bit 3 of Register 10 is set. Error 59 waiting. |
| END | Terminate normally. Bit 3 of Register 10 is set. | Terminate normally. Bit 3 of Register 10 is set. |
| EOR(END) | Finish current record. ON EOR triggered. Start new record. | Terminate normally. Bit 3 of Register 10 is set. |
| END,EOR(END) | Terminate normally. Bit 3 of Register 10 is set. | Terminate normally. Bit 3 of Register 10 is set. |

An error is logged when a transfer terminates prematurely. For overlapped transfers, this error is "waiting" and will be reported the next time the non-buffer I/O path name is referenced. At that time, any ON ERROR or ON TIMEOUT branches will be triggered. (If the WAIT parameter is specified, the error is reported immediately.) See "Error Reporting" for further explanation.

An ON END branch will be triggered only if the END parameter is not specified.

**Table 9-2. Outbound TRANSFER**

| Parameter | File | Device |
|---|---|---|
| No END | No special action. | No special action. |
| END | Update EOF pointer after TRANSFER is finished. | Send an EOI with the last byte of each record. |
| EOR(END) | Update EOF pointer after each record. | Send an EOI with the last byte of each record. |
| END,EOR(END) | Update EOF pointer after each record and when the TRANSFER is finished. | Send an EOI with the last byte of each record and with the last byte of the TRANSFER. |

For an outbound transfer to a device, no special action is taken if the device does not support EOI. The Serial, Datacomm and GPIO interfaces do not support EOI.

## Transfer Event-Initiated Branching

Two types of event-initiated branches can be defined for a transfer.

- The ON EOT statement defines and enables a branch to be taken upon completion of a transfer.
- The ON EOR statement defines and enables a branch to be taken every time a record is transferred.

```
ON EOT @Device CALL Process
ON EOR @File GOTO Parse
```

No ON EOR branches will be triggered unless the EOR parameter is specified in the TRANSFER statement and an item is transferred which satisfies one of the end-of-record conditions (COUNT, DELIM, or END).

To ensure that a branch receives service, the transfer must complete before attempting to leave the context in which the branches are defined. If the I/O path names are local to a program context, encountering SUBEND, SUBEXIT, or RETURN before the transfer has completed will cause the context switch to be deferred until completion of the transfer. If this happens, any ON EOR or ON EOT branch will not be serviced.

Certain statements wait until a transfer is completed before they are executed. A complete list of these statements is provided later in this chapter. These statements can be used to prevent overlapped operation or defer a context switch until completion of the transfer. For example, if the following I/O path names were used in a TRANSFER, either of the following statements will cause program execution to wait until the transfer is finished.

```
ASSIGN @Path TO *  (can be a device, file, or buffer)

WAIT FOR EOT @Non_buff  (can be a device or file)
```

When a TRANSFER is used inside a loop, the entire loop may execute before the transfer has completed. If this happens, the second execution of the TRANSFER statement will wait until the completion of the first. Any event-initiated branch defined for the TRANSFER (ON EOT or ON EOR) will be serviced.

While the WAIT parameter can be specified to ensure completion of a transfer before proceeding with the next statement (thus ensuring a branch can be serviced), this defeats any advantage of overlapped operation.

The WAIT FOR statement can be used to allow overlapped operation up to the point where the WAIT FOR statement is encountered. The WAIT FOR statement ensures the servicing of an event-initiated branch defined for the end-of-transfer or end-of-record.

# Terminating a Transfer

A transfer is usually terminated by satisfying the conditions specified by the transfer parameters. There are times, especially during program development, when you may wish to prematurely terminate (abort) a transfer.

A transfer can be aborted by pressing the [Reset] ([RESET]) key, which will stop the program, close all I/O paths, and destroy all buffer pointers.

To abort a transfer without stopping the program, the ABORTIO statement can be used from the program or the keyboard. For example:

```
ABORTIO @Non_buff
```

This statement will terminate any active transfer associated with the I/O path. ABORTIO has no effect if a transfer is not in progress. Using ABORTIO does not ensure all data in the buffer is transferred, but it does leave the buffer pointers and byte count in their correct state.

---

**Note**

If the destination of a TRANSFER is a mass storage file, aborting a TRANSFER with ABORTIO will not cause data already placed in the disc buffer to be written to the disc. Up to 255 bytes of data could be lost.

---

While most transfers are terminated by fulfilling the conditions specified by the parameters, a continuous TRANSFER (using the CONT parameter) requires a bit more effort to terminate.

To terminate a continuous TRANSFER without leaving data in the buffer, first cancel the continuous mode (with CONTROL), then wait for the transfer to complete. Use register 8 for inbound transfers and register 9 for outbound transfers. The following two methods are the safest ways of terminating a continuous TRANSFER.

```
CONTROL @Buff,8;0
WAIT FOR EOT @Path

CONTROL @Buff,8;0
ASSIGN @Path TO *
```

Remember that the buffer pointers are not reset to the beginning of the buffer when the transfer is finished. The RESET statement (RESET @Buff) can be used to reset the buffer pointers to the beginning of the buffer and the byte count to zero.

Transfers are not terminated by pausing the program. The I/O indicator in the lower-right corner of the CRT will indicate when a transfer is in progress.

While transfers may continue when the computer is in the paused state, all transfers must terminate before entering the stopped state. Pressing [Return] or [ENTER], after editing or adding a program line, will attempt to put the computer in the stopped state. If a transfer is still in progress, the computer will "hang" until the transfer is completed. To abort the transfer without performing a hardware reset, press [Break] ([CLR I/O]) to clear the [Return] or [ENTER] and then execute an ABORTIO on the non-buffer I/O path name for each active TRANSFER. If a hardware reset can be tolerated, press [Reset] ([RESET]) to terminate the transfer.

# More Transfer Examples

Here is a short program which sets up a continuous transfer from a device through the buffer to a BDAT file. A program of this type is useful when the data being received must be saved for later analysis.

```
10    ! Data Logging Example
20    !
30    ! Buffer size should be a multiple of disc sector (256) size.
40    ASSIGN @Device TO 717          ! Assign source device on HPIB
50    ASSIGN @Buf TO BUFFER [512]    ! Assign BUFFER
60    ASSIGN @File TO "LOG_FILE"     ! Assign destination file
70    !
80    TRANSFER @Device TO @Buf;CONT  ! Continuous TRANSFER
90    TRANSFER @Buf TO @File;CONT    ! Continuous TRANSFER
100   !
110   ! Program execution continues ...
120   ! Data logging continues as a "background" task ...
130   !
140   PAUSE                  ! TRANSFER continues in paused state
150   END
```

The following program creates and fills a BDAT file and then sends its contents to a printer. Notice that the OUTPUT statement used to fill the file placed a CR/LF at the end of each record. The TRANSFER statement (line 90) looks for the carriage-return as a record delimiter.

```
10    ON ERROR CALL Makefile
20    ASSIGN @File TO "BDAT_FILE"    ! Test for file's existence
30    OFF ERROR
40    ASSIGN @Buff TO BUFFER [2046] ! Assign buffer
50    ASSIGN @Print TO PRT           ! Assign destination
60    !
70    Cr$=CHR$(13)                  ! ASCII character for carriage return
80    PRINT "Start"
90    TRANSFER @File TO @Buff;RECORDS 10,END,EOR (DELIM Cr$)
100   !
110   TRANSFER @Buff TO @Print
120   FOR I=1 TO 10000
130     PRINT "TRANSFERS RUNNING",I
140     STATUS @Buff,11;Stat
150     IF NOT BIT(Stat,6) THEN 180
160   NEXT I
170   !
180   OUTPUT @Print;CHR$(12)         ! ASCII character for formfeed
190   PRINT "File is printed"
200   END
210   !
220   SUB Makefile
230     OFF ERROR
240     CREATE BDAT "BDAT_FILE",10,12
250     ASSIGN @File TO "BDAT_FILE";FORMAT ON
260     FOR I=1 TO 10
270       DISP "Writing";I
280       READ Word$
290       OUTPUT @File;Word$
300     NEXT I
310     DISP
320     DATA ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,TEN
330   SUBEND
```

The next program continually shows the activity of the buffer. Note that a continuous TRANSFER is used (line 90). Data is placed in the buffer a few bytes at a time (line 130) and the status is displayed by the SUB called from line 140. After a few hundred bytes are transferred, the continuous mode is canceled (line 180), the program waits for the transfer to finish (line 190), and the final status is displayed.

```
20    PRINTER IS CRT
30    PRINT USING "@"                 ! Clear Screen
40    COM @Buff,@Print,B$[47] BUFFER  ! Declare variables
50    INTEGER Characters
60    ASSIGN @Buff TO BUFFER B$       ! Assign I/O path name to buffer
70    ASSIGN @Print TO PRT            ! Assign I/O path name to 701
80    DISP "printer is off line"      ! Transfer hangs if no printer
90    TRANSFER @Buff TO @Print;CONT   ! Continuous transfer
100   DISP                            ! Clear display line
120   REPEAT
130     OUTPUT @Buff;"AB ";           ! Fill buffer with data
140     CALL Buff_status
150     Times=Times+1
160   UNTIL Times>100
180   CONTROL @Buff,9;0               ! Cancel continuous mode
190   WAIT FOR EOT @Print             ! Wait for buffer empty
200   CALL Buff_status                ! Show final status
210   END
230   SUB Buff_status ! ----------------------------------------------
240     COM @Buff,@Print,B$ BUFFER
250     STATUS @Buff;R0
260     PRINT TABXY(1,1);"Buffer Status: ";
270     STATUS @Buff,1;R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13
280     IF R1=1 THEN PRINT "Named ";
290     IF R1=2 THEN PRINT "Unnamed ";
300     PRINT "Buffer[";VAL$(R2);"]"
310     PRINT TABXY(1,3);RPT$(" ",55)
320     PRINT TABXY(R3,3);"v"          ! Show fill pointer position
330     PRINT TABXY(1,4);"""";B$;""""  ! Show buffer contents
340     PRINT TABXY(1,5);RPT$(" ",55)
350     PRINT TABXY(R5,5);"^"          ! Show empty pointer position
360     PRINT
370     PRINT "Fill pointer:  ";R3
380     PRINT "Bytes in use:  ";R4
390     PRINT "Empty pointer: ";R5
400     PRINT
410     PRINT "           inbound/outbound"
420     PRINT "Select code:   ";R6;"/";R7
430     PRINT "Continuous? :  ";R8;"/";R9
440     PRINT "Term. status:  ";R10;"/";R11
450     PRINT "Total bytes:   ";R12;"/";R13
460   SUBEND
```

Data currently in the buffer can be reused or ignored by manipulating the pointers (with CONTROL). When it is necessary to move data through the buffer without using I/O path names, the CONTROL statement can be used to modify the pointers, thus allowing a TRANSFER to take place. The next program uses this technique. The array size used in the next program is for the Model 236; change the array size in lines 50 and 60 for other computer models.

```
10    GINIT                              ! Uses graphics
20    GCLEAR
30    GRAPHICS ON
40    PRINT CHR$(12)                     ! Clear the screen
50    INTEGER I,Graph(1:12480) BUFFER    ! (1:7500) FOR 9826/9816
60    Gbytes=2*12480                     ! 2 * 7500 FOR 9826/9816
70    ASSIGN @Buff TO BUFFER Graph(*)
80    ON ERROR GOTO Record               ! Enable ERROR trap
90    ASSIGN @Read TO "PHOTOS"           ! Test if file exists
100   ASSIGN @Read TO *                  ! Close file
110   GOTO Playback                      ! If file exists then Playback
120                                      !
130 Record:OFF ERROR
140   CREATE BDAT "PHOTOS",5,Gbytes      ! 5 "PHOTOS" of graphics screen
150   ASSIGN @Write TO "PHOTOS"          ! to be written to the BDAT file
160   FOR I=1 TO 5
170     GRID I*4,I*4
180     GSTORE Graph(*)                  ! Fill buffer with GSTORE
190     GCLEAR
200     DISP "SAVING #";I
210     CONTROL @Buff,4;Gbytes           ! Tell TRANSFER "The buffer is full"
220     TRANSFER @Buff TO @Write;WAIT
230   NEXT I
240   ASSIGN @Write TO *
250   !
260 Playback:OFF ERROR
270   ASSIGN @Read TO "PHOTOS"
280   FOR I=1 TO 5
290     DISP "LOADING #";I
300     TRANSFER @Read TO @Buff;WAIT
310     GLOAD Graph(*)
320     CONTROL @Buff,4;0                ! Tell TRANSFER "The buffer is empty"
330   NEXT I
340   DISP "DONE"
350   END
```

The program creates five "photos" of the graphics raster and writes them to a disc file. The file is then read and each picture is loaded back into the graphics raster.

# Special Considerations

## Transfer with Care

Whenever possible, a transfer will take place concurrently with continued program execution. You must carefully construct a program using transfers. **A poorly designed transfer may take longer to execute than using OUTPUT and ENTER.**

A TRANSFER which uses a local I/O path name must terminate before a SUBEXIT, SUBEND, or RETURN (from a function) can return execution to the calling context. The system will detect that such a transfer is in progress and will make the SUBEXIT wait for the transfer to terminate. If this happens, the system will not process any ON EOT (or ON EOR) branch which had been defined for the transfer. To allow servicing of the branch, any statement which cannot execute in overlap with the TRANSFER can be inserted in the subprogram before the SUBEXIT. Two of the most sensible choices are:

```
WAIT FOR EOT @Non_buff
```

or

```
ASSIGN @Path to *
```

A TRANSFER which uses only non-local I/O path names can execute in overlap with a SUBEXIT. One word of caution is necessary: if a local ON EOT (or ON EOR) statement is used in the subprogram, its branch will not be serviced if the SUBEXIT is encountered before termination of the TRANSFER. To ensure the possibility of servicing the branch, insert a statement that cannot execute in overlap with the TRANSFER. This is essentially the same technique discussed in the preceding paragraph.

More than one I/O path name can be assigned to a named buffer; however, each path name will maintain its own set of pointers. Using multiple path names on the same buffer could lead to corruption of the data in the buffer.

Special care should be taken when using REAL and COMPLEX arrays as buffers, since a device may send a bit pattern that is not a valid real number. Accessing the data as a REAL or COMPLEX value may produce an error.

### Statements Which Affect Concurrency

The following statements do **not** wait for the completion of a TRANSFER statement.

| Buffer in Use | Device in Use |
|---|---|
| `STATUS @Buf` | `STATUS @Dev` |
| `CONTROL @Buf` | `ON EOR @Dev` |
| `SCRATCH A` | `ON EOT @Dev` |
| | `OFF EOR @Dev` |
| | `OFF EOT @Dev` |

Statements which wait for completion of inbound transfers.

```
OUTPUT @Buf
TRANSFER @Dev TO @Buf
```

Statements which wait for completion of outbound transfers.

```
ENTER @Buf
TRANSFER @Buf TO @Dev
```

Statements which wait for completion of inbound and outbound transfers.

| Buffer in Use | Device in Use |
|---|---|
| `ASSIGN @Buf TO *` | `ASSIGN @Dev TO *` |
| `ASSIGN @Buf TO BUFFER[bytes]` | |
| `ASSIGN @Buf TO BUFFER B$` | |
| `ASSIGN @Dev` | `ASSIGN @Dev` |
| `ASSIGN @Dev; (new attributes)` | `ASSIGN @Dev;` *(new attributes)*<br>`WAIT FOR EOT @Dev`<br>`OUTPUT @Dev`<br>`ENTER @Dev`<br>`TRANSFER @Buf TO @Dev`<br>`TRANSFER @Dev TO @Buf` |
| `END` | `END` |
| `SUBEXIT` | `SUBEXIT` |
| `SUBEND` | `SUBEND` |
| `SCRATCH C` | `SCRATCH C` |
| `SCRATCH` | `SCRATCH` |
| `LOAD "PROG"` | `LOAD "PROG"` |
| `GET "PROG"` | `GET "PROG"` |
| `STOP` | `STOP`<br>`CONTROL @Dev` |

## Error Reporting

If an error is encountered during an overlapped transfer, the error is logged in the non-buffer I/O path name and reported the next time the non-buffer I/O path name is referenced. Thus, the error line reported will be the most recently executed line containing the I/O path name and usually not the line containing the TRANSFER statement. For example:

```
10    !   This program shows delayed error reporting for TRANSFER
20    !
30    ON ERROR GOTO Ok
40    PURGE "bdat_file"              ! Zap file if it already exists
50 Ok:OFF ERROR
60    !
70    CREATE BDAT "bdat_file",1      ! CREATE an empty file
80    ASSIGN @Non_buf TO "bdat_file"! ASSIGN I/O path name to the file
90    INTEGER B(100) BUFFER          ! Declare a variable as a buffer
100   ASSIGN @Buf TO BUFFER B(*)     ! Assign I/O path name to buffer
110   PRINT
120   !
130   WAIT 2
140   LIST 150,150
150   TRANSFER @Non_buf TO @Buf;CONT ! Error occurs in this line
160   !
170   WAIT 2
180   LIST 190,190
190   STATUS @Buf,10;Status_byte     ! Error not reported with @Buf
200   !
210   WAIT 2
220   LIST 230,230
230   STATUS @Non_buf;Status_byte    ! Error reported with @Non_buf
240   END
```

Since a continuous TRANSFER was specified, the error that occurs in line 150 is reported in line 230 when the non_buffer I/O path name is referenced. For continuous transfers, the error is always logged with the non-buffer I/O path name. Referencing the buffer's I/O path name (line 190) does not cause the error to be reported. After running the program, change the CONT parameter in line 150 to WAIT. The program will now report the error in line 150 since the WAIT parameter specified a serial TRANSFER.

At the time the error is reported, any ON END (for files), ON TIMEOUT (for devices), or ON ERROR statements will be triggered. However, ON END is not triggered when the END parameter is specified.

## Suspended Transfers

When a TRANSFER statement is executed, that transfer is said to be "active". The transfer proceeds until either a termination condition is reached, or until there is nothing else the transfer can do for the time being. An example of the latter is a continuous TRANSFER, which does not terminate when the buffer is full and has not yet met any other termination conditions.

This TRANSFER will be "suspended" to give some other TRANSFER operation a chance to empty the buffer. It will not be reactivated until one of the following occurs:

1. The other TRANSFER operation reaches a record boundary, fills or empties the buffer, terminates, or is suspended.
2. An OUTPUT or ENTER operation active in the other direction fills or empties the buffer, or terminates.
3. A CONTROL statement is executed to change the fill or empty pointers, or buffer's byte count.
4. A CONTROL statement is executed to cancel continuous mode.

A TRANSFER cannot be suspended unless it has CONT as one of its transfer parameters.

# Transfer Performance

## Sector Size

For the best performance when transferring BDAT and HP-UX files, the buffer size should be a multiple of 256 or 1024 bytes (the size of a sector on the disc).[1] If the buffer is not a multiple of 256 bytes, the system must do sector buffering; this is handled automatically, but reduces the transfer rate.

## Internal Disc Drives of Models 226 and 236 Computers

While a TRANSFER can be assigned to the internal disc drives in the Model 226 and Model 236, no noticeable increase in speed (compared to OUTPUT or ENTER) will result. Transfers to and from external mass storage (except the 9885) will show an increase in speed, especially if a DMA card is present.

## Overlapped Transfers and Disc Drives

Some of the discs are capable of overlapped operation. This means that other processing can occur while a non-continuous TRANSFER to or from the disc is taking place. In other words, the program can execute other statements before the transfer has completed. Overlapped discs include:

- CS80 discs (such as the HP 9133 and 9153)
- SS80 discs (such as the HP 9122)
- "Amigo" discs (such as the HP 9895 and 82901)

Discs which are not capable of overlapped operation are called serial discs. When executing a non-continuous TRANSFER to or from a serial disc, the program will not leave the TRANSFER statement until it completes. Serial discs include the internal discs (of Models 226 and 236 computers) and the HP 9885 8-inch flexible disc drive.

[1] Discs with 512-byte sectors cannot be used with TRANSFER.

The following example illustrates the difference between a serial disc and an overlapped disc.

```
10    OPTION BASE 1
20    INTEGER B(128,10)               ! A 10-sector buffer
30    LINPUT "Enter msus:",Msus$
40    CREATE BDAT "bdat"&Msus$,10
50    ASSIGN @File TO "bdat"&Msus$
60    ASSIGN @Buffer TO BUFFER [2560];FORMAT OFF
70    OUTPUT @Buffer;B(*)             ! Fill @Buffer's with 10 sectors
80    ON EOT @File GOTO Serial_eot    ! Branch taken if TRANSFER is serial
90    TRANSFER @Buffer TO @File
100   ON EOT @File GOTO Overlapped_eot! Branch taken if TRANSFER is overlapped
110   LOOP
120     I=I+1
130     PRINT I,"OVERLAPPED"
140   END LOOP
150 Serial_eot:  !
160   PRINT "SERIAL"
170 Overlapped_eot: !
180   ASSIGN @File TO *
190   PURGE "bdat"&Msus$
200   END
```

If this program is used with a serial disc, the program stays in the TRANSFER statement until the transfer is complete. Upon completion of the transfer, the ON EOT branch to Serial_eot is taken.

If this program is used with an overlapped disc, the TRANSFER statement begins the transfer, but the program executes the next statement before the transfer completes. In this program, the next statement changes the ON EOT branch. During the transfer, a count and the word "OVERLAPPED" are printed. When the transfer is complete, the ON EOT branch to Overlapped_eot is taken.

If the CONT parameter is specified for a TRANSFER with a serial disc, the transfer appears overlapped because the program executes any statements which follow the TRANSFER statement before the transfer terminates. Here is what really happens in this case. The transfer proceeds until the buffer is full (for inbound transfers) or empty (for outbound transfers). The transfer is then suspended because CONT was specified. The TRANSFER statement is exited and the next statement is executed. The transfer will remain suspended until the continuous mode is terminated or until the buffer is filled (for inbound transfers) or until the buffer is emptied (for outbound transfers). If there is a second TRANSFER active for the buffer, an EOR or EOT condition for the second TRANSFER can also wake up the suspended TRANSFER.

In contrast to serial discs, overlapped discs would allow the statement following the TRANSFER to execute before the buffer was full or empty.

The following program illustrates a transfer to a serial device which appears overlapped.

```
10    OPTION BASE 1
20    INTEGER B(128,10)                ! A 10-sector buffer
30    LINPUT "Enter Overlapped msus:",Overlapped$
40    CREATE BDAT "bdat"&Overlapped$,10
50    LINPUT "Enter Serial msus:",Serial$
60    CREATE BDAT "bdat"&Serial$,10
70    ASSIGN @Overlapped TO "bdat"&Overlapped$
80    OUTPUT @Overlapped;B(*)
90    RESET @Overlapped                ! Position to beginning
100   ASSIGN @Buffer TO BUFFER [512];FORMAT OFF
110   ASSIGN @Serial TO "bdat"&Serial$
120   ON EOT @Overlapped GOTO Eof
130   TRANSFER @Overlapped TO @Buffer;END,CONT
140   TRANSFER @Buffer TO @Serial;CONT
150   LOOP
160     I=I+1
170     PRINT I,"OVERLAPPED"
180   END LOOP
190 Eof:  !
200   CONTROL @Buffer,9;0
210   ASSIGN @Overlapped TO *
220   PURGE "bdat"&Overlapped$
230   ASSIGN @Serial TO *
240   PURGE "bdat"&Serial$
250   END
```

In this example, an overlapped disc is used to fill the buffer while a serial disc empties the buffer. Any overlapped device could have been used. After both TRANSFER statements are executed, the program prints the count and the word "OVERLAPPED" while reading from one disc and writing to the other disc. The inbound transfer is terminated when it encounters the end of the file. The outbound transfer is terminated when the CONTROL statement cancels the CONT mode.

## Transfer Methods and Rates

The BASIC system chooses the *fastest possible* transfer method when executing a TRANSFER (you cannot explicitly choose the method).

### Available Methods

There are three types of transfers available to the BASIC system.

- DMA (direct memory access)
- FHS (fast handshake)
- INT (interrupt)

### DMA Mode

All transfers use DMA mode whenever possible. However, any one of the following reasons will prevent a DMA transfer.

- The DMA card is not present
- Both DMA channels are busy
- The device involved is not HP-IB or GPIO
- The DELIM parameter is specified

If DMA cannot be used with the HP-IB or GPIO interfaces, the FHS mode will be used if the WAIT parameter was specified and INT mode will be used if the WAIT parameter was not specified.

### INT Mode

The INT mode will always be used for the Serial and Datacomm interfaces. Note also that the handshake lines are **not** used for Serial and Datacomm transfers. Therefore, on inbound transfers through the Serial interface, it is easy to overrun the 1-byte hardware buffer on the card. The maximum transfer rate with Serial interfaces is hard to specify, because it may be affected by other operations that attempt to alter the BASIC interrupt-logging structure (statements such as ON INTR and ON KEY). In general, using the WAIT parameter will result in a higher transfer rate, with a lower potential for overrun errors, than other methods. The WAIT parameter specifies that the TRANSFER is to complete before the next BASIC statement is executed (that is, it specifies that the transfer is to be performed in non-overlapped mode).

If a very slow device is sending a few bytes at a time, the most efficient method of transfer would be to interrupt the processor whenever data is ready. Both DMA and INT modes operate in this way. The DMA hardware "steals" a single memory cycle from the processor to transfer each byte. The INT mode must completely interrupt the processor and therefore takes more time.

Either type of interrupt (DMA or INT) can occur at any time and will be handled immediately by the system. The interrupt doesn't have to wait for a statement to end before it is serviced. This is not the same as event-initiated branches which are serviced only at the end of a statement.

### Burst Interrupt Mode

The INT transfers implemented on the HP-IB and GPIO interfaces use a specialized "burst interrupt" mode. When an interrupt occurs, the system's interrupt service routine will transfer the byte (or word) then wait approximately 20 $\mu$s for another byte. If the device is fast enough to accept or generate another byte each 20 $\mu$s, the net transfer rate will be much faster than if the system must exit the service routine and then re-enter the routine for the next byte.

### Approximate Transfer Rates for Devices

The following table shows the *approximate* transfer rates of various devices.

**Table 9-3. Device Transfer Rates**

| Device | Burst Interrupt | Fast Handshake | DMA | Burst DMA |
|---|---|---|---|---|
| HP-IB (98624 and built-in)<br>inbound<br>outbound<br>(bytes/second) | <br>55K<br>75K | <br>130K<br>120K | <br>350K<br>290K | <br>—<br>— |
| GPIO (98622)<br>inbound<br>outbound<br>(transfers/second) | <br>65K<br>75K | <br>115K<br>115K | <br>540K<br>525K | <br>930K<br>1050K |
| Serial (98626, 98644, and built-in) | 19 200 Baud[1] | — | — | — |
| Datacomm (98628) | 19 200 Baud | — | — | — |

[1] Note that the maximum rate for *inbound* transfers through a Serial interface is generally *much* lower than this for two reasons: TRANSFER does not use the handshake lines, and there is only a 1-character hardware buffer on Serial cards.

## Restrictions

All data must be buffered. This means every TRANSFER statement will have one I/O path assigned to a buffer and one I/O path assigned to a device (or file). Additionally, transfers are **not permitted** with:

- The CRT or keyboard
- The HP 98623 BCD Interface card
- The tape backup on CS80 disc drives
- ASCII type files
- Discs initialized with 512-byte sectors (formatting option 2)

In addition, TRANSFER to or from a mass storage device with hierarchical directories (such as HFS and SRM volumes) will not operate in overlapped mode (because of the "extensible" nature of files on these volumes).

A buffer can only have one inbound and one outbound I/O operation (using I/O path names) at any given time. The I/O operation can use TRANSFER, OUTPUT, or ENTER statements. A second I/O operation in the same direction must wait until the completion of the current operation. A second I/O operation in the opposite direction does not have to wait.

The HP-IB and GPIO interfaces support only one I/O operation at any given time. A second operation must wait until the completion of the first operation. The Serial and Datacomm interfaces allow concurrent inbound and outbound transfer operations if each TRANSFER has a unique I/O path name assigned to the device. An OUTPUT or ENTER must wait until completion of transfers in both directions. Thus, concurrent operation requires using TRANSFER statements and not a mixture of TRANSFER, OUTPUT, and ENTER statements.

The I/O path name assigned to a device can be used in only one I/O operation at a time. However, the path name can be used with OUTPUT, ENTER, and TRANSFER interchangeably. An OUTPUT or ENTER to the I/O path name will be deferred until completion of any active TRANSFER for that path name. All file operations (including CAT, CREATE, OUTPUT, and ENTER) will be deferred until completion of any TRANSFER using the same interface select code.

## Interactions with Other Keywords

The TRANSFER statement restricts some of the interrupts on various devices. If an ON INTR statement and an ENABLE INTR statement have been executed for an interface, not all possible ON INTR conditions will be triggered during a transfer.

### GPIO

For the GPIO interface, the PFLG (data ready) interrupt is not triggered during a transfer that uses the interface. The EIR (External Interrupt Request) interrupt is triggered even if there is a transfer in progress.

### Serial

For the Serial Serial interface, the Transmitter Holding Register Empty and Receiver Buffer Full interrupts are not triggered during a transfer that uses the interface. The Receiver Line Status and Modem Status Change interrupts are triggered even if there is a transfer in progress.

### Datacomm

For the Datacomm interface, all interrupt conditions are triggered even if a transfer is in progress.

### HP-IB

For the HP-IB interface, all interrupt conditions are triggered if they occur during a transfer. However, certain interrupt conditions may occur which will cause the transfer operation to be prematurely terminated.

With the exception of the Handshake Error, the majority of interrupt conditions only occur when the HP-IB interface is configured as a non-controller. If any of the following interrupt conditions are enabled and the given interrupt occurs during a transfer to or from the interface, the user interrupt will be logged and the TRANSFER will be prematurely terminated.

- Parallel Poll Configuration Change
- My Talk Address Received
- My Listen Address Received
- Talker/Listener Address Change
- Trigger Received

- Handshake Error
- Unrecognized Universal Command
- Secondary Command While Addressed
- Clear Received
- Unrecognized Address Command

If one of these interrupt conditions occurs and the given interrupt condition has not been enabled, the interrupt will be ignored and the TRANSFER will not be terminated.

---

**Note**

When an abortive interrupt condition is ignored, it is possible for data to be corrupted. It is recommended that abortive interrupt conditions be enabled during a transfer.

---

The Active Controller and IFC Received interrupt conditions will always prematurely terminate a TRANSFER, even if they have not been enabled.

### Premature Termination

When an overlapped TRANSFER is prematurely terminated because of an abortive interrupt condition, the following error is logged in the non-buffer I/O path name associated with the given TRANSFER. The error will then be reported the next time the I/O path name is referenced.

```
ERROR 167 I/O interface status error
```

Note that if an ON INTR condition is triggered during a transfer, the ON INTR service routine will be executed at the next end-of-line. However, if a TRANSFER is using the interface specified in an ENABLE INTR statement, the ENABLE INTR statement will wait for the transfer to complete. This means that only one interrupt condition can be triggered during a TRANSFER since the interface's interrupts cannot be re-enabled until completion of the transfer.

## Changing Buffer Attributes

You can change the I/O path name's attributes without changing the current buffer pointers. Just execute another ASSIGN statement with the new attributes. For example:

```
ASSIGN @Path;PARITY OFF
```

You will not be able to change all possible attributes in this manner. The BYTE and WORD attributes cannot be changed once assigned.

By specifying just the I/O path name, the default attributes (except BYTE) can be restored. For example:

```
ASSIGN @Path
```

See the ASSIGN statement in the *BASIC Language Reference* for a complete list of attributes.

---

**Note**

It is possible to assign more than one I/O path name to a single named buffer. Using two I/O path names on the same buffer could lead to the corruption of the data in the buffer. Although each path name maintains a separate set of buffer pointers, they are pointing to the same buffer.

---

# Buffer Status and Control Registers

**STATUS Register 0**

0 = Invalid I/O path name
1 = I/O path assigned to a device
2 = I/O path assigned to a data file
3 = I/O path assigned to a buffer

When the status of register 0 indicates a buffer (3), the status and control registers have the following meanings.

**STATUS Register 1** Buffer type (1=named, 2=unnamed)

**STATUS Register 2** Buffer size in bytes

**STATUS Register 3** Current fill pointer

**CONTROL Register 3** Set fill pointer

**STATUS Register 4** Current number of bytes in buffer

**CONTROL Register 4** Set number of bytes

**STATUS Register 5** Current empty pointer

**CONTROL Register 5** Set empty pointer

**STATUS Register 6** Interface select code of inbound TRANSFER

**STATUS Register 7** Interface select code of outbound TRANSFER

**STATUS Register 8** If non-zero, inbound TRANSFER is continuous

**CONTROL Register 8** Cancel continuous mode inbound TRANSFER if zero

**STATUS Register 9** If non-zero, outbound TRANSFER is continuous

**CONTROL Register 9** Cancel continuous mode outbound TRANSFER if zero

**STATUS Register 10** Termination status for inbound TRANSFER

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| 0 | TRANS-FER Active | TRANS-FER Aborted | TRANS-FER Error | Device Termi-nation | Byte Count | Record Count | Match Character |
| Value=0 | Value=64 | Value=32 | Value=16 | Value=8 | Value=4 | Value=2 | Value=1 |

**STATUS Register 11** Termination status for outbound TRANSFER

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| 0 | TRANS-FER Active | TRANS-FER Aborted | TRANS-FER Error | Device Termi-nation | Byte Count | Record Count | 0 |
| Value=0 | Value=64 | Value=32 | Value=16 | Value=8 | Value=4 | Value=2 | Value=0 |

**STATUS Register 12** Total number of bytes transferred by last inbound TRANSFER

**STATUS Register 13** Total number of bytes transferred by last outbound TRANSFER

# Index

## a

# b

## C

# d

# e

# f

# g

# h

# i

# k

# l



# m



# n

## O

# p

## r

# S

# t

## u



## w

# MANUAL COMMENT CARD

**BASIC 5.0**

**Interfacing Techniques**

HP 9000 Series 200/300 Computers

HP Part No.98613-90022

Name: _____

Company: _____

Address: _____

_____

Phone No: _____

Thank you for taking the time to respond.

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

_____.


**HP Part Number**
**98613-90022**
Microfiche No. 98613-99022
Printed in U.S.A. 1/87


For Internal Use Only