HP 9800 Computer Systems

BASIC Interfacing Techniques

for the 9826 Computer

HEWLETT PACKARD

BASIC Interfacing Techniques

for the 9826 Computer

Part No. 09826-90020
Microfiche No. 09826-99020

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

Printing History

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

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

October 1981...First Edition

Table of Contents

Chapter 1
Manual Overview

Introduction

This manual is intended to present the concepts of computer interfacing that are relevant to programming the 9826. 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 to the 9826 computer. If you would like a more detailed discussion of these 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 quick reference.

This manual begins by discussing the terminology and fundamental concepts of interfacing and continues through the specific details of using each type of interface. To get maximum benefit from the manual, you should read it serially through Chapter 7 to gain an complete picture of the relevant terminology, the overall communication process, and the general programming techniques common to all 9826 interfaces. The remainder of the chapters then explain the details of programming each type of interface. Chapter 10, "Unified I/O", describes the overall interfacing scheme of the 9826 and provides some additional insight on using the features of the machine to their fullest extent.

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

Chapter Preview

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 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 the new data type known as "I/O path names" in I/O statements are described. All readers should read all of the information in Chapters 3 through 7.

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 - Interface Events
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 - The Internal CRT Interface
This chapter describes accessing the built-in CRT display through its interface to the computer. Since this device can be accessed via its interface, most of the programming techniques presented in Chapters 3 through 7 can be used with this device. 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 9 - The Internal Keyboard Interface
As with Chapter 8, this chapter describes several programming techniques applicable to interfacing to the built-in keyboard, and several examples are given that will help you understand many of the general programming techniques presented in previous chapters. All of the capabilities of the keyboard are explained in this chapter.

Chapter 10 - Unified I/O
This chapter presents several powerful capabilities of the I/O scheme of the 9826. 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 11 - 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 2
Interfacing Concepts

Introduction

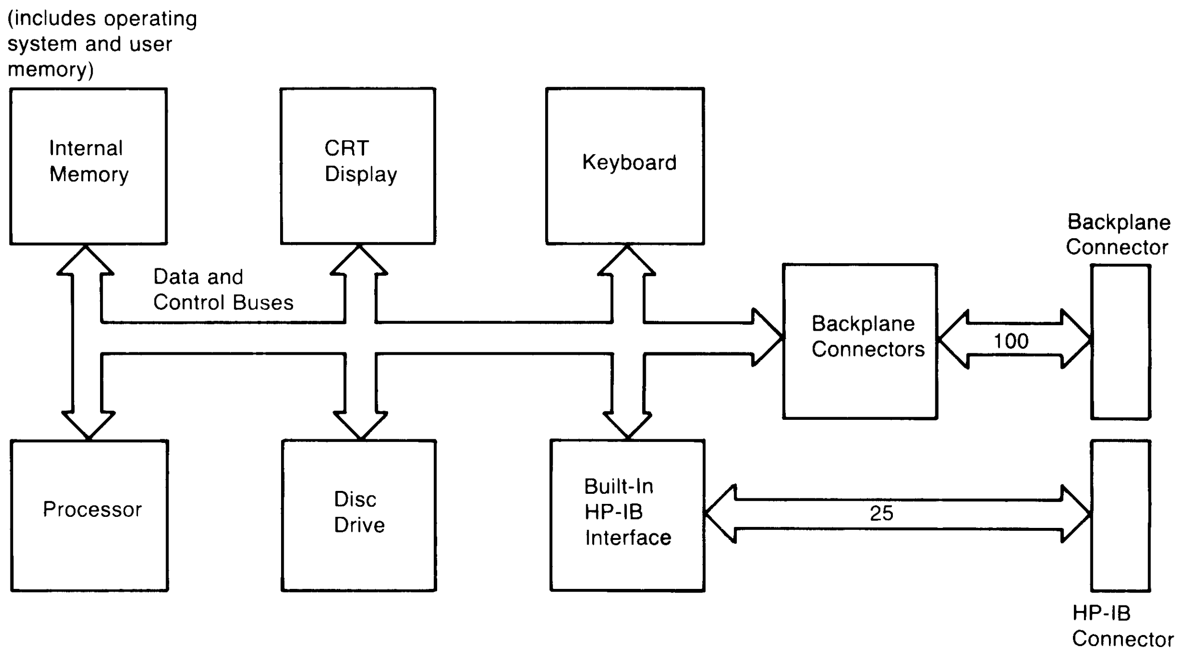
This chapter describes the functions and requirements of interfaces between the computer and its resources. Most of the concepts in this chapter are presented in an informal manner. Hopefully, **all** levels of programmers can gain useful background information that will increase their understanding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

Terminology

These terms are important to your understanding of the text of this manual. They are not highly technical, so don't worry about not having a PhD. in computer science to be able to understand all of them. The purpose of this section is to make sure that our terms have the same meanings.

The term **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. The term **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.

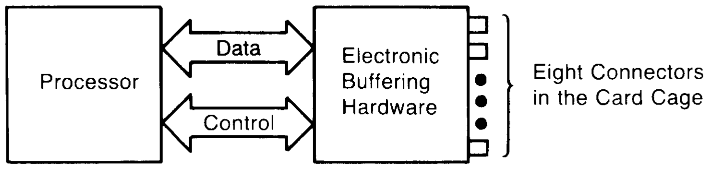
The term **hardware** describes both the electrical connections and electronic devices that make up the circuits within the computer; any piece of hardware is an actual physical device. The term **software** describes the user-written, BASIC-language programs. **Firmware** refers to the pre-programmed, machine-language programs that are invoked by BASIC-language statements and commands. As the term implies, firmware cannot be modified by the user. The machine-language routines of the operating system are firmware programs.



Block Diagram of the Computer

The term **I/O** is an acronym that comes from "Input and Output"; it refers to the process of copying data to or from computer memory. Moving data from computer memory to another resource is called **output**. During output, the **source** of data is computer memory and the **destination** is any resource, including memory. Moving data from a resource to computer memory is **input**; the source is any resource and the destination is a variable in computer memory. **Input is also referred to as enter in this manual** for the sake of avoiding confusion with the INPUT statement.

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


Processor
Data
Control
Electronic Buffering Hardware
Eight Connectors in the Card Cage

Backplane Hardware

Why Do You Need an Interface?

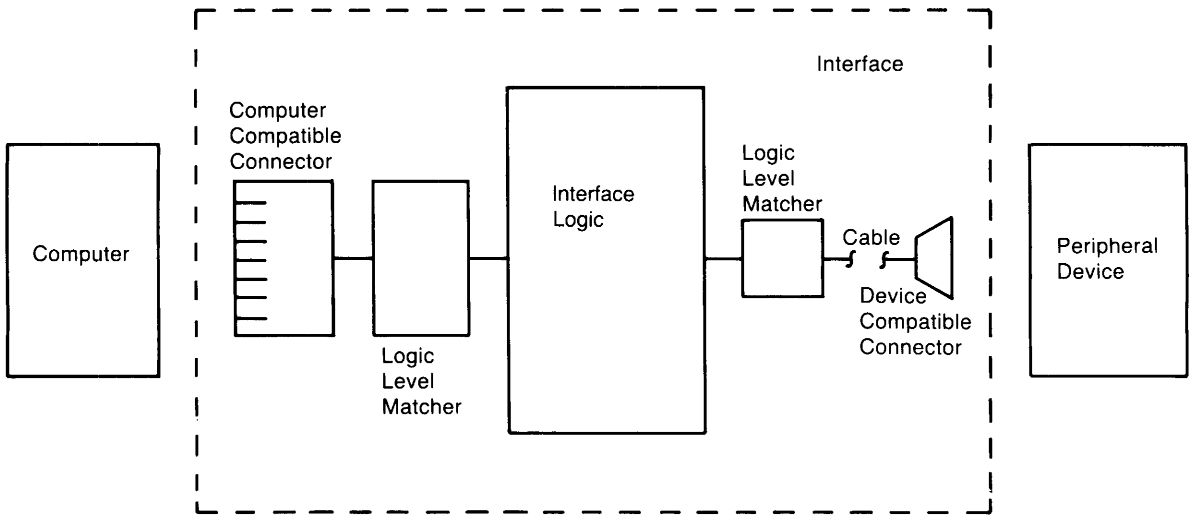
The primary function of an interface is, obviously, to provide a communication path for data and commands between the computer and its resources. Interfaces act as intermediaries between resources by handling part of the "bookkeeping" work, ensuring that this communication process flows smoothly. The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



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 64-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

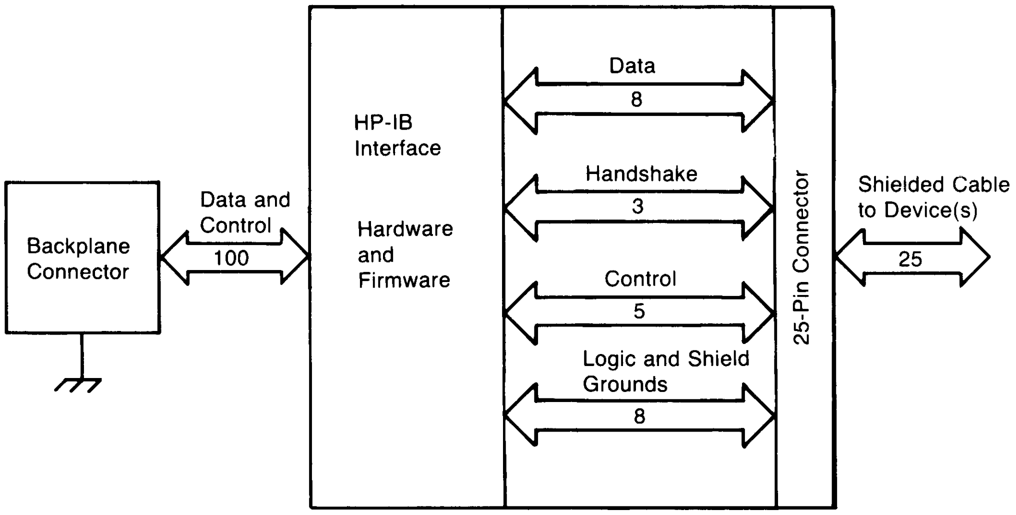
Another powerful feature of some interface cards is to relieve the computer of low-level tasks, such as performing data-transfer handshakes. This distribution of tasks eases some of the computer's burden and also decreases the otherwise-stringent response-time requirements of external devices. The actual 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 9826. 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 1975 Standard Digital Interface for Programmable Instrumentation. The acronym "HP-IB" comes from Hewlett-Packard Interface Bus, often called the "bus".



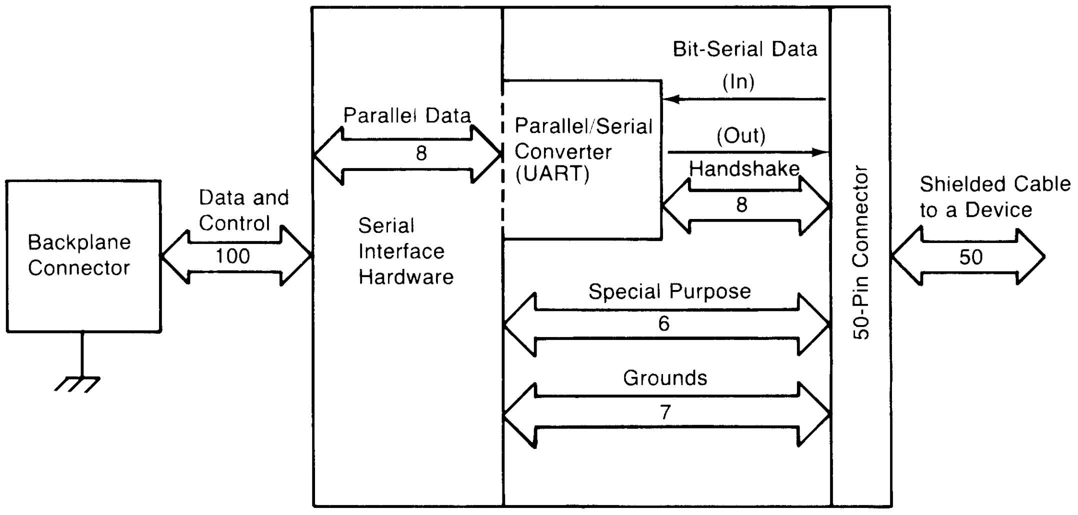
Block Diagram of the HP-IB Interface

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 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.

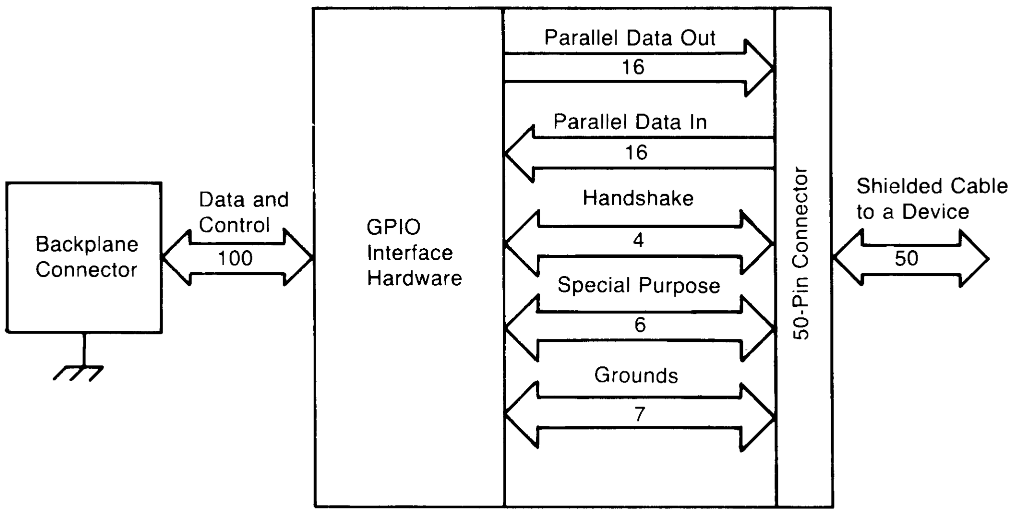


Block Diagram of the Serial Interface

Data is transmitted at several programmable rates using either a simple data handshake or no handshake at all.

The GPIO Interface

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



Block Diagram of the GPIO Interface

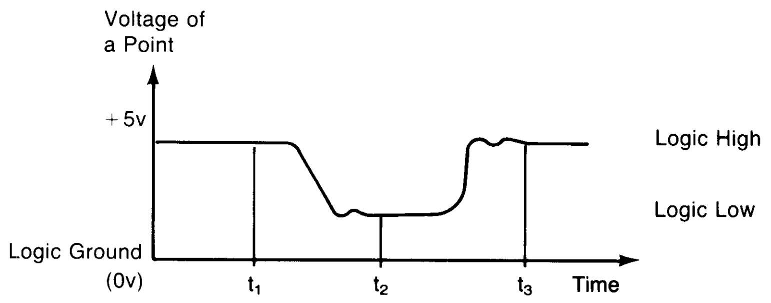
Much of the flexibility of this interface lies in the fact that you have almost direct access to the internal data bus for outputting and entering data.

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 (**bin**ary 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 logic levels to voltage levels.



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, 65536 (=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 * 2 ↑ 0 =   0
1 * 2 ↑ 1 =   2
1 * 2 ↑ 2 =   4      Number represented =
0 * 2 ↑ 3 =   0
1 * 2 ↑ 4 =  16      2 + 4 + 16 + 128 = 150
0 * 2 ↑ 5 =   0
0 * 2 ↑ 6 =   0
1 * 2 ↑ 7 = 128
```

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 9826 (bit 7 = 1). The entire set of the 256 characters on the 9826 is hereafter called the "extended ASCII" character set.

1 ASCII stands for "American Standard Code for Information Interchange". See the Appendix for the complete table.

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.

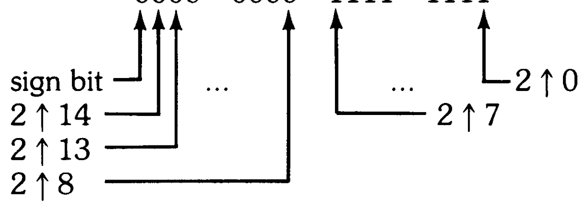
Internal Representation of Integers

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 9826, 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.

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.

Binary representation	Decimal equivalent
1111 1111 1111 1111	−1
0000 0000 0000 0001	1
1111 1111 0000 0001	−255
0000 0000 1111 1111	255



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      0000 0000 0000 0000   all places
                        not used
```

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	1	2	8
Decimal value of character	49	50	56
Binary value of character	00110001	00110010	00111000

Representing Real Numbers

Real numbers, like signed integers, can be represented in one of two ways with the 9826. They are represented in a special binary mantissa-exponent notation within the 9826 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 in the 9826 using a special binary mantissa-exponent notation[1]. With this method, all numbers are represented by a 52-bit, two's-complement, signed mantissa and a 11-bit, biased, signed exponent. Since this is a binary representation, each place value is still an integral power of two. The real number "1/3" is shown below using this representation.

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

exponent sign exponent mantissa sign mantissa

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 9826 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

1 The internal representation used for real numbers is the IEEE standard 64-bit floating-point notation.

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 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

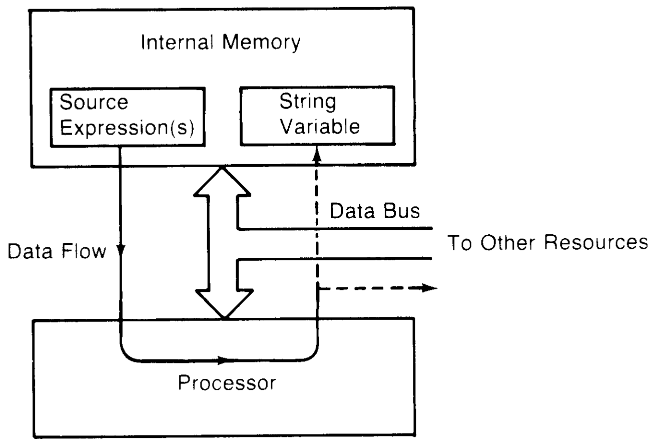
Data can be output to only one resource at a time with the OUTPUT statement (with the exception of the HP-IB Interface). This destination can be any computer resource, which is specified by the destination parameter as shown below.

the destination parameter

```
OUTPUT Destination; String$,CHR$(C+32),"That's all"
```

the source items are expressions

The source of data for output operations is always memoiy. 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.



Data is Copied from Memory to a Resource During Output

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 9826 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.

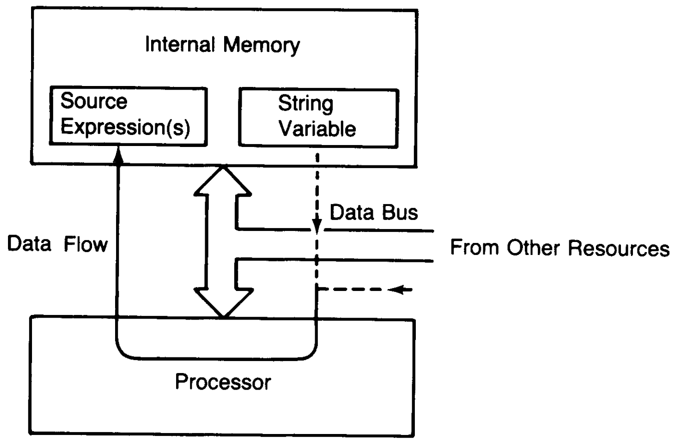
Example Enter Statement

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 source parameter
ENTER Source;Number,String$
```

destination items are program variables

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.



Data is Copied from a Resource to Memory During Enter

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.

Chapter 3
Directing Data Flow

Introduction

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

- Computer memory (string variables in memory)
- Internal and external devices
- Mass storage files

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 Chapter 10.

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$="OUTPUT data"
150    !
160    PRINTER IS 1
170    PRINT "To_dest$ 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.

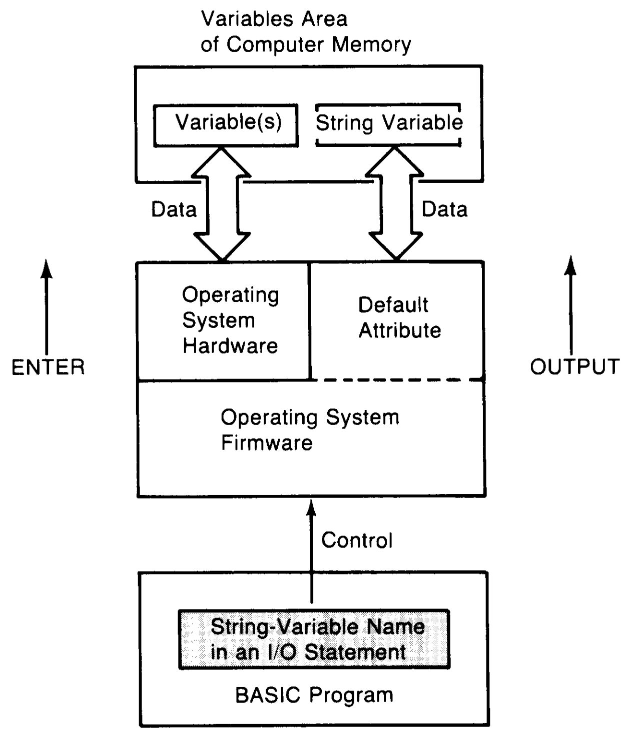


Diagram of the Default I/O Path Used for String-Variable I/O Operations

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 Chapter 10 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. Thus, each device connected to the computer can be accessed through its interface.

Each interface has a unique number by which it is identified, known as its interface select code. The internal devices are accessed with the following, permanently assigned interface select codes.

CRT Display	1
Keyboard	2
Built-in HP-IB.....	7

Optional interfaces all have switch-settable select codes. These interfaces cannot use select codes 1 through 7; the valid range is 8 through 31. 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.

HP-IB	8
Serial	9
GPIO	12

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

```
OUTPUT 1;"Data to CRT"
ENTER 1;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.

HP-IB Device Selectors

Each device on the HP-IB interface has an **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]. Two examples are shown below.

To access the device on:

interface select code 7 at address 01, use device selector 701
interface select code 10 at address 13, use device selector 1013

1 The HP-IB also has additional capabilities that add to this definition of device selectors. See Chapter 11 for further details.

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

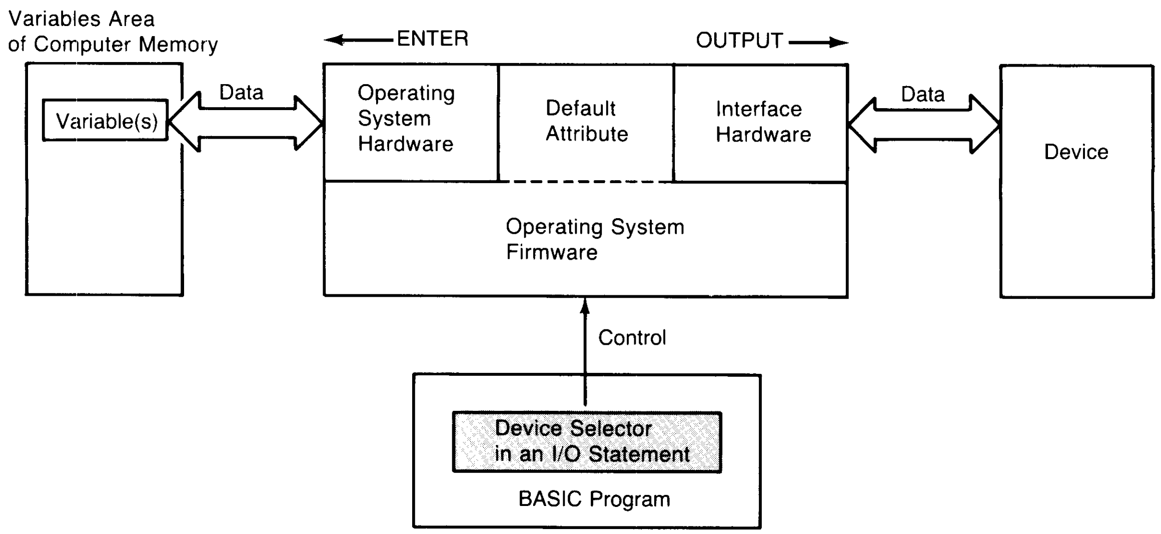

Variables Area of Computer Memory
Variable(s)
Data
ENTER
OUTPUT
Operating System Hardware
Default Attribute
Interface Hardware
Data
Device
Operating System Firmware
Control
Device Selector in an I/O Statement
BASIC Program

Diagram of the Default I/O Path Used when a Device Selector is Specified

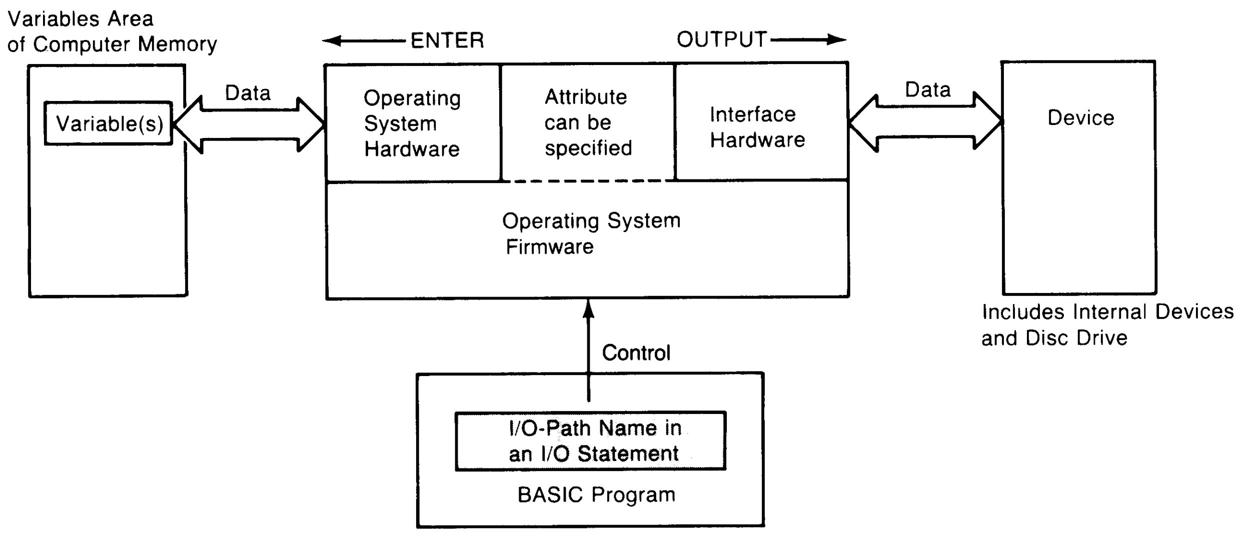
Disc drives are also considered to be devices and are connected to the computer through interfaces. However, files on the disc media cannot be uniquely accessed with only the select code of its interface; additional information specifying which file is to be accessed must be included. Accessing mass storage files is fully described in the BASIC Language Reference and is compared to accessing devices in Chapter 10 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 9826, the I/O paths to devices and mass storage files can be assigned special names; I/O paths to string variables cannot be assigned names. 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".


Variables Area
of Computer Memory
Variable(s)
Data
ENTER
OUTPUT
Operating System Hardware
Attribute can be specified
Interface Hardware
Data
Device
Operating System Firmware
Control
I/O-Path Name in an I/O Statement
BASIC Program
Includes Internal Devices and Disc Drive

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.

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.

```
OUTPUT @Display;"Display message"

OUTPUT @Printer;"Message to the Printer"

ENTER @Serial;Variable,Variable$

ENTER @Gpio;Word1,Word2
```

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.

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 Chapter 6. For now, the important point is that it contains a description of the resource sufficient to allow its access.

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

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 See the *BASIC Language Reference* for further details.

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. 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 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
```

When the subprogram is exited, all I/O path names assigned 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 upon return from a subprogram is shown below.

```
10 CALL Subprogram_x
11 OUTPUT @Log_device;"Main" <---------------- Insert into previous
20 END                                          example.
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 these lines into
2  OUTPUT @Log_device;"First Main" ←───┘  previous example.
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_device) ←───── Add formal parameter.
50 ASSIGN @Log_device TO 1 ! CRT.
60 OUTPUT @Log_device;"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
3   ASSIGN @Log_device TO 701                         statement
4   OUTPUT @Log_device;"First Main"
10  CALL Subprogram_x                                 Parameters
11  OUTPUT @Log_device;"Second Main"                  not necessary
20  END
30  !
40  SUB Subprogram_x
41  COM @Log_device                                   Insert COM
50  ASSIGN @Log_device TO 1 ! CRT.                    statement.
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.

```
              device selector expression
OUTPUT Value_1+BIT(Value_2,5)*2^3;"Data"
```

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 Chapter 10. The second additional factor is described in the next paragraph.

Attribute Control

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 device selectors are used. If the I/O path possesses the attribute of FORMAT OFF, the internal data representation is used. Further details of these attributes are discussed in Chapter 10, "Unified I/O".

The second additional factor that favors using I/O path names is that you can specify the FORMAT attribute to be assigned to the I/O path to devices and to BDAT files. If device selectors are used, this control is not possible. Chapter 10 also describes how to specify the attribute to be assigned to an I/O path.

Chapter 4 Outputting Data

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 fully describes the topic of outputting data to devices; outputting data to string variables and mass storage files is described in Chapter 10 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.

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, the computer 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. 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.

1 The ASCII representation described briefly in Chapter 2 is the default data representation used when communicating with with devices; however, the internal representation can also be used. See Chapter 10 for further details.

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 16 (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
```

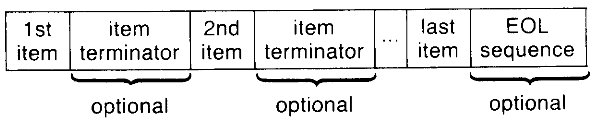
Standard String Format

The internal representation of string data consists of the string characters prefaced by a four-byte header that contains the length of the string (number of characters in the string). The data actually sent consists only of all actual data characters in the string; the length header is **not** output during free-field outputs in which the ASCII representation is being used. Thus, no leading or trailing spaces are output with the string's characters.

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.



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
```

I	t	e	m	CR	LF	–	1	2	3	4	EOL sequence

The default EOL sequence is a CR/LF sequence.

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

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

–	1	2	3	4	,	I	t	e	m	EOL sequence

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",
```

I	t	e	m	CR	LF

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

–	1	2	3	4	,

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

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

I	t	e	m	1	I	t	e	m	2	EOL sequence

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

–	1	2	–	3	4	EOL sequence

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

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

I	t	e	m	1	I	t	e	m	2

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 1;Array(*)   ! No trailing separator.
190    !
200    OUTPUT 1;Array(*), ! Trailing comma.
210    !
220    OUTPUT 1;Array(*); ! Trailing semi-colon.
230    !
240    OUTPUT 1;"Done"
250    END
```

Resultant Output

	1	1	,		1	2	,		1	3	,		2	1	,		2	2	,		2	3	CR	LF
	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	CR	LF	

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$(Row*10+Column)
150        NEXT Column
160    NEXT Row
170    !
180    OUTPUT 1;Array$(*)   ! No trailing separator.
190    !
200    OUTPUT 1;Array$(*),  ! Trailing comma.
210    !
220    OUTPUT 1;Array$(*);  ! Trailing semi-colon.
230    !
240    OUTPUT 1;"DONE"
250    !
260    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	CR	LF
1	1	CR	LF	1	2	CR	LF	1	3	CR	LF	2	1	CR	LF	2	2	CR	LF	2	3	CR	LF
1	1	1	2	1	3	2	1	2	2	2	3	D	O	N	E	CR	LF						

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.

```
1. 100   OUTPUT 1 USING "6A,SDDD.DD";" K= ",123.45

2. 100   Image_str$="6A,DDD.DDD,3X"
   110   OUTPUT 1 USING Image_str$;" K= ",123.45

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

4. 100   OUTPUT 1 USING 110;" K= ",123.45
   110   IMAGE 6A,SDDD.DDD,3X
```

Images

Images are used to specify the desired format of data to be output. Each image consists of groups of individual **image (or "field") specifiers** which either describe the desired format of each item in the source list or specify that special characters are to be output. **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.

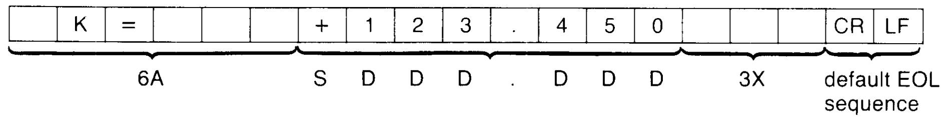
Again, each image list consists of images that each describe the format of a data item to be output. The order of images in the list corresponds to the order of data items in the source list. In addition, image specifiers can be added to output (or to suppress the output of) certain characters. The following example steps through exactly how the computer 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 1 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 groups 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 look over the examples. You are also highly encouraged to experiment with the use of these concepts.

Numeric Images

The digit, sign, and radix image specifiers are used to describe the format of numbers.

Digit, Radix and Exponent Specifiers

Image Specifier	Meaning
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 sign specifier 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 after the decimal point.
	Specifies the position of a decimal point within a number (the American radix). There can be only one decimal point in each numeric image.
E	Specifies that the number is to be output using scientific notation. The "E" must be preceded by at least one digit specifier. 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.

Numeric Examples

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

–	1	2	4	CR	LF

```
OUTPUT @Device USING "ZD";-1.2
```

–	1	CR	LF

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

0	1	.	6	8	CR	LF

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

0	.	4	CR	LF

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

1	2	.	E	+	0	3	CR	LF

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

2	0	.	0	0	E	–	0	5	CR	LF

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

1	2	.	4	CR	LF

Sign Specifiers

These specifiers are used to control the sign information for numeric images. If no sign specifier is included in the image for a negative number, one digit's place will be used for the minus-sign character. Only one or two of these specifiers can be used in any image: for instance, MZZ.DDD and SDD.DDESZZ are both legal images.

Sign Specifiers

Image Specifier	Meaning
S	Specifies a "+" for positive and a "–" for negative numbers.
M	Specifies a leading space for positive and a "–" for negative numbers.

Sign Examples

```
OUTPUT 1 USING "MDD.2D";-12.449
```

–	1	2	.	4	5	CR	LF

```
OUTPUT 1 USING "MDD.DD";2.09
```

		2	.	0	9	CR	LF

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

+	2	.	4	CR	LF

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

+	0	.	4	9	CR	LF

```
OUTPUT 1 USING "SDD.DDE";-2.35
```

–	2	3	.	5	0	E	–	0	1	CR	LF

String Images

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

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	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"
```

C	h	a	r	a	c	t	e	CR	LF

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

A	B	L	i	t	e	r	a	l	CR	LF

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

			H	e	l	l	o				CR	LF

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

			H	e	CR	LF

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.

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 −32768, CHR$(0) is output. If is greater than 32767, CHR$(255) is output. If the destination expects 16 bits of data, the eight most-significant bits are set to 0.
W	Specifies that one word of data (16 bits) are to be output. The source expression is evaluated and rounded to an integer. If it is less than −32768, then −32768 is output; if it is greater than 32767, then 32767 is output. If the interface handles 8-bit data, the most significant byte is sent first; if it handles 16-bit data, all bits are sent in a single operation.

Binary Examples

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

A	B	C	CR	LF

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

CR	CR	LF

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

A	B	CR	LF

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.

Special-Character Specifiers

Image Specifier	Meaning
X	Specifies that a space character, CHR$(32), is to be output.
/	Specifies that a carriage-return, CHR$(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"
```

M					A	CR	LF

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

←(50 spaces)→	CR	LF

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

FF	CR	LF	CR	LF

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

CR	LF	CR	LF

Termination Images

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

Termination Specifiers

Image Specifier	Meaning
L	Specifies that the current end-of-line sequence is to be output (default is a CR/LF sequence).
#	Specifies that the end-of-line 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.

Termination Examples

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

D	a	t	a	CR	LF	CR	LF

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

D	a	t	a

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

FF

Additional Image Features

Several additional features of outputs which use images are available with the 9826. 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
```

0	3	2	8	.	0	3	0	CR	LF

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

D	a	t	a		b	CR	LF

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

					D	a	CR	LF

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

CR	LF	CR	LF	D	a	t	a	CR	LF

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

T	h	e		E	n	d		FF	FF	CR	LF

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

CR	LF	CR	LF	CR	LF

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 1
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 1
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	CR	LF

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

Chapter 5 Entering Data

Introduction

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.
```

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.

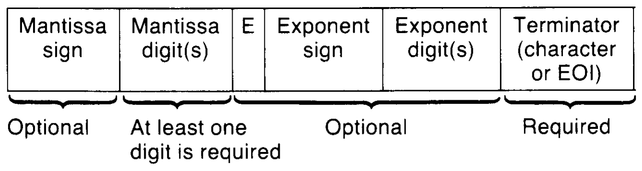
In addition to the termination characters, each item can be terminated (only with selected interfaces) with a special signal known as EOI (End-or-Identify). The EOI signal is only available with the HP-IB, CRT, and keyboard interfaces. EOI termination is further described in the next sections.

Entering Numeric Data

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.

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 "I/O Path Attributes" in Chapter 10 for further details.

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.



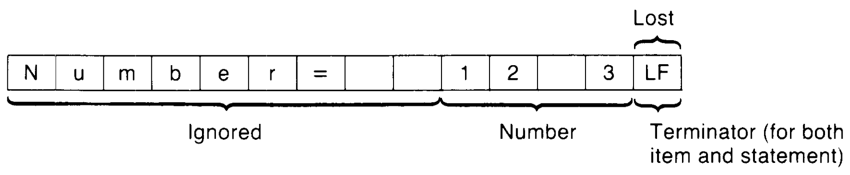
Numeric characters include decimal digits "0" through "9" and the characters ".", "+", "−", "E", and "e". These last five characters must occur in meaningful positions in the data stream to be considered numeric characters; if any of them occurs in a position in which it cannot be considered part of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

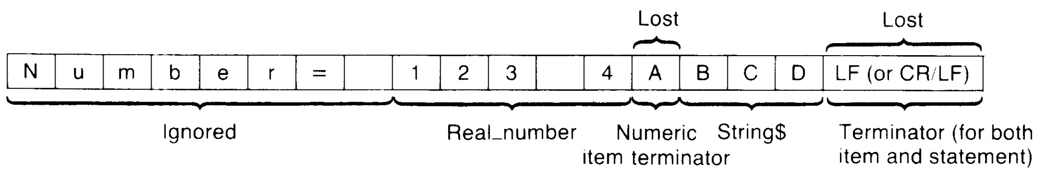


The result of entering the preceding data with the given ENTER statement is that Number receives a value of 123. The line-feed (statement terminator) is **required** since Number is the last item in the destination list.

2. Trailing non-numerics terminate entry into a numeric variable, and the terminating characters (of **both** string and numeric items) are "lost". In this manual, "lost" 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$
```



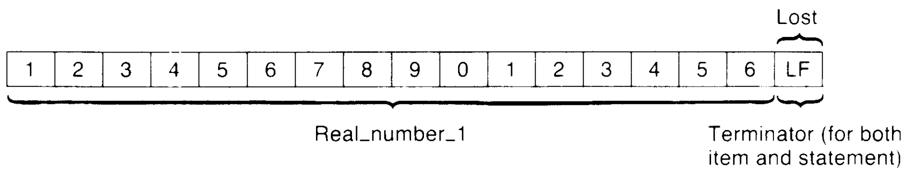
Lost
Lost
N u m b e r = 1 2 3 4 A B C D LF (or CR/LF)
Ignored
Real_number
Numeric item terminator
String$
Terminator (for both item and statement)

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
```

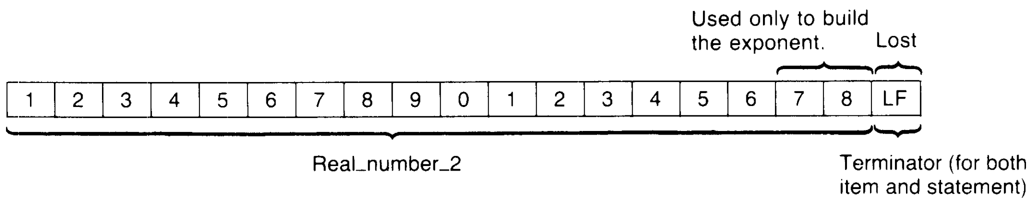


Lost
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 LF
Real_number_1
Terminator (for both item and statement)

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
```

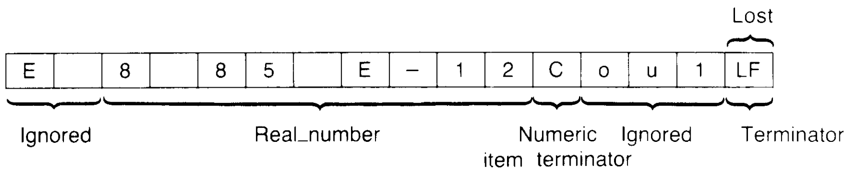

Used only to build the exponent.
Lost
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 LF
Real_number_2
Terminator (for both item and statement)

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
```

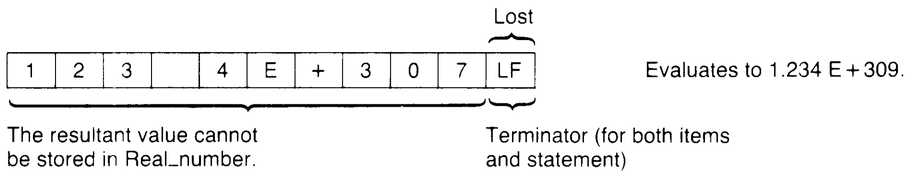


The result of entering the preceding data with the given ENTER statement is that Real_number receives a value of 8.85 E – 12. The character "C" terminates entry into Real_number, and the characters "oul" are entered (but ignored) in search of the required line-feed statement terminator. If the character "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
```


Lost
1 2 3 4 E + 3 0 7 LF
Evaluates to 1.234 E+309.
The resultant value cannot be stored in Real_number.
Terminator (for both items and statement)

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 **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 (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).

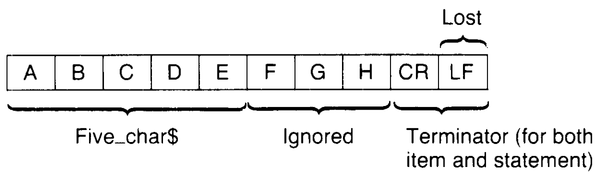
All characters received from the source are entered directly into the appropriate string variable until **any** of the following conditions occurs:

- 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

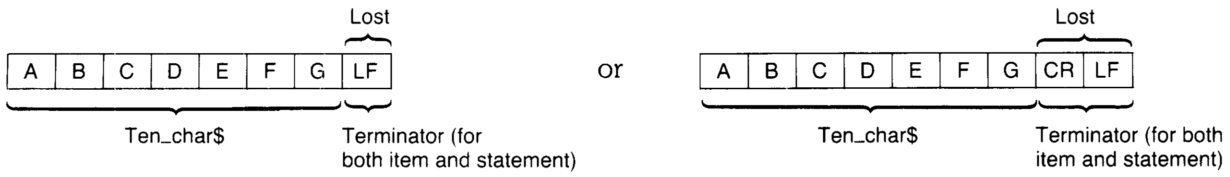
```
ENTER @Device;Five_char$
```



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

Example

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


Lost
A
B
C
D
E
F
G
LF
Ten_char$
Terminator (for both item and statement)
or
Lost
A
B
C
D
E
F
G
CR
LF
Ten_char$
Terminator (for both item and statement)

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 the preceding **statement-termination** condition has **not** occurred **but** entry into the **last item** has been terminated, up to 256 **additional** characters are entered in search of this condition. If it 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.

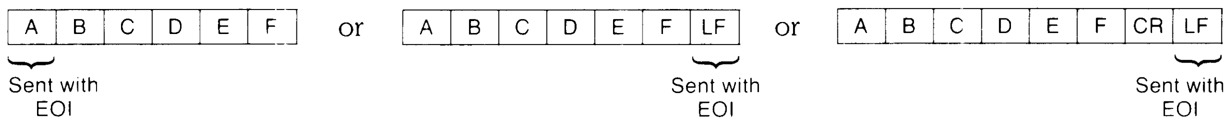
Another case involving numeric data can also occur (see the example given with "rule 4" describing the number builder). If a trailing non-numeric character terminates the last item (which is a numeric variable), additional characters will be entered in search of either a line-feed or a character accompanied 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

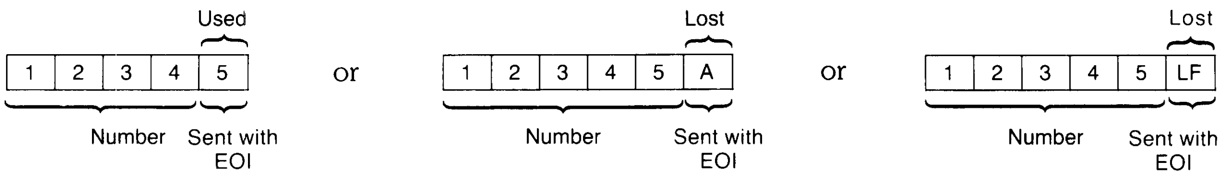
```
ENTER @Device;String$
```



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

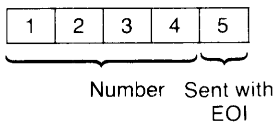
```
ENTER @Device;Number
```



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 Chapter 11 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 keyboard and CRT are contained in Chapters 8 and 9 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

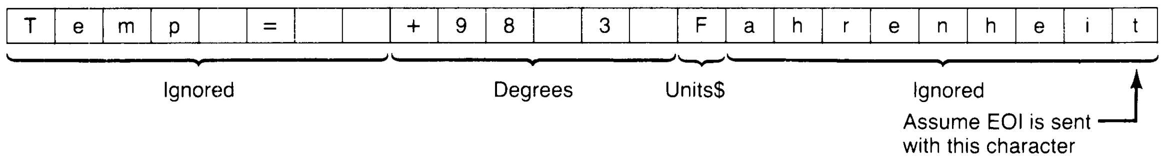
Images are used to specify how data entered from the source is to be interpreted and placed into variables; each image consists of one or more groups of individual image specifiers that determine how the computer will interpret the incoming data bytes (or words). Thus, image lists can be thought of as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


T e m p = + 9 8 3 F a h r e n h e i t
Ignored
Degrees
Units$
Ignored
Assume EOI is sent with this character

Given the above 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 variable's type (REAL or INTEGER).

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

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

Numeric Specifiers

Image Specifier	Meaning
D, Z	Specifies that one byte is to be entered and interpreted as a numeric character. If the character is a non-numeric (including terminators and leading spaces), it will still "consume" one of the digit specifiers in the image.
S, M	Like "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 the image.
	Like "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 the image.
E	Equivalent to "4D" **if** preceded by at least one digit specifier in the image. The same is true for the following three images.
ESZ	Equivalent to "3D" (if preceded by at least one "D", "Z", or "." specifier).
ESZZ	Equivalent to "4D" (if preceded by at least one "D", "Z", or "." specifier).
ESZZZ	Equivalent to "5D" (if preceded by at least one "D", "Z", or "." specifier).
K, -K	Specifies that characters are to be entered and interpreted according to the rules of the number builder.

Examples of Numeric Images

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

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

String Images

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

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	Specifies that string-freefield convention is to be used to enter data into a string variable; characters are entered directly into the variable until a terminating condition (CR/LF, LF, or EOI) is received.
−K	Similar to "K" except that line-feeds do not terminate entry into the string; instead, they are interpreted as string characters and are entered into the string variable. (Receiving EOI with a character or reaching the dimensioned string length terminates the item.)
L @	These specifiers are ignored for enter operations; however, they are allowed for compatibility so that an image can be referenced by **both** ENTER and OUTPUT statements.

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 all characters until the
                                                   string is "full" or EOI 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).

Specifiers Used to Ignore Characters

Image Specifier	Meaning
X	Specifies that a character is to be entered and ignored.
"literal"	Specifies that the number of characters in the literal are to be entered and ignored.
/	Specifies that all characters are to be ignored (i.e., entered but not used) 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.

Binary Specifiers

Image Specifier	Meaning
B	Specifies that one byte is to be entered and interpreted as an integer in the range of 0 through 255.
W	Specifies that one word is to be entered and interpreted as a two's-complement integer; the first byte received is the most significant. If the interface only handles eight bits at a time, the most significant byte will always be all zeros.

Examples of Binary Images

`ENTER @Device USING "B,B,B";N1,N2,N3`	Enter three bytes, then look for LF or EOI terminator.
`ENTER @Device USING "W,K";N,N$`	Enter the first two bytes as an INTEGER, then the rest as string data.

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 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.

1. The EOI signal is received **with** the byte that satisfies the last **image item** or **within 256 bytes after** the byte that satisfied the last image item.
2. 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.

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.

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.

Specifier	Legal Boundary
K, −K	With any character, since this specifies a variable-width field of characters.
S,M,D,E Z,.,A,X "lit" 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.
/	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.

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.
```

Additional Image Features

Several additional image features are available with the 9826. 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$,N2,N2$
```

Chapter 6
Registers

Introduction

A register is a memory location. Thus, 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 the locations at which interface cards reside (remember that the 9826 implements "memory-mapped I/O").

Registers are accessed by the computer when executing I/O statements that specify **either** 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**. The **first** level of access is automatically made by the computer when an I/O statement is executed. The **second** level of access (provided by the STATUS and CONTROL statements) allows interrogating and changing interface and I/O path registers. The **third** level of access (provided by the READIO function and the WRITEIO statement) is used to read from and write to interface hardware **directly**.

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.

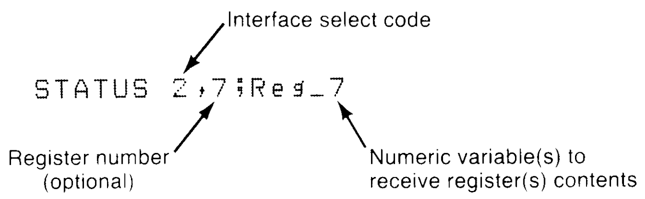
```
100   STATUS 1;Reg_0,Reg_1  ! Pgrm accessing X & Y coords.
110   OUTPUT 1;"Print coordinates before 1st OUTPUT:"
120   OUTPUT 1;"X=";Reg_0,"  Y=";Reg_1
130   OUTPUT 1
140   !
150   OUTPUT 1;"1234567";   ! Note ";".
160   STATUS 1;Reg_0,Reg_1
170   OUTPUT 1
180   OUTPUT 1;"Print coordinates after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



```
STATUS 2,7;Reg_7
```

Interface select code
Register number (optional)
Numeric variable(s) to receive register(s) contents

Example

I/O path STATUS register 0 is read with the following statement. Since the second parameter is optional and has been omitted in this instance, register 0 is accessed.

```
STATUS @Keyboard;Reg_0
```

Example

STATUS registers 4 and 5 of the interface at select code 7 are read with the following statement. Since two numeric variables are to receive register contents, the next highest register is accessed. If more than two variables are specified, successive registers are read.

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

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.

Interface select code

```
100   CONTROL 1;X_pos
```

Numeric expression(s) to be sent
to the appropriate register(s)

Example

Register 8 of interface select code 1 is modified with the following statement. This register's contents determine how many lines of display memory will be "above screen"; changing the contents of this register allows scrolling the display.

```
100   CONTROL 1,8;Line_pos
```

Register number

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 Chapter 3 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, the computer 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, the computer 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.

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:

	Returned Value		Meaning
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

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

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.

Chapter 7 Interface Events

Introduction

The 9826 computer can sense and respond to the occurrence of several 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 *BASIC Language Reference.*

Review 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.

Events

The events that can initiate branches are summarized as follows; only the last two, which are interface events, are discussed in this chapter. The KNOB event is described in Chapter 9, "The Internal Keyboard Interface"; the END, ERROR, and KEY events are described in the *BASIC Language Reference.*

END — occurs when the computer encounters the end of a mass storage file while accessing the file.

ERROR — occurs when a program-execution error is sensed.

KEY — occurs when a currently defined softkey is pressed.

KNOB — occurs when the "knob" (rotary pulse generator) is turned.

INTR — occurs when an interrupt is requested by a device or when an interrupt condition occurs at the interface.

TIMEOUT — occurs when the computer has not detected a handshake response from a device within a specified amount of time.

Service Routines

The software that is executed when an event occurs is called a **service routine**; the service routine takes action that has been programmed as the computer's response to the event. Since most events have only one cause, most service routines execute the same action each time the event occurs. However, if an event can be caused by more than one event, the service routine must also be able to determine **which** event(s) have occurred and then take the appropriate action(s).

Required Conditions

In order for any event to initiate a branch, the following **prerequisite** conditions must be met. Later sections describe how to meet these prerequisites for interface events.

1. The branch must be set up by an ON-event-branch statement, and the service routine must exist.
2. The event must currently be enabled to initiate a branch.
3. The event must occur.
4. The software priority assigned to the event must be greater than the current system priority[1].

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

A Simple Example

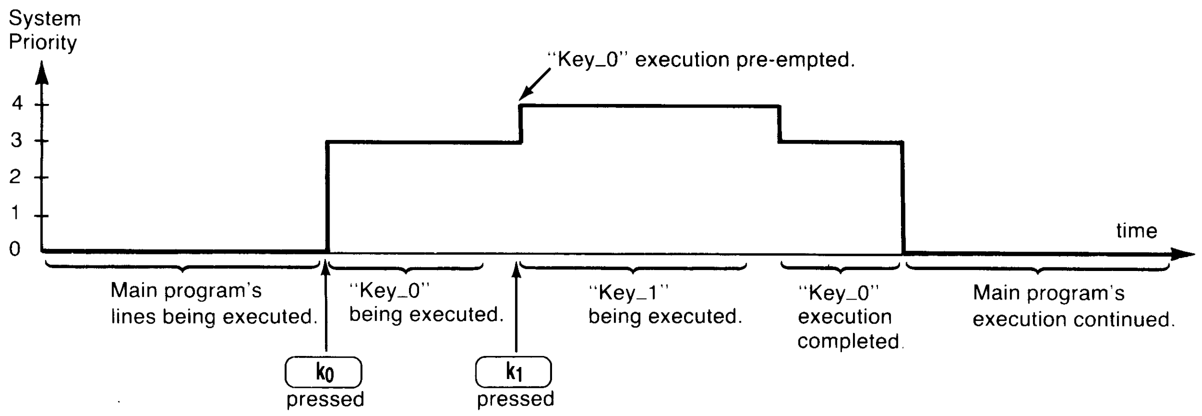
The following program shows how events (of different software priorities) are serviced by the computer. Subprograms called "Key_0" and "Key_1" are the service routines for the events of **k0** and **k1** being pressed; the software priorities assigned to these events are 3 and 4, respectively. Run the 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.

```
100 ON KEY 0,3 CALL Key_0     ! Set up events and
110 ON KEY 1,4 CALL Key_1     ! assign priorities.
120 !
130 Low_tone=100
140 Mid_tone=300
150 Hi_tone=400
160 !
170 !
180 OUTPUT 1;" System","Priority"
190 V$=CHR$(8)&CHR$(10)       ! BS & LF.
200 OUTPUT 1;"    4"&V$&"3"&V$&"2"&V$&"1"&V$&"0"
210 !
```

1 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. For further details of hardware priority, see the next sections of this chapter.

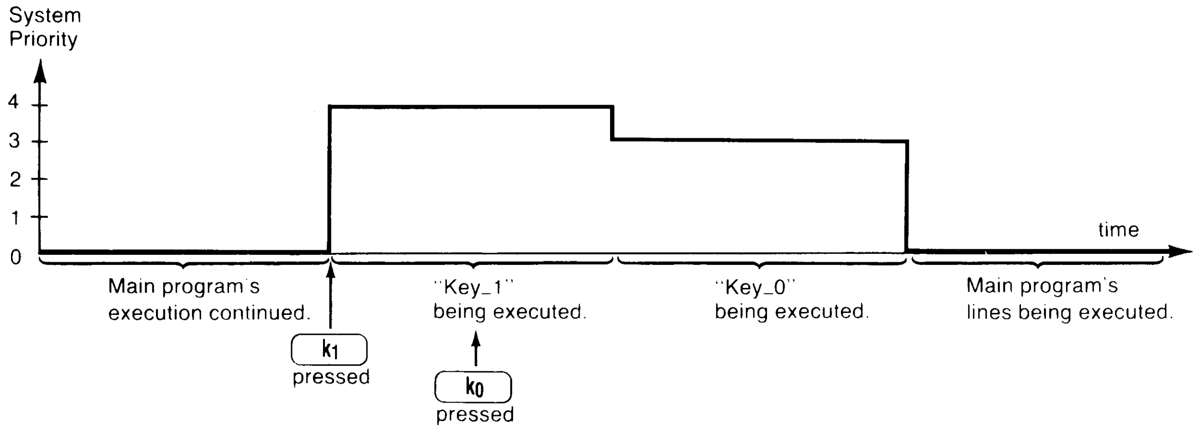
```
220 Main: CALL Bar_graph(7,"*") ! Sys. prior. is
230                             ! always >= 0.
240       BEEP Low_tone,.1
250       FOR Jiffy=1 TO 5000
260       NEXT Jiffy
270       !
280   GOTO Main               ! Main loop.
290   !
300   END
310   !
320 SUB Key_0
330     CALL Bar_graph(4,"*") ! Plot priority.
340     BEEP Mid_tone,.1
350     FOR Iota=1 TO 2000
360     NEXT Iota
370     CALL Bar_graph(4," ") ! Erase.
380   SUBEND
390   !
400 SUB Key_1
410     CALL Bar_graph(3,"*") ! Graph priority.
420     BEEP Hi_tone,.1
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 **k1** is pressed **after k0 but while** the Key_0 routine is being executed, execution of Key_0 is **temporarily interrupted** and the Key_1 routine is executed. When Key_1 is finished, execution of Key_0 is resumed at the point where it was temporarily interrupted. This occurs because **k1** was assigned a **higher software priority** than **k0**.



Events with Higher Software Priority Take Precedence

On the other hand, if **k0** is pressed **while k1** is being serviced, the computer finishes executing Key_1 **before** executing Key_0. The event of pressing **k0** 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.



An Event with Lower Software Priority Must Wait

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 computer; priority increases from 0 to 16.

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

Software Priorities of Events

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

In the above example, system priority was 0 before either of the events occured. When **k0** was pressed, the system priority became 3. When **k1** was subsequently pressed, the system first logged the event and then checked its priority against the current system priority. Since **k1** had been assigned a priority of 4, it pre-empted **k0**'s service routine because of its higher software priority.

It is important to **note that the computer 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 section further describes logging and servicing events.

When execution of Key_1 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_1 completed, the Key_0 service routine had the highest software priority, so its execution was resumed at the point at which it was interrupted.

If **k0** 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).

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, while 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.**

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 the 9826. Only the optional interfaces' hardware priorities can be changed.

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

Hardware Priorities of 9826 Interfaces

Hardware Priority	Interface(s) and Event(s) at This Priority
0	(Quiescent level; no interface is currently interrupting)
1	Internal Keyboard (KEY and KNOB events)
2	Internal Disc Drive (END event)
3	Internal HP-IB (INTR and TIMEOUT events)
3-6	Optional Interface Cards (INTR and TIMEOUT events)
7	Non-Maskable Interrupts (such as the RESET 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 the computer 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, the computer begins servicing all pending events.

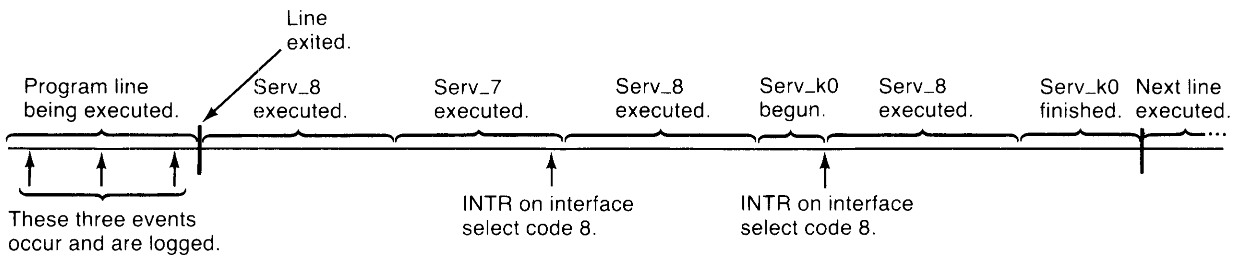
When servicing pending events, the computer begins with the event of highest software priority and executes the event of lowest software priority last. However, if two or more events have the **same software priority**, the computer services the events **in order of descending interface select codes**. 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. First, events of higher hardware priority will interrupt the current activity to be logged by the computer. Second, 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_KO    ! Hardware priority 1.
```

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


Line exited.
Program line being executed.
Serv_8 executed.
Serv_7 executed.
Serv_8 executed.
Serv_k0 begun.
Serv_8 executed.
Serv_k0 finished.
Next line executed.
These three events occur and are logged.
INTR on interface select code 8.
INTR on interface select code 8.

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. This **delay of polling** the interfaces at lower hardware priorities helps these urgent events get the immediate attention they may require. Decreasing the system's response time to these urgent events may also **increase overall system throughput**.

Setting up Branches

Again for review, the methods of setting up an event-initiated branch to a service routine are as follows for **all** events.

1. `ON` event `CALL` subprogram name
2. `ON` event `GOSUB` service routine
3. `ON` event `GOTO` service routine
4. `ON` event `RECOVER` service routine

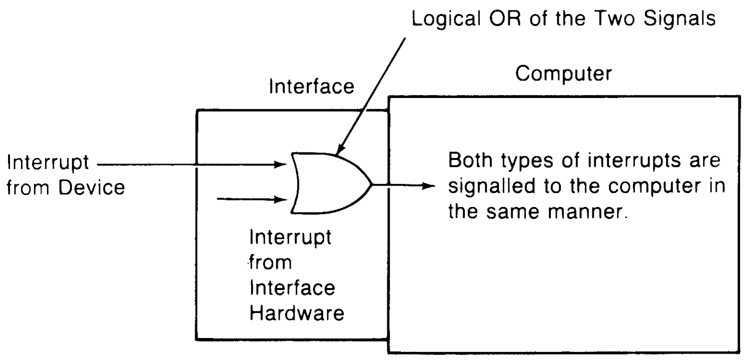
The term **service routine** is any legal branch location for the type of branch specified and current context. The *BASIC Language Reference* fully describes the differences between these types of branches.

Enabling Events to Initiate Branches

Before an event (which is set up) can initiate a branch, it must first be enabled to do so. The power-up state of the computer is that the END, ERROR, KEY, KNOB and TIMEOUT events are already enabled to initiate branches; the INTR events must be **enabled explicitly** with separate statements. Further details of enabling these events are described in the "Interface Interrupts" and "Interface Timeouts" sections of this chapter.

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.


Logical OR of the Two Signals
Interface
Computer
Interrupt from Device
Both types of interrupts are signalled to the computer in the same manner.
Interrupt from Interface Hardware

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

Setting Up Interrupt Events

Both of the preceding types of interrupt-initiated branches are set up with statements such as those in the following examples.

Example

Set up an interrupt event to be logged, and define the location and software priority of the service routine.

```
ON INTR Int_sel_code,Priority CALL Service_routine
```

The select code of the interface is specified by the first parameter; I/O path names **cannot** be used to specify the interface. The second parameter specifies the software priority assigned to the event. A subrogram called Service_routine must exist in computer memory at the time the program is run. Parameters cannot be passed to the service routine in the ON INTR statement; any variables to be used jointly by the service routine and other contexts must be defined in common. See the *BASIC Language Reference* for further details.

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

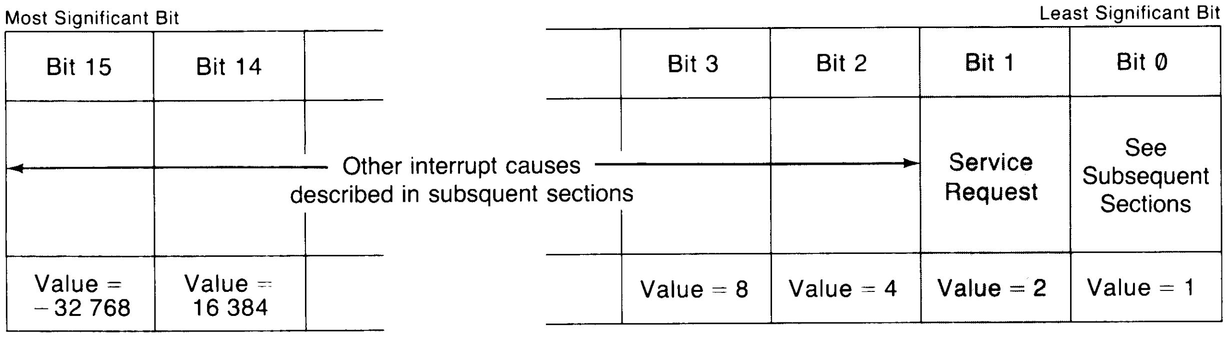
Enable interrupts occuring at interface select code 7 to initiate the branch set up by an ON-event-branch statement.

```
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
```



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** (all interrupt events **disabled**) is used.

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.

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. 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 with the other interfaces.
5. The actual requested action is performed (line 250).
6. If subsequent events are to be enabled to initiate branches, they must be 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 here.

```
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, the time in which the computer responds to the event's occurrence is only accurate to within +/- 25% of the specified time, even though the resolution of the time parameter is 0.001 seconds.

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 IO** key. This key aborts the I/O operation that was left "hanging" by the failure of the device to respond to 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.

Chapter 8
The Internal CRT Interface

Introduction

This chapter describes programming techniques for "interfacing" the computer to the internal CRT. Access to this device with I/O statements (OUTPUT, ENTER, STATUS, and CONTROL) is described herein. Many of the concepts and programming techniques presented in the previous chapters of this manual are applied in this chapter.

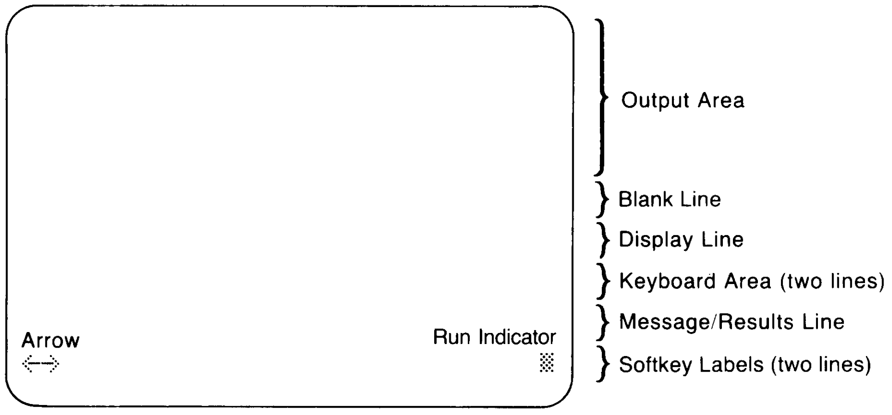
CRT Display Description

The CRT is accessed through the interface permanently assigned to select code 1. This display features the following capabilities.

- Both alphanumerics and graphics information can be displayed on this device either simultaneously or separately[1]. The alphanumeric display consists of 25 lines by 50 columns of dot-matrix characters.
- Characters OUTPUT to the CRT appear in the top 18 display lines (known as the output area).
- All character positions in display memory can be addressed individually, and the display can be scrolled up and down under program control.
- Characters can be read from any location of the display with the ENTER statement. The EOI signal (simulated) is sent with the line-feed following the last non-blank character in the line.

1 Programming the graphics display is described in the *BASIC Language Reference.*

The 25 lines of the alphanumeric display are organized as follows.



The Output Area and the Disp Line

The alphanumeric display is divided into several areas which are used for different purposes. Characters sent to the CRT with the PRINT statement appear in the top 18 lines of the display, known as the CRT's "output area". Characters sent to the CRT with the DISP statement appear in the "DISP line". Type in and run the following example program to see these two different areas.

```
100    PRINTER IS 1
110    !
120    FOR Line=1 TO 18
130        PRINT "The OUTPUT Area"
140    NEXT Line
150    !
160    DISP "The DISP Area"
170    !
180    END
```

Output to the CRT

Data can also be sent to the output area by directing OUTPUT statements to interface select code 1. The following example uses an I/O path name to direct the data to the CRT; the default data representation used with the CRT is the ASCII representation.

```
100    ASSIGN @Printer TO 1
110    !
120    FOR Line=1 TO 18
130        OUTPUT @Printer;"The OUTPUT Area"
140    NEXT Line
150    !
160    END
```

Numeric Outputs

When numbers are output to the CRT by the free-field form of the OUTPUT statement, the standard numeric format is used[1]. The following statements show how trailing punctuation within the OUTPUT statement affects the item terminators output after each numeric item.

Examples **Results**

```
OUTPUT 1;123,456        123, 456
OUTPUT 1;-123,456      -123, 456
OUTPUT 1;-123,-456     -123,-456
OUTPUT 1;-123;-456     -123-456
OUTPUT 1;123;456        123 456

leading "+" signs
replaced by a space
```

String Outputs

Strings are output to the CRT in a similar manner with free-field outputs; trailing punctuation in the statement determines whether or not string-item and statement terminators are output. The following examples show how trailing punctuation within the OUTPUT statement affects the output of string-item terminators.

Examples **Results**

```
OUTPUT 1;"One","Two"      One
                          Two
OUTPUT 1;"Three";"Four"   ThreeFour
```

As with free-field outputs to other devices, a trailing semicolon causes the separator of the item that it follows to be suppressed. In the above case, the carriage-return and line-feed separators which normally follow the output of a string item are suppressed by the semicolon. The next paragraphs describe how the carriage-return and line-feed (control characters) are interpreted by the CRT.

1 "Standard numeric format" is further described in Chapter 4, "Outputting Data".

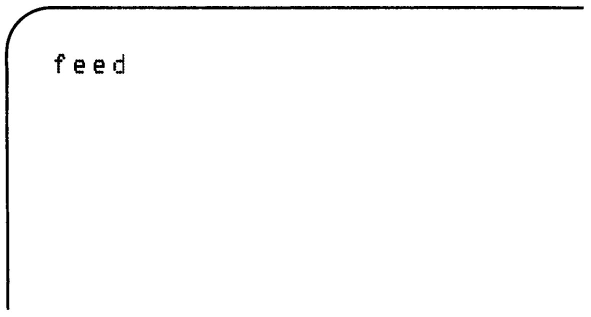
Control Characters

ASCII characters with codes 0 through 31 are defined to be "control" characters. When one of these characters is sent to a system resource, it is usually interpreted as a **command**, rather than as data. The complete list of control characters and their corresponding codes and definitions is given in the ASCII table in the Appendix.

Four of these characters are used for controlling the 9826's CRT display, and all others are ignored (i.e., are not displayed and cause no special action when received by the CRT). Run the following program and note the results.

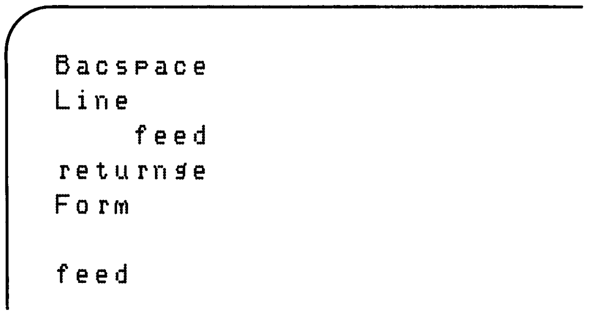
```
130    Backspace$=CHR$(8)
140    Line_feed$=CHR$(10)
150    Form_feed$=CHR$(12)
160    Carriage_return$=CHR$(13)
170    !
180    !
190    ASSIGN @Crt TO 1
200    !
210    OUTPUT @Crt;"Back";
220    WAIT 1
230    OUTPUT @Crt;Backspace$;"space"
240    WAIT 1
250    !
260    OUTPUT @Crt;"Line";
270    WAIT 1
280    OUTPUT @Crt;Line_feed$;"feed"
290    WAIT 1
300    !
310    OUTPUT @Crt;"Carriage";
320    WAIT 1
330    OUTPUT @Crt;Carriage_return$;"return"
340    WAIT 1
350    !
360    OUTPUT @Crt;"Form";
370    WAIT 1
380    OUTPUT @Crt;Form_feed$;"feed"
390    DISP "Scroll down to view previous display"
400    !
410    END
```

Display Before Scroll



```
feed
```

Display After Scroll



```
Bacspace
Line
    feed
returnge
Form

feed
```

The following table describes the display functions invoked when the specified control character is sent to the CRT (in the "Display functions off" mode). The **print position** is the column and line at which the next character sent to the display will appear.

Control-Character Functions on the CRT

Character	Value	Defined Action
Bell	7	Causes beeper to output the standard tone; no display action is invoked.
Backspace (BS)	8	If the print position was not in column 1, it is moved "back" one character position; if it was in the first column, no action is invoked.
Line-feed (LF)	10	Moves the print position "down" one line.
Form-feed (FF)	12	Scrolls the screen "up" as far as possible, prints two blank lines, and places the print position at column 1 of the second, printed blank line.
Carriage-return (CR)	13	Causes the print position to be moved to the beginning (first column) of the current screen line.
All other control characters.		Ignored.

The Display Functions Mode

The preceding program showed the control characters which are defined to invoke a special display function when sent to the CRT. To display **all** control characters sent to the CRT, rather than have the CRT interpret them as commands, turn the Display functions mode **on** by pressing the **DISPLAY FCTNS** key. Repeatedly pressing this key toggles this display mode between "on" and "off". Using the CRT with Display functions on is very useful when you need to see exactly which control characters have been output.

Except for the carriage-return character, all subsequent control characters sent to the display (while in this mode) do not invoke their defined function, but are **only displayed**. The carriage-return is **both** displayed **and** causes the print position to move to the beginning of the next line (both CR and LF functions invoked).

The Display functions mode can also be enabled from BASIC programs with the use of the CONTROL statement. The following program shows how this is accomplished. Notice that the carriage-return invokes both carriage-return and line-feed functions.

```
100    CONTROL 1,4;1      ! Non-zero => set.
110    !
120    ! First send with default CR/LF sequence.
130    OUTPUT 1;"DISPLAY FUNCTIONS ON"
140    !
150    ! Then suppress the CR/LF (with ";").
160    OUTPUT 1;CHR$(12);
170    END
```

Notice that the "Display functions on" message normally displayed when the **DISPLAY FCTNS** key is pressed is not automatically displayed when the value of CONTROL register 4 is changed; instead, the program must display the message, if so desired.

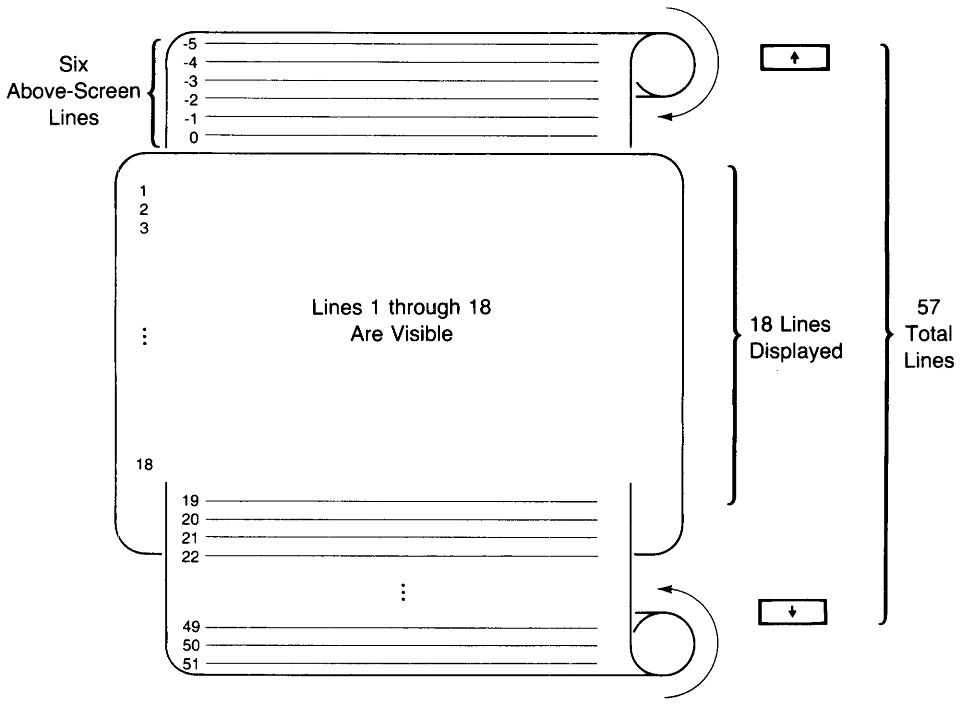
Output-Area Memory

In addition to the 18 visible display lines in the output area, there are 39 additional lines (57 lines total) available within output-area memory. These additional lines of display memory can be viewed by running the following program and then scrolling the display down (turning the knob counterclockwise). The 18 **visible** lines of output-area memory will hereafter be called the **screen**.

```
100    ! Example to show scrolling.
110    !
120    PRINTER IS 1                  ! PRINT on CRT.
130    !
140    FOR Line=1 TO 57              ! Write 57 lines.
150        PRINT Line
160    NEXT Line
170    !                        Now scroll manually.
180    END
```

Determining Above-Screen Lines

Scrolling the display up and down allows you to view different 18-line portions of the (up to) 57 lines within output-area memory. If the display is scrolled **down** as far as possible (i.e., the first 18 lines of output-area memory are visible), there are no lines "above screen". Similarly, if the display is scrolled **up** as far as possible, there are (up to) 39 lines above screen. The following drawing shows 6 lines above screen.



Line Positions of the Output Area

The number of lines that are above screen can be determined from BASIC programs by reading STATUS register 3 of interface select code 1. The returned value is the number of lines currently **above screen**.

If the screen has just been cleared (**SHIFT-CLR LN**), the following program displays `0 lines above screen`. Running the program a second time displays `18 lines above screen`, and so forth until `39 lines above screen` is displayed continually.

```
100  FOR Line=1 TO 18
110       OUTPUT 1;Line
120  NEXT Line
130  !
140  STATUS 1,3;Lines_above
150  DISP Lines_above;" lines above screen"
160  END
```

Screen Addresses

All of the characters in output-area memory can be addressed individually by the character's screen column and line. The character in the upper left corner of the screen is in column 1 and line 1, and the character in the lower right corner is in column 50 and line 18. The addresses of the characters "off screen" are limited by the number of lines currently above screen.

The screen addresses (both column and line) at which a subsequent character sent to the display will appear on the screen are known as the **print position**. The current print position is automatically changed as characters are output to the display. For instance, the print-position column is incremented each time a character is sent; when the 51st character is sent to a line, the print-position column is reset to 1 and the print-position line is incremented, sending the character to the next line. The following program shows how the print-position line varies during output to the CRT.

```
100   FOR Line=1 TO 57
110       OUTPUT 1;Line
120       STATUS 1,1;Print_line
130       DISP "Print-position line = ";Print_line
140       IF Line<25 THEN WAIT .2
150   NEXT Line
160   !
170   END
```

Notice that **the print-position line is always relative to the first line of the current screen**. This accounts for the print position (read with STATUS) remaining at a value of 19 while the 19th through 57th lines are being printed. When the print position is **off screen**, the display is scrolled (when it receives a character) so that the character appears on the screen. When the display is finished scrolling, all line addresses are again relative to the **new** top screen line. The next section describes using this feature to scroll the display from the program.

Scrolling the Display

The program can scroll the display up and down by changing the print position to a location **off screen** and then outputting character(s) to the CRT. Thus, in order to **scroll up**, values greater than 18 must be written to register 1. If the screen is to be scrolled up 4 lines, the following statements can be used. In this case, the OUTPUT statement outputs the "Null" control character so that no characters will be overwritten.

```
100    CONTROL 1,1;18+4     !  Move print position off screen;
110    OUTPUT 1;CHR$(0);    !  scrolling takes place when next
120                         !  character sent to the CRT.
```

The screen is not scrolled up until the OUTPUT statement actually writes to the CRT at the current print position (even though, in this case, no visible character is actually output to the display).

In order to **scroll down**, a non-positive number must be written into register 1. For instance, to scroll down one line, a 0 would be written into register 1. Again, the display is not actually scrolled until an OUTPUT (or PRINT) to the CRT is executed.

The only **restriction** on the value of the line number is that it must not attempt to scroll the screen down **past** the first line of output-area memory. In other words, to scroll down as far as possible, the following value would be used; using smaller values results in an error.

Top line's address = − (number of lines above screen) + 1

Thus, if no lines are above screen, the top line's address is 1.

An example of scrolling down "as far as possible" is shown in the following program.

```
100    FOR Line=1 TO 57
110        OUTPUT 1;Line
120    NEXT Line
130    !
140    STATUS 1,1;Line_pos
150    DISP "Print-position line =";Line_pos;" after OUTPUT,"
160    WAIT 2
170    !
180    STATUS 1,3;Lines_above  ! Find # lines above screen.
190    DISP "and";Lines_above;" lines are above screen"
200    WAIT 3
210    !
220    CONTROL 1,1;-Lines_above+1   ! Change line-pos.
230    OUTPUT 1;"Line 1"            ! Scroll made when 1st
240                                 ! character is sent.
250    !
260    STATUS 1,3;Lines_above
270    DISP "Now, number of lines above screen =";L_above
280    END
```

Entering from the CRT

Data is entered from the CRT beginning at the current print position. As characters are read from the screen (from left to right), the print position is updated. When the ENTER statement attempts to read past the last non-blank character on a line, the CRT display's hardware sends a line-feed character accompanied by a (simulated) EOI signal, and the print position is advanced to the beginning of the next line.

Reading a Screen Line

The following program uses the line-feed accompanied by EOI to terminate entry into a string variable. Since the free-field ENTER statement is used, only one line can be read because of the EOI sent with the line-feed character.

```
100   CONTROL 1;5,8         ! Move print position to
110                         ! 5th column of line 8,
120   OUTPUT 1;"ABCDEFGH"   ! then OUTPUT (with CR/LF).
130   !
140   OUTPUT 1;"IJKLMNOP      " ! OUTPUT to line 9 with
150                             ! trailing spaces.
160                             !
170   CONTROL 1,1;8         ! Move print position back
180                         ! to 1st column of line 8.
190                         !
200   ENTER 1;Line_8$
210   DISP LEN(Line_8$);"characters read from line 8"
220   WAIT 2
230   !
240   ENTER 1;Line_9$
250   DISP LEN(Line_9$);"characters read from line 9"
260   END
```

This feature of the CRT is very useful when simulating entry from the HP-IB interface; however, keep in mind that **no spaces can be read after the last visible character at the end of each line**. Notice in the preceding example that the trailing space characters sent to the display were **not** read back by the ENTER statement. These trailing characters are treated as "blanks" by the CRT, which sends the line-feed with EOI when the ENTER statement attempts to read the first one.

Reading the Entire Output-Area Memory

In order to read all lines within output-area memory, an ENTER statement that uses an image must be used to prevent the EOI signal from terminating the statement prematurely (since the EOI signal acts as an **item** terminator during ENTER-USING-image statements which contain no "%" image specifiers). The following program shows the entire contents of output-area memory being read.

```
100   OPTION BASE 1
110   DIM Memory$(57)[50] ! To read all 57 lines.
120   !
130   FOR Screen_line=1 TO 57
140       OUTPUT 1;"Line";Screen_line
150   NEXT Screen_line
160   WAIT 1
170   !
180   STATUS 1,3;Lines_above
190   CONTROL 1;1,-Lines_above+1   ! Scroll to read
200   ENTER 1 USING "K";Memory$(*) ! entire memory.
210   !
220   FOR Screen_line=1 TO 57      ! Display all lines;
230       PRINT Memory$(Screen_line);" "; ! no CR/LF.
240   NEXT Screen_line
250   END
```

Final Display

```
Line 49
Line 50
Line 51
Line 52
Line 53
Line 54
Line 55
Line 56
Line 57
Line 1 Line 2 Line 3 Line 4 Line 5 Line 6 Line 7 L
ine 8 Line 9 Line 10 Line 11 Line 12 Line 13 Line
14 Line 15 Line 16 Line 17 Line 18 Line 19 Line 20
Line 21 Line 22 Line 23 Line 24 Line 25 Line 26 L
ine 27 Line 28 Line 29 Line 30 Line 31 Line 32 Lin
e 33 Line 34 Line 35 Line 36 Line 37 Line 38 Line
39 Line 40 Line 41 Line 42 Line 43 Line 44 Line 45
Line 46 Line 47 Line 48 Line 49 Line 50 Line 51 L
ine 52 Line 53 Line 54 Line 55 Line 56 Line 57
```

Notice that the print position was moved to the top line before attempting to read memory contents, since the ENTER statement reads characters beginning at the print position. If the print position is not at the "top line" of memory before attempting to read all 57 lines, the lines above screen will not be read. However, the statement executes with no errors, because the CRT sends line-feeds (with EOI) for each line that does not really exist "below screen". For instance, if the print position is at line 10 when the ENTER begins, only the last 47 lines of output-area memory will be read (and placed into the first 47 elements of Memory$). When the ENTER statement attempts to fill last ten elements of Memory$, the CRT sends only line-feeds accompanied by EOI because the print position is past the last non-blank character.

Additional CRT Features

This section describes the remainder of features of the CRT display controllable by BASIC programs. **Interrupt and timeout events are not available with the CRT interface.**

The DISP Line

BASIC programs can output characters to the DISP Line with the DISP statement, as described in the *BASIC Language Reference.* As with the output-area's print position, the position (column) within the DISP line at which subsequent characters will appear can be read and changed explicitly by BASIC programs. This **DISP-line position** can be read and changed with STATUS register 8 and CONTROL register 8 (of interface select code 1), respectively.

```
100   FOR Disp_pos=46 TO 1 STEP -1
110       CONTROL 1,8;Disp_pos
120       DISP "HELLO"
130       WAIT .2
140   NEXT Disp_pos
150   END
```

Keep in mind that if trailing carriage-return and line-feed characters are output to the DISP line, the carriage-return returns the DISP-line position to column 1. A subsequent DISP statement clears the entire line. However, if these trailing characters are suppressed, the DISP-line position is left unchanged. Run the following program to see these effects.

```
100   PRINT "First with trailing CR/LF,"
110   DISP "HI"
120   WAIT .5
130   DISP " THERE"
140   WAIT 1
150   !
160   PRINT "then with no CR/LF."
170   DISP "HI";
180   WAIT .5
190   DISP " THERE"
200   END
```

Also notice that if a DISP attempts to send characters to the DISP line so that any character will be past the 50th column, the entire line is shifted left so that all of the new characters will be displayed (i.e., so that the last character written will end up in column 50).

```
100   CONTROL 1,8;40
110   DISP "CHARACTERS";   ! No CR/LF.
120   WAIT 1
130   DISP " SHIFTED LEFT"
140   !
150   END
```

Disabling the Cursor Character

BASIC programs even have control over whether any cursor is displayed (during all computer modes, such as during EDITs and other keyboard-entry modes). The cursor is **disabled** with the following statement.

```
CONTROL 1,10;0
```

Any **non-zero** value written to this register **re-enables** the cursor to be displayed. Resetting the computer also re-enables the cursor being displayed.

```
CONTROL 1,10;1
```

Enabling the Insert Mode

The insert mode of the keyboard area can be enabled and disabled with STATUS and CONTROL statements. If any **non-zero** numeric value is written to register 2, the insert mode is **enabled**. All subsequent characters typed into this area are "inserted" between the cursor and the character to its immediate left, and characters to its right are shifted appropriately.

The following program turns insert mode on for approximately five seconds. During this time, use the knob to move the cursor left and right while typing in characters from the keyboard.

```
100  Insert_mode=1
110  CONTROL 1,2;Insert_mode
120  !
130  DISP "Insert mode is now being used"
140  BEEP 200,.2
150  WAIT 5
160  !
170  Insert_mode=0
180  CONTROL 1,2;Insert_mode
190  DISP "Now the mode has changed to overwrite"
200  BEEP 100,.2
210  WAIT 5
220  !
230  BEEP 50,.2
240  DISP "Program ended"
250  END
```

Summary of CRT STATUS and CONTROL Registers

STATUS Register 0	— Current print position (column)
CONTROL Register 0	— Set print position (column)
STATUS Register 1	— Current print position (line)
CONTROL Register 1	— Set print position (line)
STATUS Register 2	— Insert-character mode
CONTROL Register 2	— Set insert character mode if non-0
STATUS Register 3	— Number of lines "above screen".
CONTROL Register 3	— Undefined
STATUS Register 4	— Display functions mode
CONTROL Register 4	— Set display functions mode if non-0
STATUS Register 5	— Undefined
CONTROL Register 5	— Undefined
STATUS Register 6	— ALPHA ON flag
CONTROL Register 6	— Undefined
STATUS Register 7	— GRAPHICS ON flag
CONTROL Register 7	— Undefined
STATUS Register 8	— Display line position (column)
CONTROL Register 8	— Set display line position (column)
STATUS Register 9	— Screenwidth (number of characters)
CONTROL Register 9	— Undefined
STATUS Register 10	— Cursor-enable flag
CONTROL Register 10	— Cursor-enable; 0 = cursor visible. non-0 = cursor visible.
STATUS Register 11	— CRT character mapping flag
CONTROL Register 11	— Disable CRT character mapping if non-0

Chapter 9
The Internal Keyboard Interface

Introduction

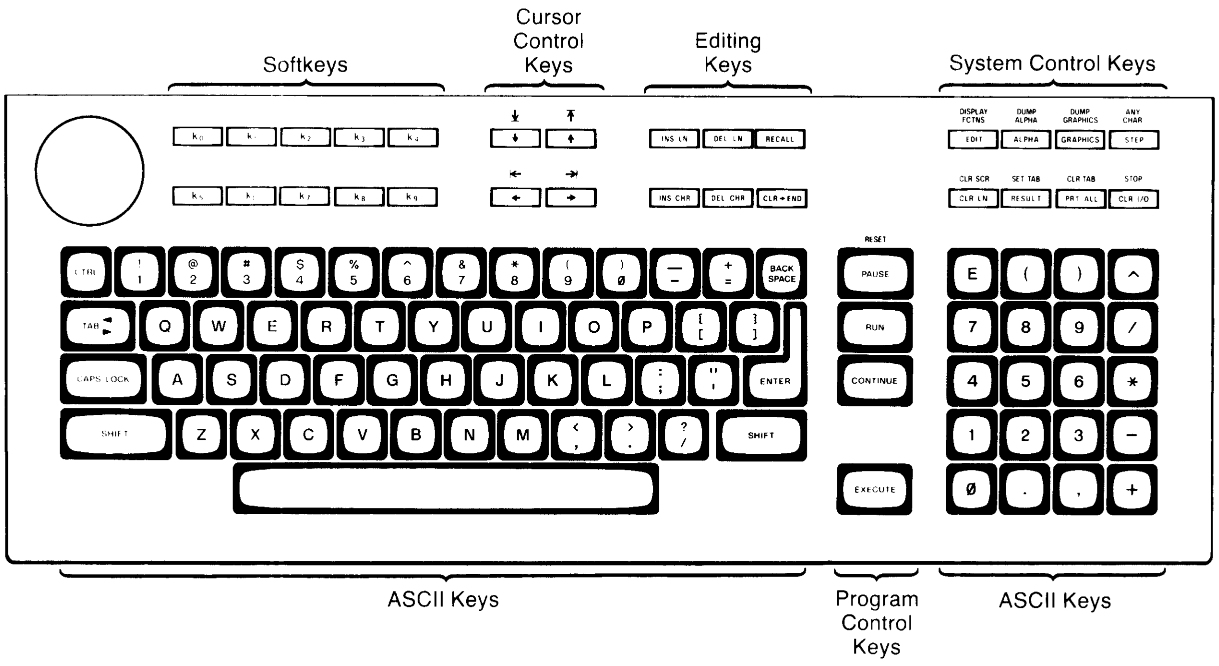
As with the CRT, access to the internal keyboard can be made with the OUTPUT, ENTER, STATUS, and CONTROL statements. This chapter describes programming techniques for "interfacing" to this internal device.

Keyboard Description

The internal (or built-in) keyboard of the 9826 is controlled by its own processor, allowing many more capabilities than most other desktop-computer keyboards. This keyboard is a device that resides at interface select code 2 and provides the following capabilities.

- Data can be entered using the ENTER statement, allowing simulation of other devices and program debugging.
- Commands can be output to the keyboard to simulate an operator; data can be output to the keyboard allowing the operator to edit the data.
- The keyboard processor maintains a real-time clock, which can be read by BASIC programs.
- The processor monitors the "knob" (the rotary pulse generator) and can periodically interrupt the program when the knob is turned.
- The processor controls the programmable beeper.

The INTR and TIMEOUT interface events **cannot** be sensed by the keyboard.



ASCII and Non-ASCII Keys

The keys of the 9826 can be generally grouped by function into the ASCII and non-ASCII keys. The **ASCII** (or alphanumeric) keys all **produce an ASCII character when pressed**, and include the character entry and numeric keys. The **non-ASCII** (or non-alphanumeric) keys do not produce characters but **initiate specific action** when pressed; the **ENTER**, **CAPS LOCK**, **TAB**, and **BACK SPACE** keys are non-ASCII keys for this reason. Non-ASCII keys also include all program control, editing, cursor control, and system control keys.

The Shift and Control Keys

The **SHIFT** and **CTRL** keys are not really either type of key because neither can cause action on its own; instead, they are used only with the other types of keys. Pressing the **SHIFT** key with another key **qualifies** the other keypress, allowing the other key to have a second meaning. For instance, in the "Caps lock off" mode, pressing an alphabetic ASCII key generates a lowercase alphabetic character. Pressing the **SHIFT** key **simultaneously** with an alphabetic key in the "Caps lock off" mode generates an uppercase character. The **SHIFT** key is used similarly with the non-ASCII keys, allowing many of those keys to have a second function.

The **CTRL** (Control) key is also used to further qualify **both** ASCII and non-ASCII keypresses. Pressing the **CTRL** key simultaneously with an ASCII key generates an ASCII control character in the display, and is often faster than using the **ANY CHAR** key. The following table shows how to generate control characters by simultaneously pressing the **CTRL** key and typing key(s).

Generating Control Characters with CTRL and ASCII Keys

Key Code	ASCII Character	Character's Description	Key(s) Pressed with CTRL	Character on CRT
0	NUL	Null	(space bar)	$^{N}{}_{U}$
1	SOH	Start of Header	A	$^{S}{}_{H}$
2	STX	Start of Text	B	$^{S}{}_{X}$
3	ETX	End of Text	C	$^{E}{}_{X}$
4	EOT	End of Transmission	D	$^{E}{}_{T}$
5	ENQ	Enquiry	E	$^{E}{}_{Q}$
6	ACK	Acknowledgement	F	$^{A}{}_{K}$
7	BEL	Bell	G	
8	BS	Backspace	H	$^{B}{}_{S}$
9	HT	Horizontal Tab	I	$^{H}{}_{T}$
10	LF	Line-Feed	J	$^{L}{}_{F}$
11	VT	Vertical Tab	K	$^{V}{}_{T}$
12	FF	Form-Feed	L	$^{F}{}_{F}$
13	CR	Carriage-Return	M	$^{C}{}_{R}$
14	SO	Shift Out	N	$^{S}{}_{O}$
15	SI	Shift In	O	$^{S}{}_{I}$
16	DLE	Data Link Escape	P	$^{D}{}_{L}$
17	DC1	Device Control	Q	$^{D}{}_{1}$
18	DC2	Device Control	R	$^{D}{}_{2}$
19	DC3	Device Control	S	$^{D}{}_{3}$
20	DC4	Device Control	T	$^{D}{}_{4}$
21	NAK	Neg. Acknowlegdement	U	$^{N}{}_{K}$
22	SYN	Synchronous Idle	V	$^{S}{}_{Y}$
23	ETB	End of Text Block	W	$^{E}{}_{B}$
24	CAN	Cancel	X	$^{C}{}_{N}$
25	EM	End of Media	Y	$^{E}{}_{M}$
26	SUB	Substitute	Z	$^{S}{}_{B}$
27	ESC	Escape	[	$^{E}{}_{C}$
28	FS	File Separator	SHIFT - [	$^{F}{}_{S}$
29	GS	Group Separator	]	$^{G}{}_{S}$
30	RS	Record Separator	↑	$^{R}{}_{S}$
31	US	Unit Separator	SHIFT - /	$^{U}{}_{S}$

The keys listed in the preceding table are **not the only** ways to generate control characters, but are generally the simplest and most easily memorized method. For instance, to generate a line-feed character, press the **CTRL** and the **J** keys simultaneously; alternate methods are also shown below.

CTRL - J or CTRL - SHIFT - J or CTRL - SHIFT - 8 or CTRL - *

Pressing the **CTRL** key with a non-ASCII key is used to generate and store non-ASCII keystrokes within strings and is further discussed in "Outputs to the Keyboard".

Keyboard Operating Modes

The keyboard has two operating modes which can be changed either manually by pressing the **CAPS LOCK** or the **PRT ALL** key or from the program with the CONTROL statement. This section describes changing these modes from the program.

The Caps Lock Mode

Pressing the **CAPS LOCK** key toggles the keyboard between the "Caps lock on" and "Caps lock off" modes. In the "Caps lock on" mode, pressing any alphabetic key causes an uppercase letter to be displayed on the screen; in the "Caps lock off" mode, these keys generate lowercase letters. This mode can be changed with the CONTROL statement and sensed with the STATUS statement. Writing **any non-zero** numeric value into register 0 (of interface select code 2) sets the caps lock mode **on**; writing a zero into this register sets the mode off.

```
100   STATUS 2;Caps_lock  ! Check mode.
110   !
120   PRINT "Initially, ";
130   IF Caps_lock=1 THEN
140       Mode$="ON"
150   ELSE
160       Mode$="OFF"
170   END IF
180   !
190   PRINT "CAPS LOCK was "&Mode$&CHR$(10) ! Skip line.
200   BEEP
210   WAIT 1
220   !
230   CONTROL 2;1
240   PRINT "CAPS LOCK now ON"
250   PRINT "Type in a few characters"&CHR$(10)
260   WAIT 4
270   !
280   CONTROL 2;0
290   PRINT "CAPS LOCK now OFF"
300   PRINT
310   BEEP
320   END
```

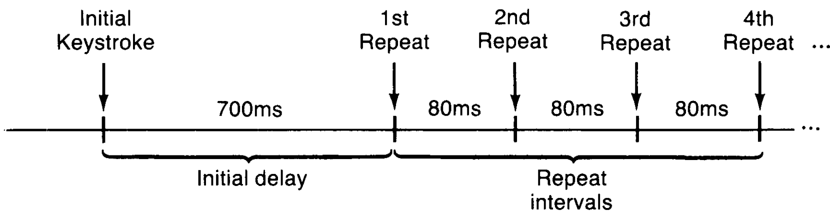
The Print All Mode

Pressing the **PRT ALL** key toggles the "Print all" mode "on" and "off". The "Print all" mode can also be sensed and changed by reading and writing to STATUS register 1 and CONTROL register 1 (of interface select code 2). Writing a **non-zero** numeric value into this register sets the "Print all" mode **on**; writing a value of zero turns this mode "off". The following statement turns the "Print all" mode off.

```
CONTROL 2,1;0
```

Modifying the Repeat and Delay Intervals

The keyboard has an auto-repeat feature which allows you to hold a key down to repeat its function rather than pressing and releasing it repeatedly. Holding a key down will cause it to be repeated every 80 milliseconds for as long as it is is held down, resulting in a repeat rate of approximately 12.5 characters per second. However, you may have noticed that the initial delay between the key being pressed and the key being repeated is longer than successive delays between repeats; the initial delay before a key is repeated for the first time is 700 milliseconds (7/10 second). The following plot of a key's **default** repeat function shows these two intervals.



These intervals can be changed from the program, if desired, by writing different values into CONTROL registers 3 and 4 (of interface select code 2). Register 3 contains the parameter that controls the auto-repeat interval, and register 4 contains the parameter that controls the initial delay. The values of these parameters, multiplied by 10, give the respective intervals in milliseconds with the following exception; if register 3 is set to 256, the auto-repeat is disabled.

The following program sets up softkeys 0, 4, 6, and 8 to change these parameters. Run the program and experiment with these intervals to optimize them for your own preferences and needs.

```
100   ON KEY 0 LABEL "Faster" GOSUB Decr_interval
110   ON KEY 4 LABEL "Slower" GOSUB Incr_interval
120   ON KEY 6 LABEL "Sooner" GOSUB Decr_delay
130   ON KEY 8 LABEL "Later" GOSUB Incr_delay
140   !
150   Interval=80  ! Defaults.
160   Delay=700
170   !
180   DISP "Interval=";Interval;" Delay= ";Delay
190   GOTO 180    ! Loop.
200   !
210 Incr_interval:Interval=Interval+10*(Interval<2560)
220              CONTROL 2,3;Interval/10
230              RETURN
240              !
250 Decr_interval: Interval=Interval-10*(Interval<>10)
260               CONTROL 2,3;Interval/10
270               RETURN
280               !
```

```
290 Incr_delay: Delay=Delay+10*(Delay<2560)
300             CONTROL 2,4;Delay/10
310             RETURN
320             !
330 Decr_delay: Delay=Delay-10*(Delay>10)
340             CONTROL 2,4;Delay/10
350             RETURN
360             !
370       END
```

Entering from the Keyboard

When the keyboard is specified as the source of data in an ENTER statement, the computer executes the process just as if entering data from any other device. The computer signals to the keyboard that the keyboard is to be the sender of data. The keyboard in turn signals that it is not ready to send data and waits for you to type in and edit the desired data.

The characters you type in appear in the keyboard area of the display, but they are not automatically sent to the computer. As long as you can see the characters, you can edit them before sending them to the computer, **just as during an INPUT statement**. Available characters include all 256 characters that can be generated either with keystrokes or with the **ANY CHAR** key.

Pressing the **ENTER**, **STEP**, or **CONTINUE** key signals the keyboard that the data is to be sent to the computer. The data is then sent byte-serially according to an agreed-upon handshake convention. The computer enters the data in byte-serial fashion and processes it according to the specified variable(s), type of ENTER statement, and image (if it is an ENTER USING statement).

The differences in pressing the **ENTER**, **STEP**, and **CONTINUE** keys are as follows. Keep in mind that the ENTER statement is still being executed as long as the "?" appears in the lower right corner of the display.

ENTER or **STEP** All of the characters displayed in the keyboard area are sent to the computer, followed by carriage-return and line-feed characters. These last two characters **usually** terminate entry into the current item in the ENTER statement. In addition, the **STEP** key causes the computer to remain in the single-step mode after the ENTER statement has been completely executed.

CONTINUE All of the characters displayed in the keyboard area are sent to the computer for processing; **no** trailing carriage-return and line-feed characters are sent. The **CONTINUE** key is pressed if more characters are to be entered into the current variable in the destination list of the ENTER statement.

Type in and run the following program. Experiment with how entry into each variable item is terminated by using the different keys (i.e., the **CONTINUE** key versus the **ENTER** or **STEP** keys). Pressing the **ENTER** or **STEP** key terminates entry into the current variable, while pressing the **CONTINUE** key allows additional characters to be entered into the current variable.

```
100    DIM String_array$(1:3)[100]
110    ASSIGN @Device_simulate TO 2
120    !
130    ENTER @Device_simulate;String_array$(*)
140    !
150    OUTPUT 1;String_array$(*)
160    !
170    END
```

This use of the keyboard is very powerful when tracing the cause of an error in an enter operation. With this tool, you can "debug" or verify any type of ENTER statement, including ENTER statements whose source is intended to be a device on the HP-IB interface. The next section describes this topic.

Sending the EOI Signal

The EOI signal is implemented on the HP-IB interface. This line ordinarily signals to the computer that the data byte being received is the last byte of the item; thus, it is either an item terminator or a terminating condition for the ENTER statement[1].

The EOI signal can be simulated from the keyboard when this feature is properly enabled. CONTROL register 12 of interface select code 2 controls this feature; the following example statement shows how to enable this feature.

```
CONTROL 2,12;1
```

To simulate the EOI signal with a character, the **CTRL** and **E** keys are pressed **simultaneously** before the character to be accompanied with EOI is typed in. For instance, if the characters "DATA" are to be entered and the EOI is to accompany the last "A", the following key sequence should be pressed before pressing the **ENTER**, **STEP**, or **CONTINUE** key.

D A T CTRL-E A

The same result can be obtained by placing an ENQ character (ASCII control character CHR$(5),$^{E}{}_{Q}$) in front of the character to be accompanied by the EOI signal (see the previous section for further details).

1 See Chapter 5 for further explanation of the EOI signal's effects during ENTER.

Outputs to the Keyboard

Characters output to the keyboard are indistinguishable from characters typed in from the keyboard. All characters output to the keyboard, **including control characters**, are displayed in the keyboard area. The following program outputs the BEEP statement to the keyboard.

```
80     ! "KBD_2"
90     !
100    OUTPUT 2;"BEEP"; ! No CR/LF.
110    !
120    END
```

Sending Non-ASCII Keystrokes to the Keyboard

The preceding program sent the characters BEEP to the keyboard, but the statement was **not executed**. Pressing the **EXECUTE** key after the program has ended executes the statement. Modify the program to "press" the **EXECUTE** key by typing in **CTRL-EXECUTE** following the BEEP. Sending this special two-character sequence to the keyboard is equivalent to the operator pressing the **EXECUTE** key. Thus, **in general**, to store a non-ASCII "keystroke" within a program line, press the **CTRL** key while simultaneously pressing the desired non-ASCII key.

Since CHR$(255) does not generate the same character on most printers as it does on the 9826 display, it is recommended that some explicit means of documenting these character sequences be employed. For instance, string variables can be defined to contain these sequences; then when the program is listed on an external printer, it will be much easier to determine which non-typing keys are being represented. The **CTRL** key is still used with the non-ASCII key to generate the two-character sequence, but the special character should be changed to a CHR$(255).

```
100    Execute_Key$=CHR$(255)&"X"
110    Printall_Key$=CHR$(255)&"A"
120    !
130    OUTPUT 2;Printall_Key$; ! Use ";" to suppress CR/LF.
140    OUTPUT 2;"BEEP"&Execute_Key$;
150    END
```

Note

Since this type of output can be used to send immediately executed commands (such as SCRATCH A) it is very important that you be very cautious when outputting commands to the keyboard. It is also advised that you use care when editing statements and commands sent to the keyboard due to the two-character non-ASCII key sequences; unexpected results may occur when carelessly editing non-ASCII key sequences output by a program.

The following table shows the resultant characters that follow CHR$(255) in the two-character sequences generated by these keystrokes. The table is included only to show the general mnemonic nature of the second character in these sequences. The next table can be used to look up which non-ASCII key is to be output if the second character is known.

Mnemonic Nature of Non-ASCII Key Sequences

Key	Character	Key	Character	Key	Character
		Softkeys		**Cursor Controls**	
k0	0	k10	a	↓	V
k1	1	k10	b	↑	^
k2	2	k12	c	←	<
k3	3	k13	d	→	>
k4	4	k14	e	SHIFT - ↓	T
k5	5	k15	f	SHIFT - ↑	W
k6	6	k16	g	SHIFT - ←	H
k7	7	k17	h	SHIFT - →	G
k8	8	k18	i		
k9	9	k19	j	**Editing Keys**	
		System Keys		INS LN	*
				DEL LN	/
EDIT	D	DISPLAY FCTNS	F	RECALL	?
GRAPHICS	L	DUMP GRAPHICS	N	SHIFT - RECALL	@
ALPHA	M	DUMP ALPHA	O	INS CHR	+
STEP	S	ANY CHAR	$	DEL CHR	-
				CLR→END	%
CLR LN	#	CLR SCR	K		
RESULT	=	SET TAB	]	**Character Entry**	
PRT ALL	A	CLR TAB	[		
CLR I/O	I	STOP	!	TAB	)
				SHIFT - TAB	(
		Program Controls		CAPS LOCK	U
				ENTER	E
PAUSE	P	RUN	R	Roman	Y
CONTINUE	C	EXECUTE	X	Katakana	J

Look-Up Table for Non-ASCII Key Sequences

Character	Key	Character	Key	Character	Key
space	[1]	@	SHIFT - RECALL	`	[1]
!	STOP [2]	A	PRT ALL [2]	a	k10 [2]
"	[1]	B	BACK SPACE	b	k11 [2]
#	CLR LN	C	CONTINUE [2]	c	k12 [2]
$	ANY CHAR [2]	D	EDIT	d	k13 [2]
%	CLR→END	E	ENTER [2]	e	k14 [2]
&	[1]	F	DISPLAY FCTNS [2]	f	k15 [2]
'	[1]	G	SHIFT - →	g	k16 [2]
(	SHIFT - TAB	H	SHIFT - ←	h	k17 [2]
)	TAB	I	CLR I/O	i	k18 [2]
*	INS LN [2]	J	Katakana Mode[2]	j	k19 [2]
+	INS CHR	K	CLR SCR [2]	k	[1]
,	[1]	L	GRAPHICS [2]	l	[1]
-	DEL CHR	M	ALPHA [2]	m	[1]
.	Ignored	N	DUMP GRAPHICS [2]	n	[1]
/	DEL LN [2]	O	DUMP ALPHA [2]	o	[1]
0	k0 [2]	P	PAUSE [2]	p	[1]
1	k1 [2]	Q	[1]	q	[1]
2	k2 [2]	R	RUN [2]	r	[1]
3	k3 [2]	S	STEP [2]	s	[1]
4	k4 [2]	T	SHIFT - ↓ [2]	t	[1]
5	k5 [2]	U	CAPS LOCK [2]	u	[1]
6	k6 [2]	V	↓ [2]	v	[1]
7	k7 [2]	W	SHIFT - ↑ [2]	w	[1]
8	k8 [2]	X	EXECUTE [2]	x	[1]
9	k9 [2]	Y	Roman Mode[2]	y	[1]
:	[1]	Z	[1]	z	[1]
;	[1]	[	CLR TAB	{	[1]
<	←	\	[1]	|	[1]
=	RESULT	]	SET TAB	}	[1]
>	→	^	↑ [2]	~	[1]
?	RECALL	_	[1]	▒	[1]

1 These characters cannot be generated by pressing the CTRL key and a non-ASCII key. If one of these characters follows CHR$(255) in an output to the keyboard, an error is reported (`Error 131  Bad non-alphanumeric keycode`).

2 Processing of these keys, known as "closure keys", is described in the following section.

Closure Keys

Several of the non-ASCII keys are known as "closure keys". Closure keys are so named because of the way the computer processes these keys when output to the keyboard. The important feature of closure keys is that **the computer can only process two closure keys between program lines during a running program**. If more than two appear in the data output to the keyboard, the additional keys may be processed in an unexpected order.

As an example, the following program sends four closure keys to the keyboard with a single OUTPUT statement. Only the first two closure keys are processed **after** this OUTPUT statement (but **before** `DISP "Next BASIC line"` is executed). The third and fourth closure keys are processed after `DISP "Next BASIC line"` is executed (but before `DISP "2nd BASIC line"` is executed). This accounts for the following display after running the program, since the "Printall" command was not executed until after `DISP "Next BASIC line"` was executed.

```
100   !   Define non-ASCII Keys.
110   Ex$=CHR$(255)&"X"   ! EXECUTE Key.
120   Up$=CHR$(255)&"^"   ! Up arrow Key.
130   Prt$=CHR$(255)&"A"  ! PRT ALL Key.
140   !
150   CONTROL 2,1;0    ! Turn PRINTALL off.
160   CONTROL 1,1;1    ! Begin on top screen line.
170   OUTPUT 1;"Line 1"
180   OUTPUT 1;"Line 2"
190   OUTPUT 1;"Line 3"
200   WAIT 1
210   !
220   !   Now send statement with 4 closure keys.
230   OUTPUT 2;"DISP ""Hello""";Ex$;Up$;Up$;Prt$;
240   DISP "Next BASIC line"  ! PRT ALL still off.
250   DISP "2nd BASIC line"   ! Now PRT ALL is on.
260   !
270   END
```

Display After Running Program

```
Line 3
2nd BASIC line










2nd BASIC line


Printall on
```

In addition, if the last character sent to the keyboard is a CHR$(255), the next character typed in by the user will give unexpected results. Again, it is important to exercise care when using this feature.

Locking Out the Keyboard

There are certain times during program execution when it is expedient to prevent the operator from using the keyboard, such as during a critical experiment which cannot be disturbed. The the knob and groups of keyboard keys can be enabled and disabled separately.

Setting bit 0 of register 7 (of interface select code 2) **disables** all keys (excluding the **RESET** key) and the knob. The following program first sets up the KNOB and KEY events to initiate program branches. It is assumed that the keyboard is already enabled; if you are not sure, press the **RESET** key. When the program is run, the keyboard and knob remain enabled for about five seconds, after which they are disabled. The program then displays the time of day indefinitely; the only way to stop the program is to press the **RESET** key.

```
100   ON KEY 0 LABEL "SFK 0" GOSUB Key0
110   ON KNOB .2 GOSUB Knob
120   !
130   PRINT "You've got 5 seconds.  GO! "
140   FOR Iteration=1 TO 20
150       WAIT .25
160   NEXT Iteration
170   !
180   Reset_disable=0  ! RESET remains ENABLED.
190   Ky_Knb_disable=1 ! DISABLE rest of Kbd.
200   CONTROL 2,7;2*Reset_disable+Ky_Knb_disable
210   PRINT "Time's up!"
220   BEEP
230   !
240       SET TIME 0
250 Loop: DISP DROUND(TIMEDATE MOD (24*60*60.),4)
260       GOTO Loop
270       !
280       !
290 Key0:    PRINT "Special function key 0 pressed."
300          RETURN
310          !
320 Knob:   PRINT "Knob rotation sensed."
330         RETURN
340   END
```

If the value of the variable RESET_disable is set to 1 in the preceding program, the only way to **prematurely stop** the program is to turn off power to the 9826, losing the program and all data currently in computer memory.

Note

Use care when locking out **both** the **RESET** key and the keyboard keys. If both are locked out, the **only** way to prematurely stop the program is to turn the computer off.

Sensing Knob Rotation

The "event" of the knob (rotary pulse generator) being rotated can be sensed by the program. The branch location, interval at which the computer interrogates the knob for the occurrence of rotation, and branch priority are set up with a statement such as the following.

```
ON KNOB Interval,Priority CALL Knob_turned
```

In addition to the program being able to sense rotations of the knob, it can also determine how many pulses the knob has produced and whether or not either or both of the **CTRL** or **SHIFT** keys are being pressed. This ability to "qualify" the use of the knob allows it to be used for up to four different purposes. The following program shows how to set up the branch, how to determine the number of pulses, and how to determine the direction of rotation.

```
100    ON KNOB .25 GOSUB Knob ! Check Knob every 1/4 sec.
110    !
120    FOR Iteration=1 TO 200
130        WAIT .1
140        DISP Iteration
150    NEXT Iteration
160    !
170    STOP
180    !
190 Knob:  STATUS 2,10;Key_with_Knob
200        PRINT KNOBX;" pulses ";
210        IF Key_with_Knob=0 THEN
220          PRINT ! CR/LF.
230        ELSE
240          IF Key_with_Knob=1 THEN PRINT "with SHIFT"
250          IF Key_with_Knob=2 THEN PRINT "with CTRL"
260          IF Key_with_Knob=3 THEN PRINT "with SHIFT and
 CTRL"
270        END IF
280        RETURN
290    END
```

The interval parameter of 0.25 seconds was specified in the preceding program; consequently, the knob will be interrogated approximately every 0.25 seconds. If any pulses have occurred since the last interrogation, the specified branch will be initiated.

One full rotation of the knob produces 120 pulses. The service routine calls the KNOBX function to determine how many pulses (only **net** rotation) have been generated **since the last call** to this function. If the number is positive, a net clockwise rotation has occurred; a negative number signifies that a net counterclockwise rotation has occurred. Since the pulse counter can only sense +128 to −127 pulses **during the specified interval**, the interval parameter should be chosen small enough to interrogate the knob before the pulse counter reaches one of these values. **Experiment** with this parameter to adjust it for your particular application.

Summary of Keyboard STATUS and CONTROL Registers

STATUS Register 0	— CAPS LOCK flag
CONTROL Register 0	— Set CAPS LOCK if non-0
STATUS Register 1	— PRINTALL flag
CONTROL Register 1	— Set PRINTALL if non-0
STATUS Register 2	— Undefined
CONTROL Register 2	— Undefined
STATUS Register 3	— Undefined
CONTROL Register 3	— Set auto-repeat interval. If 1 thru 255, repeat interval in milliseconds is 10 times this value. 256 = turn off auto-repeat. The power-on default is 8, causing repeat intervals of 80 milliseconds.
STATUS Register 4	— Undefined
CONTROL Register 4	— Set delay before auto-repeat. If 1 thru 256, delay in milliseconds is 10 times this value. The power-on default is 70, causing a delay before auto-repeat of 700 milliseconds.
STATUS Register 5	— Undefined
CONTROL Register 5	— Undefined
STATUS Register 6	— Undefined
CONTROL Register 6	— Undefined

STATUS Register 7 **Interrupt Status**

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	INITIALIZE Timeout Interrupt Disabled	Reserved For Future Use	Reserved For Future Use	RESET Key Interrupt Disabled	Keyboard and Knob Interrupt Disabled
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Control Register 7 (Set bit to disable) **Interrupt Disable Mask**

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not Used			INITIALIZE Timeout	Reserved For Future Use	Reserved For Future Use	RESET Key	Keyboard and Knob
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

STATUS Register 8 — Keyboard language jumper

0 = US ASCII
1 = French
2 = German
3 = Swedish/Finnish
4 = Spanish
5 = Katakana

CONTROL Register 8 — Undefined

STATUS Register 9 — Keyboard configuration jumper (0 thru 8)
CONTROL Register 9 — Undefined

STATUS Register 10 **Keyboard State at Last Knob Pulse**

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	0	CTRL Key Pressed	SHIFT Key Pressed
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

CONTROL Register 10 — Undefined

STATUS Register 11 — Reserved for future use
CONTROL Register 11 — Undefined

STATUS Register 12 — "Pseudo-EOI for **CTRL-E**" flag
CONTROL Register 12 — Enable pseudo-EOI for **CTRL-E** if non-0

STATUS Register 13 — Katakana flag
CONTROL Register 13 — Set Katakana if non-0 (**must** be a katakana keyboard)

Chapter 10
Unified I/O

Introduction

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 provide the ability of using either internal or ASCII data representation during I/O operations. The FORMAT attribute assigned to an I/O path determines which representation is used.

The second topic is that the OUTPUT and ENTER statements can be used to access most system resources, including the CRT display, the keyboard, and mass storage files (rather than using a separate set of BASIC statements for each type of resource). This second topic is herein called "unified I/O". This chapter also presents several useful applications of this powerful I/O scheme.

The Format Attributes

All I/O paths used as means to move data have certain attributes; the general attributes of a particular I/O path consist of both hardware and software characteristics. However, the attribute of interest in this discussion is that of data format, or how the computer represents the data it sends and how it interprets the data it receives through I/O paths.

All I/O paths possess either the FORMAT ON or the FORMAT OFF attribute. If an I/O path possesses the FORMAT ON attribute, the ASCII data representation is used during output and enter operations. If the I/O path possesses the FORMAT OFF attribute, the 9826's internal data representation is used. This section first describes how the FORMAT ON attribute is automatically assigned to I/O paths to devices and then shows how to assign the FORMAT OFF attribute to I/O paths. The actual internal representations are described in "The Format Off Attribute".

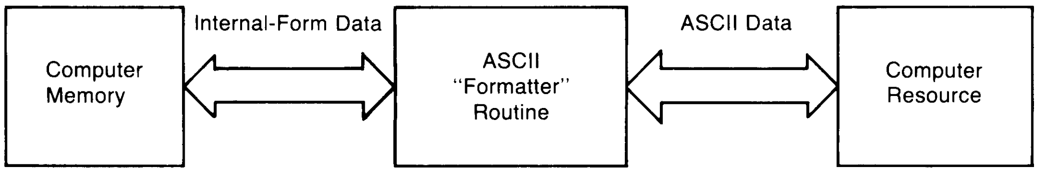
The Format On Attribute

Names are assigned to I/O paths between the computer and devices with the ASSIGN statement. A typical example is shown below.

```
110 ASSIGN @Any_name TO Device_selector
```

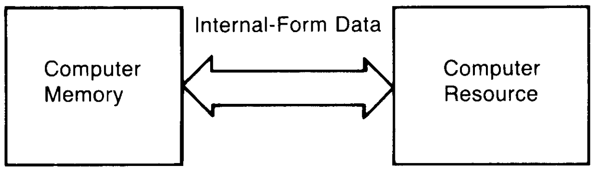
As you know from Chapter 3, this assignment fills a fixed amount of memory space with information describing the I/O path between the specified device and the computer. This information includes the device selector, the FORMAT attribute possessed by the path, and other relevant information. When the I/O path name is specified in subsequent ENTER and OUTPUT statements, this information adequately describes the I/O path to be used.

Since most devices use an ASCII data representation, the **default attribute** automatically assigned to the I/O path between the computer and **devices** is **FORMAT ON**. When an I/O path possesses this attribute, the **ASCII data representation** is automatically used by the computer when executing the OUTPUT and ENTER statements. Data output from the computer is "formatted" into an ASCII representation, and data entered into the computer is converted back into its internal representation[1]. The following diagram pictorially describes these operations.



The FORMAT ON Attribute Requires Data To Be Formatted

Data items moved through I/O paths which possess this attribute are formatted by operating-system firmware. This formatting process takes a finite amount of time for each data item to be moved, but is required for data compatibility when communicating with devices which use this data representation. Contrast the preceding diagram to the following diagram which shows data being moved through an I/O path possessing the FORMAT OFF attribute.



The Internal Data Representation Is Maintained with FORMAT OFF

Using the internal data representation during communication does not require the additional formatting time taken when the ASCII representation is used. However, the device must also use the internal data representation.

1 The ASCII data representation used when an I/O path possesses the FORMAT ON attribute was fully described in Chapters 4 and 5. The internal data representation used when an I/O path possesses the FORMAT OFF attribute is described later in this chapter.

One of the most powerful features of an I/O path is that its **FORMAT attribute can be changed from BASIC programs**. The next section describes specifying the FORMAT attribute and describes outputting and entering data through an I/O path which possesses the FORMAT OFF attribute.

Specifying I/O Path Attributes

There are two methods of explicitly specifying attributes. The first is to specify the desired attribute when the name is **initially assigned** to the resource, as shown below. Either the default FORMAT attribute or the alternate FORMAT attribute may be specified, as required for the application.

Example

```
100    ASSIGN @Device TO Dev_selector;FORMAT OFF
100    ASSIGN @Device TO Int_sel_code;FORMAT ON
```

Example

The second method allows you to change **only the attribute** currently assigned to the I/O path. As a result, the "TO resource" portion of the ASSIGN statement is not necessary; however, the I/O path name must currently be assigned in this context, or an error is reported.

`ASSIGN @Device;FORMAT OFF` Assign only the attribute.

The result of executing this 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 executed when the "TO resource" portion is included, is **not** executed. Also, the I/O path name must currently be assigned (in this context) to an I/O path, or an error results.

Example

If any attribute is specified, the corresponding entry in the I/O-path-name table is changed (as above). If no attribute is explicitly specified (as below), the attribute is changed to its default state (FORMAT ON for devices).

`340    ASSIGN @Device` Assigns the default attribute

The Format Off Attribute

Chapter 2 briefly described the internal data representations used for both computations and data storage. These internal representations are also used when moving data through I/O paths that possess the FORMAT OFF attribute. Since this chapter has already described how to assign the FORMAT OFF attribute to I/O paths, the only remaining information needed is a description of the actual FORMAT OFF (internal) data representations.

Notice that, in all cases, when an I/O path has been assigned the FORMAT OFF attribute:

- no item terminator and no EOL sequence are sent by the OUTPUT statement.
- no item terminator and no statement-termination condition are required by the ENTER statement.
- if either an OUTPUT or an ENTER statement uses an **image**, the FORMAT ON attribute is **automatically used**.

Compare this lack of terminators to those sent by the OUTPUT statement (and required by the ENTER statement) when using an I/O path possessing the FORMAT ON attribute (see Chapters 4 and 5). The next section describes the rationale behind the design of the following internal representations.

Integers
Integers are internally represented by two bytes (one word) of data. When an integer is output, only two bytes are sent with no trailing item terminator; no EOL sequence follows the last item in the source list. The most significant byte is sent first (on an eight-bit interface). When an integer is entered, only two bytes are entered from the source, and no search for an item terminator or statement-termination condition is made. If the source does not send two characters, a timeout may occur (if the event is set up on the interface involved)[1].

Real Numbers
Real numbers are internally represented by eight bytes. When a real number is output, only eight bytes are sent with no trailing item terminator; no EOL sequence follows the last item of the source list. When a real number is entered, only eight bytes are entered, and no search is made for item terminators. If eight bytes are not sent by the source, a timeout may occur (if set up on the interface).

String Data
String-data items are internally represented by a four-byte, binary length header followed by the actual string characters. When a string is output, this length header is output (most significant byte first) followed by the actual string characters. If the number of characters in the string is odd, a trailing space character (CHR$(32)) is sent to make an even number of characters. No trailing item terminator is output after the item, and no EOL sequence follows the last item in the source list.

When string data is entered into a string variable, the first four bytes entered determine the number of characters that the computer will attempt to enter. The source is expected to send the specified number of characters, so there is no need to search for item terminator or statement-termination condition.

1 Timeout events are discussed in Chapter 7, "Interface Events".

If the length specified by the header is greater than the dimensioned length of the string variable, an error is reported (`Error 18 String ovfl. or substring err`) and the string retains its former value. If the number of characters sent is less than that specified by the length header, an interface timeout may occur while the computer is waiting for the last character(s) to be sent by the source. If a timeout does occur (or if the **CLR IO** key is pressed before all characters have been received), the variable contains the characters that have been received.

Concepts of Unified I/O

The 9826's BASIC language and hardware provide the ability to communicate with the several system resources with the OUTPUT and ENTER statements. Chapters 8 and 9 described how to communicate with the operator 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. Chapter 11 describes how these I/O statements are used to communicate with HP-IB peripheral devices. And, if you have read about mass storage operations, 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 9826'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 these two representations in the 9826. The remainder of this chapter then presents several uses of this language structure.

Data-Representation Design Criteria

As you know, the 9826 supports two general data representations — the ASCII and the internal representations. This discussion describes these representations and 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 9826 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 two types of internal-disc **data** files, known as BDAT and ASCII files. **Only** the ASCII representation is used with ASCII files, but **either** representation can be used with BDAT files. The I/O paths to these files are described in this section to further justify the internal data representations implemented in the 9826 and to preface the applications presented in the last section of this chapter.

BDAT Files

BDAT (binary data) files have been designed with the first three of the preceding design criteria in mind. These internal representations allow much more data to be stored on a disc because there is **no storage overhead** (i.e., data items do not require a header that describes the type and length of the item). The **transfer time** required for each data item is also **decreased**. Numeric output operations are always much faster because the data are not formatted; all enter operations are also much faster because the computer does not have to search for item or statement terminators.

In addition, I/O paths to BDAT data files can use either the ASCII or the internal data representation; however, unless otherwise specified, the I/O path to a BDAT file is automatically assigned the **default attribute of FORMAT OFF**.

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. The positions of the file's pointers are shown during the program to illustrate how they are updated as data is output to the file.

```
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   !
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 both for compatibility with other HP disc drives and for program files. This compatibility requirement imposes the restriction that the data must be in its ASCII representation. Each data item sent to these files is a **special case** of the FORMAT ON representation; each item is preceded by a two-byte length header (analogous to the internal form of string data). Also, the FORMAT OFF attribute **cannot be assigned** to I/O paths to ASCII files, and OUTPUT or ENTER statements **cannot use images** when sending data to or receiving data from ASCII files.

The following program shows the I/O path name "@Io_path" being assigned to the ASCII file called "ASC_FILE". Notice that the file name is in all uppercase letters; this is also a compatibility requirement if the file is to be used with other disc drives. The program creates a program file, then gets and runs the program it has created. If you type in and run the program, be sure to save (or store) it before running it, as the program is scratched before running the "new" program.

```
100   DIM Line$(1:3)[100]
110   ON ERROR GOTO Already_exists
120   CREATE ASCII "ASC_FILE",1   ! 1 record.
130 Already_exists:  OFF ERROR
140   !
150   ASSIGN @Io_path TO "ASC_FILE"
160   STATUS @Io_path,6;Pointer
170   PRINT "Initially: file pointer= ";Pointer
180   PRINT
190   !
200   Line$(1)="100  PRINT ""New program.""  "
210   Line$(2)="110  BEEP"
220   Line$(3)="120  END"
230   !
240   OUTPUT @Io_path;Line$(*)
250   STATUS @Io_path,6;Pointer
260   PRINT "After OUTPUT: file pointer= ";Pointer
270   PRINT
280   !
290   GET "ASC_FILE" ! Implicitly closes I/O path.
300   !
310   END
```

Data Representation Summary

The following table summarizes the control that the program has over which FORMAT attribute is assigned to I/O paths.

Type of Resource	Default Format Attribute Used	Can Default Format Attribute Be Changed?
Devices	FORMAT ON	Yes, if an I/O path name is used[1]
BDAT Files	FORMAT OFF	Yes[1]
ASCII Files	FORMAT ON	No[2]
String Variables	FORMAT ON	No

Applications of Unified I/O

This section describes two uses of the powerful unified-I/O scheme of the 9826. 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.

1 FORMAT ON is **automatically used** as the attribute whenever an **image** is used by an OUTPUT or ENTER statement, **regardless** of the attribute currently assigned to the I/O path.

2 The data representation used with ASCII files is a special case of the FORMAT ON representation.

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
```

Final Display

```
---------------------
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
---------------------
```

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 9826'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 9826 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. The subprogram can then be expanded, tested, and stored in a separate disc file. The following display shows that only the "Equip_setup" subprogram is currently in memory.

```
660   SUB Equip_setup
670       PRINT "Equipment setup."
680       SUBEND
690       !
```

At this point, two steps can be taken. The temperature-measuring device's initialization routine can be written, or a test routine which simulates this device by returning a known set of data can be written. The most convenient approach at this point is to simulate the device. And with the 9826's BASIC language, the "Read_temp" subroutine will not have to be re-written later when the experiment is performed with the actual device.

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 to
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 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 is to be run with these subprograms. These parameters can also be passed to the subprograms by declaring them in variable common (i.e., by including them in the appropriate COM statements).

After the subprogram has been expanded, tested, and refined, it should be stored in a disc file with the STORE command (not the SAVE command). For instance, store the subprogram by executing `STORE "SETUP1"`. When the main program is to be tested again, the "Equip_setup" subprogram can be loaded back into memory by executing a `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_temp" 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 a statement such as `STORE "READ_T1"`. Now that both of the expanded subprograms have been stored, the main program can be retrieved and modified as necessary. Perform a `GET "MAIN1"` (or `LOAD "MAIN1"`), and 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 `DELSUB Equip_setup` and `DELSUB Read_temp` will delete only these subprograms and leave the rest of the program intact.

Now that the main program has been modified to CALL the expanded subroutines, you may want to save (or store) a copy of it on the disc. This will relieve you of deleting the old subprograms from the program every time it is retrieved. Execute a `SAVE "MAIN2"` (or `STORE "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 the actual temperature-measuring device. As you can see, this approach can be used very easily on the 9826. In addition, the "Read_temp" subprogram **need not be revised** to access the real device. Only "Equip_setup" needs to be revised to assign the I/O path name "@Temp_meter" to the real device. This unified-I/O scheme makes the 9826 very powerful and reduces "throw-away" code when using this top-down approach.

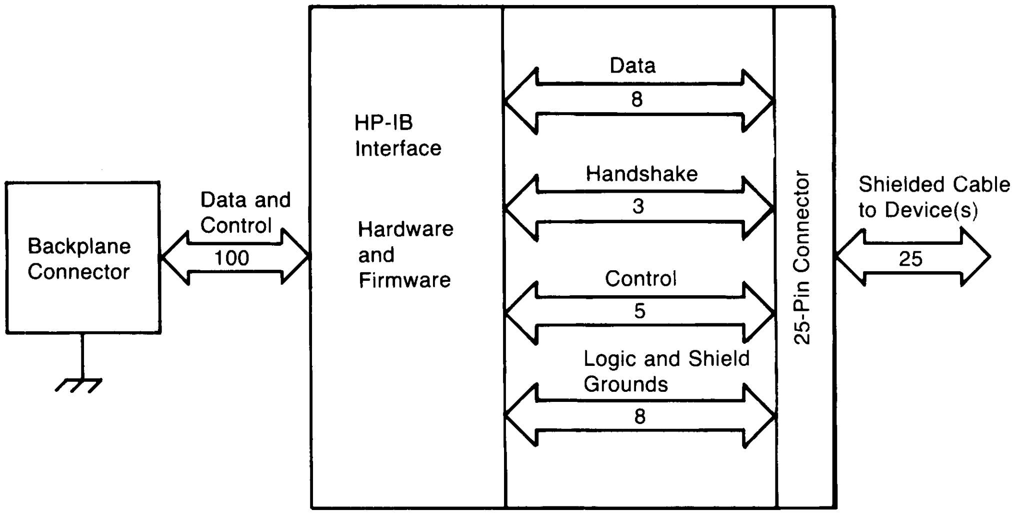
The remainder of the solution of this problem is to fill in the details of each remaining step of the process. Each major step of the program can be expanded tested, and refined separately. The use of hypothetical data is also a very good technique to isolate program errors before performing the experiment.

Chapter 11
The HP-IB Interface

Introduction

This chapter describes the techniques necessary for programming the HP-IB interface. Many of the elementary concepts have been discussed in previous chapters; this chapter describes the specific details of how this interface works and how it is used to communicate with and control systems consisting of various HP-IB devices.

The HP-IB (Hewlett-Packard Interface Bus), commonly called the "bus", provides compatibility between the computer and external devices conforming to the IEEE 488-1978 standard. Electrical, mechanical, and timing compatibility requirements are all satisfied by this interface.



The HP-IB Interface is both easy to use and allows great flexibility in communicating data and control information between the computer and external devices. It is one of the easiest methods to connect more than one device to the same interface.

Initial Installation

Refer to the HP-IB Installation Note for information about setting the switches and installing an external HP-IB interface. Once the interface has been properly installed, you can verify that the switch settings are what you intended by running the following program. The defaults of the internal HP-IB interface can also be checked with the program. The results are displayed on the CRT.

```
100    PRINTER IS 1
110    PRINT CHR$(12) ! Clear screen w/ FF.
120    !
130 Ask: INPUT "Enter HP-IB interface select code",Isc
140    IF Isc<7 OR Isc>30 THEN GOTO Ask
150    !
160    STATUS Isc;Card_id
170    IF Card_id<>1 THEN
180      PRINT "Interface at select code";Isc;
190      PRINT "is not an HP-IB"
200      PRINT
210      STOP
220    END IF
230    !
240    PRINT "HP-IB interface present"
250    PRINT " at select code";Isc
260    PRINT
270    !
280    STATUS Isc,1;Intr_dma
290    Level=3+(BINAND(32+16,Intr_dma) DIV 16)
300    PRINT "Hardware interrupt level =";Level
310    !
320    STATUS Isc,3;Addr_ctrlr
330    Address=Addr_ctrlr MOD 32
340    PRINT "Primary address =";Address
350    !
360    Sys_ctrl=BIT(Addr_ctrlr,7)
370    IF Sys_ctrl THEN
380      PRINT "System Controller"
390    ELSE
400      PRINT "Non-system Controller"
410    END IF
420    !
430    END
```

The hardware interrupt level is described in Chapter 7. Hardware interrupt level is set to 3 on the internal HP-IB interface, but can range from 3 to 6 on external interfaces. Primary address is further described in "HP-IB Device Selectors" in the next section.

The term "system controller" is also further described later in this chapter in "General Structure of the HP-IB". The internal HP-IB has a jumper that is set at the factory to make it a system controller. This jumper is located below the lowest interface slot at the computer backplane. The lowest interface (or memory board) in the backplane must be removed to access this jumper. If the **jumper in the center of the clear plastic cover is placed on the middle and rightmost pins**, (as seen from the rear of the computer), the computer is set to be a system controller. If it is **on the middle and leftmost pins**, the computer is **not** a system controller. External HP-IB interfaces have a switch that controls this interface state.

Communicating with Devices

This section describes programming techniques used to output data to and enter data from HP-IB devices. General bus operation is also briefly described in this chapter. Later chapters will describe: further details of specific bus commands, handling interrupts, and advanced programming techniques.

HP-IB Device Selectors

Since the HP-IB allows the interconnection of several devices, each device must have a means of being uniquely accessed. Specifying just the interface select code of the HP-IB interface through which a device is connected to the computer is not sufficient to uniquely identify a specific device on the bus.

Each device "on the bus" has an **address** by which it can be identified; this address must be unique to allow individual access of each device. Each HP-IB device has a set of switches that are used to set its address. Thus, when a particular HP-IB device is to be accessed, it must be identified with both its interface select code and its bus address.

The interface select code is the first part of an HP-IB device selector. The interface select code of the internal HP-IB is 7; external interfaces can range from 8 to 31. The second part of an HP-IB device selector is the device's primary address, which are in the range of 0 through 30. For example, to specify the device:

on interface select code 7 with primary address 22	use device selector = 722
on interface select code 10 with primary address 2	use device selector = 1002

Remember that each device's address must be unique. The procedure for setting the address of an HP-IB device is given in the installation manual for each device. The HP-IB interface also has an address. The default address of the internal HP-IB is 21 or 20, depending on whether or not it is a system controller. The addresses of external HP-IB interfaces are set by configuring the address switches on each interface card. Each HP-IB interface's address can be determined by reading STATUS register 3 of the appropriate interface select code, and each interface's address can be changed by writing to CONTROL register 3. See "Determining Controller Status and Address" and "Changing the Controller Address" for further details.

Moving Data Through the HP-IB

Data is output from and entered into the computer through the HP-IB with the OUTPUT and ENTER statements, respectively; all of the techniques described in Chapters 4 and 5 are completely applicable with the HP-IB. The only difference between the OUTPUT and ENTER statements for the HP-IB and those for other interfaces is the addressing information within HP-IB device selectors.

Examples

```
100   Hpib=7
110   Device_addr=22
120   Device_selector=Hpib*100+Device_addr
130   !
140   OUTPUT Device_selector;"F1R7T2T3"
150   ENTER Device_selector;Reading

320   ASSIGN @Hpib_device TO 702
330   OUTPUT @Hpib_device;"Data message"
340   ENTER @Hpib_device;Number

440   OUTPUT 822;"F1R7T2T3"

380   ENTER 724;Readings(*)
```

All of the IMAGE specifiers described in Chapters 4 and 5 can also be used by OUTPUT and ENTER statements that access the HP-IB interface, and the definitions of all specifiers remain exactly as stated in those chapters.

Examples

```
100   ASSIGN @Printer TO 701
110   OUTPUT @Printer USING "6A,3X,2D.D";Item$,Quantity

860   ASSIGN @Device TO 825
870   OUTPUT @Device USING "#,B";65,66,67,13,10
870   ENTER @Device USING "#,K";Data$
```

General Structure of the HP-IB

Communications through the HP-IB are made according to a precisely defined set of rules. These rules help to ensure that only orderly communication may take place on the bus. For conceptual purposes, the organization of the HP-IB can be compared to that of a committee. A committee has certain "rules of order" that govern the manner in which business is to be conducted. For the HP-IB, these rules of order are the IEEE 488-1978 standard.

One member, designated the "committee chairman," is set apart for the purpose of conducting communications between members during the meetings. This chairman is responsible for overseeing the actions of the committee and generally enforces the rules of order to ensure the proper conduct of business. If the committee chairman cannot attend a meeting, he designates some other member to be "acting chairman."

On the HP-IB, the **system controller** corresponds to the committee chairman. The system controller is generally designated by setting a switch on the interface and cannot be changed under program control. However, it is possible to designate an "acting chairman" on the HP-IB. On the HP-IB, this device is called the **active controller**, and may be any device capable of directing HP-IB activities, such as a desktop computer.

When the system controller is first turned on or reset, it assumes the role of active controller. Thus, only one device can be designated system controller. These responsibilities may be subsequently passed to another device while the system controller tends to other business. This ability to pass control allows more than one computer to be connected to the HP-IB at the same time.

In a committee, only one person at a time may speak. It is the chairman's responsibility to "recognize" which one member is to speak. Usually, all committee members present always listen; however, this is not always the case on the HP-IB. One of the most powerful features of the bus is the ability to selectively send data to individual (or groups of) devices.

Imagine slow note takers and a fast note takers on the committee. Suppose that the speaker is allowed to talk no faster than the slowest note taker can write. This would guarantee that everybody gets the full set of notes and that no one misses any information. However, requiring all presentations to go at that slow pace certainly imposes a restriction on our committee, especially if the slow note takers do not need the information. Now, if the chairman knows which presentations are not important to the slow note takers, he can direct them to put away their notes for those presentations. That way, the speaker and the fast note taker(s) can cover more items in less time.

A similar situation may exist on the HP-IB. Suppose that a printer and a flexible disc are connected to the bus. Both devices do not need to listen to all data messages sent through the bus. Also, if all the data transfers must be slow enough for the printer to keep up, saving a program on the disc would take as long as listing the program on the printer. That would certainly not be a very effective use of the speed of the disc drive if it was the only device to receive the data. Instead, by "unlistening" the printer whenever it does not need to receive a data message, the computer can save a program as fast as the disc can accept it.

During a committee meeting, the current chairman is responsible for telling the committee which member is to be the talker and which is (are) to be the listener(s). Before these assignments are given, he must get the **attention** of all members. The talker and listener(s) are then designated, and the next data message is presented to the listener(s) by the talker. When the talker has finished the message, the designation process may be repeated.

On the HP-IB, the active controller takes similar action. When talker and listener(s) are to be designated, the **attention signal line** (ATN) is asserted while the talker and listener(s) are being addressed. ATN is then cleared, signaling that those devices not addressed to listen may ignore all subsequent data messages. Thus, **the ATN line separates data from commands**; commands are accompanied by the ATN line being true, while data messages are sent with the ATN line false.

On the HP-IB, devices are **addressed to talk** and **addressed to listen** in the following orderly manner. The active controller first sends a single command which causes all devices to **unlisten**. The talker's address is then sent, followed by the address(es) of the listener(s). After all listeners have been addressed, the data can be sent from the talker to the listener(s). Only device(s) addressed to listen accept any data that is sent through the bus (until the bus is reconfigured by subsequent addressing commands).

The data transfer, or **data message**, allows for the exchange of information between devices on the HP-IB. Our committee conducts business by exchanging ideas and information between the speaker and those listening to his presentation. On the HP-IB, **data is transferred from the active talker to the active listener(s) at a rate determined by the slowest active listener on the bus**. This restriction on the transfer rate is necessary to ensure that no data is lost by any device addressed to listen. The **handshake** used to transfer each data byte ensures that all data output by the talker is received by all active listeners.

Examples of Bus Sequences

Most data transfers through the HP-IB involve a talker and only one listener. For instance, when an OUTPUT statement is used to send data to an HP-IB device, the following sequence of commands and data is sent through the bus.

```
OUTPUT 701;"Data"
```

1. The unlisten command is sent.
2. The talker's address is sent (here, the address of the computer; "My Talk Address"), which is also a command.
3. The listener's address (01) is sent, which is also a command.
4. The data bytes "D", "a", "t", "a", CR, and LF are sent; all bytes are sent using the HP-IB's interlocking handshake to ensure that the listener has received each byte.

Similarly, most ENTER statements involve transferring data from a talker to only one listener. For instance, the following ENTER statement invokes the following sequence of commands and data-transfer operations.

```
ENTER 722;Voltage
```

1. The unlisten command is sent.
2. The talker's address (22) is sent, which is a command.
3. The listener's address is sent (here, the computer's address; "My Listen Address"), also a command.
4. The data is sent by device 22 to the computer using the HP-IB handshake.

Bus sequences, hardware signal lines, and more specific HP-IB operations are discussed in the "HP-IB Control Lines" and "Advanced Bus Management" sections.

Addressing Multiple Listeners

HP-IB allows more than one device to listen simultaneously to data sent through the bus (even though the data may be accepted at differing rates). The following examples show how to address multiple listeners on the bus.

```
100   ASSIGN @Listeners TO 701,702,703
110   OUTPUT @Listeners;String$
120   OUTPUT @Listeners USING Image_1;Array$(*)
```

This capability allows a single OUTPUT statement to send data to several devices simultaneously. It is however, necessary for all the devices to be on the same interface. When the preceding OUTPUT statement is executed, the unlisten command is sent first, followed by the computer's talk address and then listen addresses 01, 02, and 03. Data is then sent by the computer and accepted by devices at addresses 1, 2, and 3.

If an ENTER statement is performed using the same I/O path name, the first device is addressed as the talker (the source of data) and all the rest of the devices, including the 9826, are addressed as listeners. The data is then sent from device at address 01 to devices at addresses 02 and 03 and to the computer.

```
130   ENTER @Listeners;String$
140   ENTER @Listeners USING Image_2;Array$(*)
```

Addressing a Non-Controller 9826

The bus standard states that **a non-active controller cannot perform any bus addressing**. When **only the interface select code** is specified in an ENTER or OUTPUT statement that uses an HP-IB interface, **no bus addressing is performed**.

If the 9826 currently is **not the active controller**, it can still act as either talker or listener, provided it has been **previously addressed** as such. Thus, if an ENTER or OUTPUT statement is executed while the 9826 is not an active controller, the computer first determines whether or not it is an active talker or listener. If not addressed to talk or listen, the computer waits until it is properly addressed and then finishes executing the statement. It relies on the active controller (another computer or device) to perform the bus addressing, and then simply participates as a device in the exchange of the data. Example statements which send and receive data while the computer is not an active controller are as follows.

```
100  OUTPUT 7;"Data" ! If not talker, then wait until
110                  ! addressed as talker to send data.

200  ENTER 7;Data$ ! If not listener, then wait until
210                ! addressed as listener to accept data.
```

If the 9826 is the **active controller**, it proceeds with the data transfer without addressing which devices are talker and listener(s). However, if the bus has not been configured previously, an error is reported (`Error 170 I/O operation not allowed`). The following program does not require the "overhead" of addressing talker and listeners each time the OUTPUT statement in the FOR-NEXT loop is executed, because the bus is not reconfigured each time.

```
100   OUTPUT 701 USING "#,K"  ! Configure the bus:
110                           ! 9826 = talker, and
120                           ! printer (701) = listener.
130                           !
140   FOR Iteration=1 TO 25
150       OUTPUT 7;"Data message"
160   NEXT Iteration
170   !
180   END
```

This type of HP-IB addressing should be used with the understanding that if an event initiates a branch between the time that the initial addressing was made (line 100) and the time that any of the OUTPUT statements are executed (line 150), the event's service routine may reconfigure the bus differently than the initial configuration. If so, the data will be directed to the device(s) that have been addressed to listen by the last I/O statement executed in the service routine. Events may need to be disabled if this method of addressing is used.

In general, most applications do not require this type of bus-overhead minimization; the 9826's I/O language has already been optimized to provide excellent performance. Advanced methods of explicit bus management will be described in the section called "Advanced Bus Management".

Secondary Addressing

Many devices have operating modes which are accessed through the extended addressing capabilities defined in the bus standard. Extended addressing provides for a second address parameter in addition to the primary address. Examples of statements that use extended addressing are as follows.

```
100    ASSIGN @Device TO 72205   ! 22=primary, 05=secondary.
110    OUTPUT @Device;Message$

200    OUTPUT 72205;Message$

150    ASSIGN @Device TO 7220529 ! Additional secondary
160                              ! address of 29.
170    OUTPUT @Device;Message$

120    OUTPUT 7220529;Message$
```

The range of secondary addresses is 00-31; up to six secondary addresses may be specified (a total of 15 digits including interface select code and primary address). Refer to the device's operating manual for programming information associated with the extended addressing capability. The HP-IB interface also has a mechanism for detecting secondary commands. For further details, see the discussion of interrupts.

Determining Controller Status and Address

It is often necessary to determine if an interface is the system controller and to determine whether or not it is the current active controller. It is also often necessary to determine or change the interface's primary address. The example program shown in the beginning of this chapter interrogated interface STATUS registers and printed the resultant system-controller status and primary address. Those operations are explained in the following paragraphs.

Example

Executing the following statement reads STATUS register 3 (of the internal HP-IB) and places the current value into the variable Stat_and_addr. Remember that if the statement is executed from the keyboard, the variable Stat_and_addr must be defined in the current context.

```
STATUS 7,3;Stat_and_addr
```

Status Register 3 **Controller Status and Address**

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
System Controller	Active Controller	0	Primary Address of Interface				
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

If **bit 7** is set (1), it signifies that the interface is the system controller; if clear (0), it is not the system controller. Only one controller on each HP-IB interface should be configured as the system controller.

If **bit 6** is set (1), it signifies that the interface is currently the active controller; if it is clear (0), another controller is currently the active controller.

Bits 4 through 0 represent the current value of the interface's primary address, which is in the range of 0 through 30. The power-on default value for the internal HP-IB is 21 (if it is the system controller) and 20 (if not the system controller). For external HP-IB interfaces, the default address is set to 21 at the factory but may be changed by setting the address switches on the card itself.

Example

Calculate the primary address of the interface from the value previously read from STATUS register 3.

```
Intf_addr=Stat_and_addr MOD 32
```

This numerical value corresponds to the talk (or listen) address sent by the computer when an OUTPUT (or ENTER) statement containing primary-address information is executed. Talk and listen addresses are further described in "Advanced Bus Management".

Changing the Controller Address

It is possible to use the CONTROL statement to change an HP-IB interface's address.

Example

```
CONTROL 7,3;Intf_addr
```

The value of Intf_addr is used to set the address of the HP-IB interface (in this case, the internal HP-IB). The valid range of addresses is 0 through 30; **address 31 is not used**. Thus, if a value greater than 30 is specified, the value MOD 32 is used (for example: 32 MOD 32 equals 0, 33 MOD 32 equals 1, 62 MOD 32 equals 30, and so forth).

General Bus Management

The HP-IB standard provides several mechanisms that allow managing the bus and the devices on the bus. Here is a summary of the statements that invoke these control mechanisms.

ABORT is used to abruptly terminate all bus activity and reset all devices to power-on states.

CLEAR is used to set all (or only selected) devices to a pre-defined, device-dependent state.

LOCAL is used to return all (or selected) devices to local (front-panel) control.

LOCAL LOCKOUT is used to disable all devices' front-panel controls.

PPOLL is used to perform a parallel poll on all devices (which are configured and capable of responding).

PPOLL CONFIGURE is used to setup the parallel poll response of a particular device.

PPOLL UNCONFIGURE is used to disable the parallel poll response of a device (or all devices on an interface).

REMOTE is used to put all (or selected) devices into their device-dependent, remote modes.

SEND is used to manage the bus by sending explicit command or data messages.

SPOLL is used to perform a serial poll of the specified device (which must be capable of responding).

TRIGGER is used to send the trigger message to a device (or selected group of devices).

These statements (and functions) are described in the following discussion. However, the actions that a device takes upon receiving each of the above commands are, in general, different for each device. Refer to a particular device's manuals to determine how it will respond. Detailed descriptions of the actual sequence of bus messages invoked by these statements are contained in "Advanced Bus Management" near the end of this chapter.

Remote Control of Devices

Most HP-IB devices can be controlled either from the front panel or from the bus. If the device's front-panel controls are currently functional, it is in the Local state. If it is being controlled through the HP-IB, it is in the Remote state. Pressing the front-panel "Local" key will return the device to Local (front-panel) control, unless the device is in the Local Lockout state (described in a subsequent discussion).

The Remote message is automatically sent to all devices whenever the system controller is powered on, reset, or sends the Abort message. A device also enters the Remote state automatically whenever it is addressed. The REMOTE statement also outputs the Remote message, which causes all (or specified) devices on the bus to change from local control to remote control. The 9826 must be the system controller to execute the REMOTE statement.

Examples

```
REMOTE 7

ASSIGN @Device TO 700
REMOTE @Device

REMOTE 700
```

Locking Out Local Control

The Local Lockout message effectively locks out the "local" switch present on most HP-IB device front panels, preventing a device's user from interfering with system operations by pressing buttons and thereby maintaining system integrity. As long as Local Lockout is in effect, no bus device can be returned to local control from its front panel.

The Local Lockout message is sent by executing the LOCAL LOCKOUT statement. This message is sent to all device on the specified HP-IB interface, and it can only be sent by the 9826 when it is the active controller.

Examples

```
ASSIGN @Hpib TO 7
LOCAL LOCKOUT @Hpib

LOCAL LOCKOUT 7
```

The Local Lockout message is cleared when the Local message is sent by executing the LOCAL statement. However, executing the ABORT statement does not cancel the Local Lockout message.

Enabling Local Control

During system operation, it may be necessary for an operator to interact with one or more devices. For instance, an operator might need to work from the front panel to make special tests or to troubleshoot. And, in general, it is good systems practice to return all devices to local control upon conclusion of remote-control operations. Executing the LOCAL statement returns the specified devices to local (front-panel) control. The 9826 must be the active controller to send the LOCAL message.

Examples

```
ASSIGN @Hpib TO 7
LOCAL @Hpib

ASSIGN @Device TO 700
LOCAL @Device
```

If primary addressing is specified, the Go-to-Local message is sent only to the specified device(s). However, if only the interface select code is specified, the Local message is sent to all devices on the specified HP-IB interface and any previous Local Lockout message (which is still in effect) is automatically cleared. The 9826 must be the system controller to send the Local message (by specifying only the interface select code).

Triggering HP-IB Devices

The TRIGGER statement sends a Trigger message from the controller to a selected device or group of devices. The purpose of the Trigger message is to initiate some device-dependent action; for example, it can be used to trigger a digital voltmeter to perform its measurement cycle. Because the response of a device to a Trigger Message is strictly device-dependent, neither the Trigger message nor the interface indicates what action is initiated by the device.

Examples

```
ASSIGN @Hpib TO 7
TRIGGER @Hpib

ASSIGN @Device TO 707
TRIGGER @Device
```

Specifying only the interface select code outputs a Trigger message to all devices currently addressed to listen on the bus. Including device addresses in the statement triggers only those devices addressed by the statement. The 9826 can also respond to a trigger from another controller on the bus. See "HP-IB Interrupts" for details.

Clearing HP-IB Devices

The CLEAR statement provides a means of "initializing" a device to its predefined, device-dependent state. When the CLEAR statement is executed, the Clear message is sent either to all devices or to the specified device(s), depending on the information contained within the device selector. If only the interface select code is specified, all devices on the specified HP-IB interface are cleared. If primary-address information is specified, the Clear message is sent only to the specified device. Only the active controller can send the Clear message.

Examples

```
ASSIGN @Hpib TO 7
CLEAR @Hpib

ASSIGN @Device TO 700
CLEAR @Device
```

Aborting Bus Activity

This statement may be used to terminate all activity on the bus and return all the HP-IB interfaces of all devices to a reset (or power-on) condition. Whether this affects other modes of the device depends on the device itself. The 9826 must be either the active or the system controller to perform this function. If the system controller (which is not the current active controller) executes this statement, it regains active control of the bus. **Only the interface select code may be specified**; device selectors which contain primary-addressing information (such as 724) may not be used.

Examples

```
ASSIGN @Hpib TO 7
ABORT @Hpib

ABORT 7
```

Polling HP-IB Devices

The parallel poll is the fastest means of gathering device status when several devices are connected to the bus. Each device (with this capability) can be programmed to respond with one bit of status when parallel polled, making it possible to obtain the status of several devices in one operation. If a device responds affirmatively to a parallel poll, more information as to its specific status can be obtained by conducting a serial poll of the device.

Configuring Parallel Poll Responses

Certain devices can be remotely programmed by the active controller to respond to a parallel poll. A device which is currently configured for a parallel poll responds to the poll by placing its current status on one of the bus data lines. The logic sense of the response and the data-bit number can be programmed by the PARALLEL POLL CONFIGURE statement. No multiple listeners can be specified in the statement; if more than one device is to respond on a single bit, each device must be configured with a separate PARALLEL POLL CONFIGURE statement.

Example

```
ASSIGN @Device TO 701
PPOLL CONFIGURE @Device;Mask
```

The value of **Mask** (any numeric expression can be specified) is first rounded and then used to configure the device's parallel response. The least significant 3 bits (bits 0 through 2) of the expression are used to determine which data line the device is to respond on (place its status on). Bit 3 specifies the "true" state of the parallel poll response bit of the device. A value of 0 implies that the device's response is 0 when its status-bit message is true.

Example

The following statement configures device at address 01 on interface select code 7 to respond by placing a 0 on bit 4 when its status response is "true".

```
PPOLL CONFIGURE 701;4
```

Conducting a Parallel Poll

The PPOLL function returns a single byte containing up to 8 status bit messages of all devices on the bus capable of responding to the poll. Each bit returned by the function corresponds to the status bit of the device(s) configured to respond to the parallel poll. (Recall that one or more devices can respond on a single line.) The PPOLL function can only be executed by the 9826 when it is the active controller.

Example

```
Response=PPOLL(7)
```

Disabling Parallel Poll Responses

The PARALLEL POLL UNCONFIGURE statement gives the 9826 (as active controller) the capability of disabling the parallel poll responses of one or more devices on the bus.

Examples

The following statement disables device 5 only.

```
PPOLL UNCONFIGURE 705
```

This statement disables all devices on interface select code 8 from responding to a parallel poll.

```
PPOLL UNCONFIGURE 8
```

If no primary addressing is specified, all bus devices are disabled from responding to a parallel poll. If primary addressing is specified, only the specified devices (which have the parallel poll configure capability) are disabled.

Conducting a Serial Poll

A sequential poll of individual devices on the bus is known as a serial poll. One entire byte of status is returned by the specified device in response to a serial poll. This byte is called the Status Byte message and, depending on the device, may indicate an overload, a request for service, or a printer being out of paper. The particular response of each device depends on the device.

The SPOLL function performs a serial poll of the specified device; the 9826 must be the active controller.

Examples

```
ASSIGN @Device TO 700
Status_byte=SPOLL(@Device)

Spoll_24=SPOLL(724)
```

Just as the parallel poll is not defined for individual devices, the serial poll is meaningless for an interface; therefore, primary addressing must be used with the SPOLL function.

HP-IB Interrupts

Interrupts allow the most efficient use of the power of the computer by allowing the computer to proceed with meaningful tasks while waiting for external devices to complete some action requested or intiated by the computer. Chapter 7 described several uses of interrupts and presented a few general techniques for their use. This chapter describes HP-IB interrupts in particular.

There are two general types of interrupts in an HP-IB system. Interrupts may be generated either by external devices or by the interface when it detects a specific change in a bus operating mode. First, service requests will be discussed, which are the most common interrupt condition for most HP-IB systems. Then, the second type of interrupts are described in "Interrupts while Non-Active Controller".

The SRQ Interrupt

The mnemonic SRQ stands for service request. Many HP-IB devices, such as voltmeters, counters, and spectrum analyzers, are capable of generating a service request when they have completed some action, such as taking a reading or finishing a scan. Devices such as printers and plotters are often able to generate a service request when out of paper, and many other devices are able to signal error conditions using this mechanism.

The HP-IB standard has made provision for a signal line within the interface cable over which devices may signal to the computer their service request. This line has the mnemonic SRQ. Though this method of requesting service is defined by the standard, it is completely up to the device as to the meaning of any request. The operating and programming manuals for the device provide this information.

Example

The following program segment sets up and enables an SRQ interrupt.

```
100   ASSIGN @Hpib TO 7
110   ON INTR @Hpib GOSUB Service_routine
120   !
130   Mask=2  ! Bit 1 enables SRQ interrupts.
140   ENABLE INTR @Hpib;Mask
```

The value of the mask in the ENABLE INTR statement determines which type(s) of interrupts are to be enabled. The value of the mask is automatically written into the HP-IB interface's interrupt-enable register (CONTROL register 4) when this statement is executed. Bit 1 is set in the preceding example, enabling SRQ interrupts to initiate a program branch. Reading STATUS register 4 at this point would return a value of 2.

When an SRQ interrupt is generated by any device on the bus, the program branches to the service routine when the current line is exited (either when the line's execution is finished or when the line is exited by a call to a user-defined function subprogram). The service routine must (in general):

- determine which device is requesting service (parallel poll)
- determine what action is requested (serial poll)
- clear the SRQ line (automatic with serial poll)
- perform the desired action
- re-enable interrupts
- return to the former task (if applicable)

Servicing External Requests
The SRQ is a level-sensitive interrupt; in other words, if an SRQ is present momentarily but does not remain long enough to be sensed by the computer, the interrupt will not be generated. The level-sensitive nature of the SRQ line also has further implications, which are described in the following paragraphs.

Example

Assume that only one device is currently on the bus. The following service routine first serially polls the device requesting service, thereby clearing the interrupt request. In this case, the computer did not have to determine which device was requesting service because it is the only device on the bus. It is also assumed that only service request interrupts have been enabled; therefore, the type of interrupt need not be determined either. The service is then performed, and the SRQ event is re-enabled.

```
500  Serv_rtn:  Ser_poll=SPOLL(@Device)
510               ENTER @Device;Value
520               PRINT Value
530               ENABLE INTR 7   ! Use previous mask.
540               RETURN
```

The standard has defined that when an interrupting device is serially polled, it is to stop interrupting until a new condition arises (or the same condition arises again). In order to "clear" the SRQ line, it is necessary to perform a serial poll on the device. The poll is an acknowledgement from the controller to the device that it has seen the request for service and is responding.

Had the SRQ line not been cleared, the computer would have branched to the service routine immediately upon re-enabling interrupts on this interface. This is another implication of the level-sensitive nature of the SRQ interrupt.

It is also important to note that once an interrupt is sensed and logged, the interface cannot generate another interrupt until the initial interrupt is serviced. The computer disables all subsequent interrupts from an interface until a pending interrupt is serviced. For this reason, it was necessary to re-enable the interrupt to allow for subsequent branching.

Requesting Service from Another Computer
Imagine a system where the computer is one of two or more controllers on the same bus. Remember that only one of the controllers may be the active controller on the bus at any time. However, suppose that one of the non-controller computers needs service from the active controller.

Example

A service request is generated when the computer executes the following statement.

```
CONTROL 7,1;64   ! Setting bit 6 generates an SRQ.
```

This statement instructs the HP-IB interface to generate a service request by setting the SRQ signal line true. The computer's response to the serial poll is executed within a service routine that is initiated by the serial poll from the active controller. Interrupts of this type (while the computer is a non-active controller) are described in the following section.

Interrupts While Non-Active Controller

When the computer is not an active controller, it must be able to detect and respond to many types of bus messages and events. One way to do this is to continually monitor the HP-IB interface by executing the STATUS statement and then taking action when the values returned match the values desired. This is obviously a great waste of computer time if the computer could be performing other tasks. Instead, the interface hardware can be enabled to monitor bus activity and then generate interrupts when certain events take place.

The computer (as a non-active controller) needs to keep track of the following information.

- It must keep track of itself being addressed as a listener so that it can enter data from the current active talker.
- It must keep track of itself being addressed as a talker so that it can transmit the information desired by the active controller.
- It must keep track of being sent a Clear, Trigger, Local or Local Lockout message so that it can take appropriate action.
- It must keep track of control being passed from another controller.

The 9826 has the ability to keep track of the occurrences of all of these events. In fact, it can monitor up to 16 different interrupt conditions. STATUS registers 4, 5 and 6 provide access to the interface state and interrupt information necessary to design very powerful systems with a great degree of flexibility.

Each individual bit of STATUS register 4 corresponds to the same bit of STATUS register 5. Register 4 provides information as to which condition **caused** an interrupt, while register 5 keeps track of which interrupt conditions are **currently enabled**. To enable a combination of conditions, add the decimal values for each bit that you want set in the interrupt-enable register. This total is then used as the mask parameter in an ENABLE INTR statement.

HP-IB Interrupt Registers

Status Register 5 **Interrupt Enable Mask**

Most Significant Bit

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
Active Controller	Parallel Poll Configuration Change	My Talk Address Received	My Listen Address Received	EOI Received	SPAS	Remote/ Local Change	Talker/ Listener Address Change
Value = −32 768	Value = 16 384	Value = 8 192	Value = 4 096	Value = 2 048	Value = 1 024	Value = 512	Value = 256

Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Trigger Received	Handshake Error	Unrecognized Universal Command	Secondary Command While Addressed	Clear Received	Unrecognized Addressed Command	SRQ Received	IFC Received
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Bit 0 enables an interrupt upon detecting an Interface Clear (IFC). The interrupt is generated only when the computer is not the system controller, as only a system controller is allowed to set the Interface Clear signal line. The service routine typically is used to recover from the abrupt termination of an I/O operation caused by another controller sending the IFC message.

Bit 1 enables an interrupt upon detecting a Service Request.

Bit 2[1] enables an interrupt upon receiving an unrecognized Addressed Command, if the computer is currently addressed to listen. This interrupt is used to intercept and respond to bus commands which are not defined by the standard.

Bit 3 enables an interrupt on receiving a Clear message. Reception of either a Device Clear message (to all devices) or a Selected Device Clear message (addressed to the computer) will cause this type of interrupt. The computer is free to take any "device-dependent" action; such as, setting up all default values again, or even restarting the program, if that is defined by the programmer to be the "cleared" state of the machine.

Bit 4[1] enables an interrupt upon receiving a Secondary Command (extended addressing) while in the command mode, if the 9826 is addressed to either talk or listen. Again, this interrupt provides the computer with a way to detect and respond to special messages from another controller.

1 This condition requires accepting data from the bus and then explicitly releasing the bus. Refer to the "Advanced Bus Management" section for further details.

Bit 5[1] enables an interrupt upon receiving an unrecognized Universal Command. This interrupt condition provides the computer with the capability of responding to new definitions that may be adopted by the IEEE standards committee.

Bit 6 enables an interrupt if a bus error occurs during an OUTPUT statement. Particularly, the error occurs if none of the devices on the bus respond to the HP-IB's interlocking handshake (see "HP-IB Control Lines"). The error typically indicates that either a device is not connected or that its power is off.

Bit 7 enables an interrupt upon receiving a Trigger message, if the computer is currently addressed to listen. This interrupt can be used in situations where the computer may be "armed and waiting" to initiate action; the active controller sends the Trigger message to the computer to cause it to begin its task.

Bit 8 enables an interrupt upon a change in talk or listen address. An interrupt will be generated if the computer is addressed to listen or talk or "idled" by an unlisten command.

Bit 9 enables an interrupt upon receiving either the Remote or the Local message from the active controller, if addressed to listen. The action taken by the computer is, of course, dependent on the user-programmed service routine.

Bit 10 enables an interrupt when the active controller performs a serial poll on the computer (in response to its service request).

Bit 11 enables an interrupt when an EOI is received during an ENTER operation (the EOI signal line is also described in "HP-IB Control Lines").

Bit 12 enables an interrupt upon being addressed as an active listener by the active controller.

Bit 13 enables an interrupt upon being addressed as an active talker by the active controller.

Bit 14[1] enables an interrupt upon detecting a change in parallel poll configuration.

Bit 15 enables an interrupt upon becoming the active controller. The computer then has the ability to manage bus activities.

Note that most of the conditions are state- or event-sensitive; the exception is the SRQ event, which is level-sensitive. State-or event-sensitive events can never go unnoticed by the computer as can service requests; the event's occurrence is "remembered" by the computer until serviced.

For instance, if the computer is enabled to generate an interrupt on becoming addressed as a talker, it would interrupt the first time it received its own talk address. After having responded to the service request (most likely with some sort of OUTPUT operation), it would not generate another interrupt, even if it was still left assigned as a talker by the active controller. Thus, it would not generate another interrupt until the event occurred a second time.

1 This condition requires accepting data from the bus and then explicitly releasing the bus. Refer to the "Advanced Bus Management" section for further details.

A simple example of a service routine that is to respond to multiple conditions might be as follows.

```
100   ON INTR @Hpib GOSUB Service
110   Mask=INT(2^13)+INT(2^12)
120   ENABLE INTR @Hpib;Mask ! Interrupt on receiving
130                          ! talk or listen addr.
140 Idle: GOTO Idle
150       !
160 Service: STATUS @Hpib,4;Status,Mask
170          IF BIT(13,Status) THEN Talker
180          IF BIT(12,Status) THEN Listener
190          !
200 Talker: ! Take action for talker.
210         GOTO Exit_point
220         !
230 Listener: ! Take action for listener.
240           !
250 Exit_point: ENABLE INTR @Hpib;Mask
260             RETURN
270   END
```

Register 4, the interrupt status register, is a "read-destructive" register; reading the register with a STATUS statement returns its contents and then **clears the register** (to a value of 0). If the service routine's action depends on the contents of STATUS register 4, the variable in which it is stored must not be used for any other purposes before all of the information that it contains has been used by the service routine.

The computer is automatically addressed to talk (by the active controller) whenever it is serially polled. If interrupts are concurrently enabled for My Address Change and/or Talker Active, the ON INTR branch will be initiated due to the reception of the computer's talk address. However, since the Serial Poll is automatically finished with the Untalk Command, the computer may no longer be addressed to talk by the time the interrupt service routine begins execution.

Interface-State Information

It is often necessary to determine which state the interface is in. STATUS register 6 contains interface-state information in its upper byte; it also contains the same information as STATUS register 3 in its lower byte. In advanced applications, it may be necessary to detect and act on the interface's current state. Register 6's definition is shown below.

Status Register 6 **Interface Status**

Most Significant Bit

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
REM	LLO	ATN True	LPAS	TPAS	LADS	TADS	*
Value = −32 768	Value = 16 384	Value = 8 192	Value = 4 096	Value = 2 048	Value = 1 024	Value = 512	Value = 256

Most Significant Bit — Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
System Controller	Active Controller	0	Primary Address of Interface				
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

* Least-significant bit of last address recognized

Bit 15 set indicates that the interface is in the Remote state.

Bit 14 set indicates that the interface is in the Local Lockout state.

Bit 13 set indicates that the ATN line is currently set (true).

Bit 12 set indicates that the interface is in the Listener Primary Addressed State (has received its primary listen address and is currently an active listener).

Bit 11 set indicates that the interface is in the Talker Primary Addressed State (has received its primary talk address and is currently an active talker).

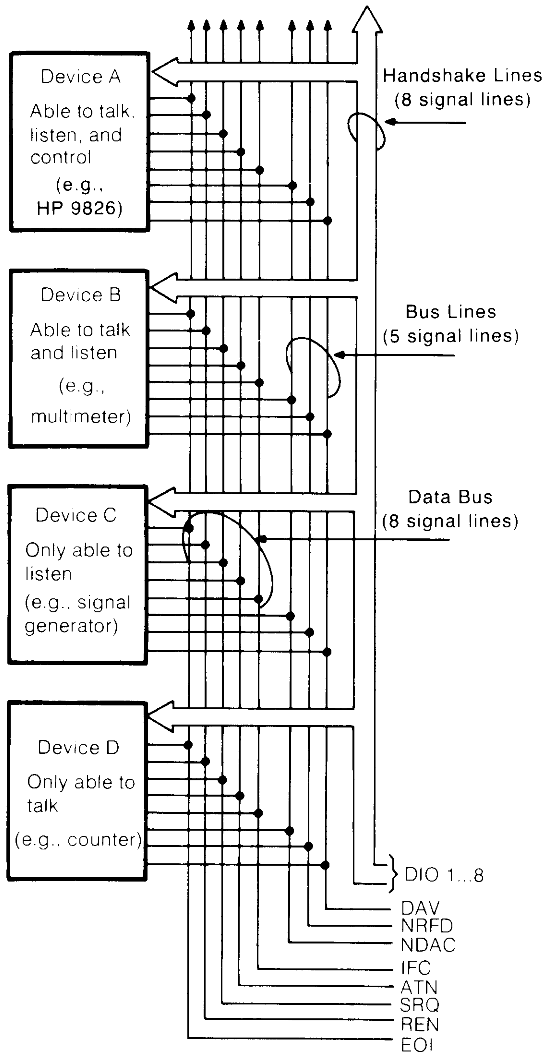
Bit 10 set indicates that the interface is in the Listener Addressed State. This state is entered only after the interface has received a secondary command **and accepted it as valid** by writing a non-zero value to CONTROL register 4 to release the NDAC holdoff.

Bit 9 set indicates that the interface is in the Talker Addressed State. This state is entered in a manner similar to the Listener Addressed State.

Bit 8 contains the least-significant bit of the last address recognized by this interface.

Bits 7 through 0 have the same definitions as STATUS register 3.

HP-IB Control Lines



Handshake Lines

The preceding figure shows the names given to the eight control lines that make up the HP-IB. Three of these lines are designated as the "handshake" lines and are used to control the timing of data byte exchanges so that the talker does not get ahead of the listener(s). The three handshake lines are as follows.

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

The **HP-IB interlocking handshake** uses the lines as follows. All devices currently designated as active listeners would indicate when they are ready for data by using the NRFD line. A device not ready would pull this line low (true) to signal that it is not ready for data, while any device that is ready would let the line float high. Since an active low overrides a passive high, this line will stay low until all active listeners are ready for data.

When the talker senses that all devices are ready, it places the next data byte on the data lines and then pulls DAV low (true). This tells the listeners that the information on the data lines is valid and that they may read it. Each listener then accepts the data and lets the NDAC line float high (false). As with NRFD, only when all listeners have let NDAC go high will the talker sense that all listeners have read the data. It can then float DAV (let it go high) and start the entire sequence over again for the next byte of data.

The Attention Line (ATN)

Command messages are encoded on the data lines as 7-bit ASCII characters, and are distinguished from normal data characters by the logic state of the attention line (ATN). That is, when ATN is **false**, the states of the data lines are interpreted as **data**. When ATN is **true**, the data lines are interpreted as **commands**. The set of 128 ASCII characters that can be placed on the data lines during this ATN-true mode are divided into four classes by the states of data lines DIO6 and DIO7. These classes of commands are shown in a table in the section called ''Advanced Bus Management''.

The Interface Clear Line (IFC)

Only the system controller can set the IFC line true. By asserting IFC, all bus activity is unconditionally terminated, the system controller regains the capability of active controller (if it has been passed to another device), and any current talker and listeners become unaddressed. Normally, this line is only used to terminate all current operations, or to allow the system controller to regain control of the bus. It overrides any other activity that is currently taking place on the bus.

The Remote Enable Line (REN)

This line is used to allow instruments on the bus to be programmed remotely by the active controller. Any device that is addressed to listen while REN is true is placed in the Remote mode of operation.

The End or Identify Line (EOI)

Normally, data messages sent over the HP-IB are sent using the standard ASCII code and are terminated by the ASCII line-feed character, CHR$(10). However, certain devices may wish to send blocks of information that contain data bytes which have the bit pattern of the line-feed character but which are actually part of the data message. Thus, no bit pattern can be designated as a terminating character, since it could occur anywhere in the data stream. For this reason, the EOI line is used to mark the end of the data message.

The EOI line is used during ENTER statements and during an identify sequence (the response to parallel poll). During data messages, the EOI line is set true by the talker to signal that the current data byte is the last one of the data transmission. Generally, when a listener detects that the EOI line is true, it assumes that the data message is concluded. However, EOI may either be used or ignored by the computer when entering data with an ENTER statement that uses an image. Chapter 5 fully describes the definitions of EOI during all ENTER statements and shows how to use the image specifiers that modify the statement-termination conditions.

ENTER statements can use images to re-define the meaning of EOI to provide a very great degree of flexibility. Using the "#" or "%" specifier in an ENTER statement affects the definition of the EOI signal as shown in the following table.

Definition of EOI During ENTER Statements

	Free-field ENTER statements	**ENTER statements that use an image:**		
		without "#" or "%"	with "#"	with "%"
Definition of EOI	Immediate statement terminator	Item separator or statement terminator	Item separator terminator	Immediate statement terminator
Statement terminator required?	Yes	Yes	No	No
Early termination allowed?	No	No	No	Yes

The Service Request Line (SRQ)

The active controller is always in charge of the order of events that occur on the HP-IB. If a device on the bus needs the controller's help, it can set the service request line true. This line sends a request, not a demand, and it is up to the controller to choose when and how it will service that device. However, the device will continue to assert SRQ until it has been "satisfied". Exactly what will satisfy a service request depends on the requesting device, which is explained in the device's operating manual.

Determining Bus-Line States

STATUS register 7 contains the current states of all bus hardware lines. Reading this register returns the states of these lines in the specified numeric variable.

```
STATUS Hpib,7;Bus_lines
```

Status Register 7 **Bus Control and Data Lines**

Most Significant Bit

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
ATN True	DAV True	NDAC* True	NRFD* True	EOI True	SRQ** True	IFC True	REN True
Value = −32 768	Value = 16 384	Value = 8 192	Value = 4 096	Value = 2 048	Value = 1 024	Value = 512	Value = 256

Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DIO8	DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

* Only if addressed to TALK, else not valid.
** Only if Active Controller, else not valid.

Note

Due to the way the bi-directional buffers work, NDAC and NRFD are not accurately read by this STATUS statement unless the interface is currently addressed to talk. Also, SRQ is not accurately shown unless the interface is currently the active controller.

Advanced Bus Management

Bus communication involves both sending data to devices and sending commands to devices and the interface itself. "General Structure of the HP-IB" stated that this communication must be made in an orderly fashion and presented a brief sketch of the differences between data and commands. However, most of the bus operations described so far in this chapter involve sequences of commands and/or data which are sent automatically by the computer when HP-IB statements are executed. This section describes both the commands and data sent by HP-IB statements and how to construct your own, custom bus sequences.

The Message Concept

The main purpose of the bus is to send information between two (or more) devices. These quantities of information sent from talker to listener(s) can be thought of as messages. However, before data can be sent through the bus, it must be properly configured. A sequence of commands is generally sent before the data to inform bus devices which is to send and which is (or are) to listen to the subsequent message(s). These commands can also be thought of as messages.

Most bus messages are transmitted by sending a byte (or sequence of bytes) with numeric values of 0 through 255 through the bus data lines. When the Attention line (ATN) is true, these bytes are considered commands; when ATN is false, they are interpreted as data. Bus command groups and their ASCII characters and codes are shown in "Bus Commands and Codes".

Types of Bus Messages

The messages can be classified into twelve types. This computer is capable of implementing all twelve types of interface messages. The following list describes each type of message.

1. A Data message consists of information which is sent from the talker to the listener(s) through the bus data lines.
2. The Trigger message causes the listening device(s) to initiate device-dependent action(s).
3. The Clear message causes either the listening device(s) or all of the devices on the bus to return to their device-dependent "clear" states.
4. The Remote message causes listening devices to change to remote program control when addressed to listen.
5. The Local message clears the Remote message from the listening device(s) and returns the device(s) to local front-panel control.
6. The Local Lockout message disables a device's front-panel controls, preventing a device's operator from manually interfering with remote program control.
7. The Clear Lockout/Local message causes all devices on the bus to be removed from Local Lockout and to revert to the Local state. This message also clears the Remote message from all devices on the bus.
8. The Service Request message can be sent by a device at any time to signify that the device needs to interact with the the active controller. This message is cleared by sending the device's Status Byte message, if the device no longer requires service.
9. A Status Byte message is a byte that represents the status of a single device on the bus. This byte is sent in response to a serial poll performed by the active controller. Bit 6 indicates whether the device is sending the Service Request message, and the remaining bits indicate other operational conditions of the device.
10. A Status Bit message is a single bit of device-dependent status. Since more than one device can respond on the same line, this Status Bit may be logically combined and/or concatenated with Status Bit messages from many devices. Status Bit messages are returned in response to a parallel poll conducted by the active controller.
11. The Pass Control message transfers the bus management responsibilities from the active controller to another controller.
12. The Abort message is sent by the system controller to assume control of the bus unconditionally from the active controller. This message terminates all bus communications, but is not the same as the Clear message.

These messages represent the full implementation of all HP-IB system capabilities; all of these messages can be sent by this computer. However, each device in a system may be designed to use only the messages that are applicable to its purpose in the system. It is important for you to be aware of the HP-IB functions implemented on each device in your HP-IB system to ensure its operational compatibility with your system.

Bus Commands and Codes

The table below shows the decimal values of IEEE-488 command messages. Remember that **ATN is true** during all of these commands. Notice also that these commands are separated into four general categories: Primary Command Group, Listen Address Group, Talk Address Group, and Secondary Command Group. Subsequent discussions further describe these commands.

Decimal Value	ASCII Character	Interface Message	Description
		PCG	**Primary Command Group**
1	SOH	GTL	Go to Local
4	EOT	SDC	Selected Device Clear
5	ENQ	PPC	Parallel Poll Configure
8	BS	GET	Group Execute Trigger
9	HT	TCT	Take Control
17	DC1	LLO	Local Lockout
20	DC4	DCL	Device Clear
21	NAK	PPU	Parallel Poll Unconfigure
24	CAN	SPE	Serial Poll Enable
25	EM	SPD	Serial Poll Disable
		LAG	**Listen Address Group**
32-62	Space through > (Numbers & Special Chars.)		Listen Addresses 0 through 30
63	?	UNL	Unlisten
		TAG	**Talk Address Group**
64-94	@ through ↑ (Uppercase ASCII)		Talk Addresses 0 through 30
95	_ (underscore)	UNT	Untalk
		SCG	**Secondary Command Group**
96-126	` through ~ (Lowercase ASCII)		Secondary Commands 0 through 30
127	DEL		Ignored

Address Commands and Codes

The following table shows the ASCII characters and corresponding codes of the Listen Address Group and Talk Address Group commands. The next section describes how to send these commands.

Address Characters		Address Code	Address Switch Settings				
Listen	Talk	Decimal	(5)	(4)	(3)	(2)	(1)
Space	@	0	0	0	0	0	0
!	A	1	0	0	0	0	1
”	B	2	0	0	0	1	0
#	C	3	0	0	0	1	1
$	D	4	0	0	1	0	0
%	E	5	0	0	1	0	1
&	F	6	0	0	1	1	0
’	G	7	0	0	1	1	1
(	H	8	0	1	0	0	0
)	I	9	0	1	0	0	1
*	J	10	0	1	0	1	0
+	K	11	0	1	0	1	1
,	L	12	0	1	1	0	0
–	M	13	0	1	1	0	1
.	N	14	0	1	1	1	0
/	O	15	0	1	1	1	1
0	P	16	1	0	0	0	0
1	Q	17	1	0	0	0	1
2	R	18	1	0	0	1	0
3	S	19	1	0	0	1	1
4	T	20	1	0	1	0	0
5	U	21	1	0	1	0	1
6	V	22	1	0	1	1	0
7	W	23	1	0	1	1	1
8	X	24	1	1	0	0	0
9	Y	25	1	1	0	0	1
:	Z	26	1	1	0	1	0
;	[	27	1	1	0	1	1
<	/	28	1	1	1	0	0
=	]	29	1	1	1	0	1
>	↑	30	1	1	1	1	0

The table implicitly shows that:

- listen address commands can be calculated from the primary address by using one of the following equations

```
Listen_address=32+Primary_address
```

or

```
Listen_address$=CHR$(32+Primary_address)
```

- similarly, talk address commands can be calculated from the primary address by using one of the following equations

```
Talk_address=64+Primary_address
```

or

```
Talk_address$=CHR$(64+Primary_address)
```

However, the table does not show that:

- the Unlisten command is "?", CHR$(63)
- the Untalk command is "_", CHR$(95)
- therefore, primary address 31 is an unusable device address, but can be used to send the Unlisten and Untalk commands

Explicit Bus Messages

It is often desirable (or necessary) to manage the bus by sending explicit sequences of bus messages. The SEND statement is the vehicle by which explicit commands and data can be sent through the bus. The SEND statement is also the only method of sending data with odd parity through the bus. This section shows several uses of this statement.

Examples of Sending Commands

As a simple example, suppose the following statement is executed to configure the bus (i.e., to address the talker and listener).

```
OUTPUT 701 USING "#,K"
```

The SEND statement can be used to send the same sequence of commands, as shown in the following statement.

```
SEND 7;CMD "?U!"
```

This statement configures the bus explicitly by sending the following commands:

- the unlisten command (ASCII character "?"; decimal code 63)
- talk address 21 (ASCII character "U"; decimal code 85)
- listen address 1 (ASCII character "!"; decimal code 33)

The same sequence of commands and data is sent with any of the following statements.

```
SEND 7;CMD UNL MTA LISTEN 1

SEND 7;CMD UNL TALK 21 LISTEN 1

SEND 7;CMD 32+31,64+21,32+1
```

Commands can be sent by specifying the secondary keyword CMD. The list of commands (following CMD) be any numeric or string expressions. If more than one expression is listed, they must be separated by commas. A numeric expression will be evaluated, rounded to an integer (MOD 256), and sent as one byte. Each character of a string expression will be sent individually. **All bytes are sent with ATN true.** The computer must be the current active controller to send commands.

```
SEND Isc;CMD 8          ! Group Execute Trigger

SEND 8;CMD 1            ! Go to Local
```

If SEC is used, the specified secondary commands will be sent. An extended talker may be addressed by using SEC after the talk address; extended listener(s) may be addressed by using SEC after the listen address(es).

```
SEND 7;MTA UNL LISTEN 1 CMD 5 SEC 16 ! SEND PPD.
```

The computer must be the active controller to send CMD, LISTEN, UNL, MLA, TALK, UNT, MTA, and SEC. If a non-active controller attempts to send any of these messages, an error is reported.

Simulate the following SPOLL function with a SEND statement.

```
A=SPOLL(724)
```

When an SPOLL is performed, the resulting bus activity is:

- Unlisten command
- My Listen Address (the computer's listen address; MLA)
- device's talk address (one of the TAG commands)
- Serial Poll Enable command (SPE; decimal code 24)
- one data byte is read (the Status Byte message)
- Serial Poll Disable (SPD; decimal code 25)
- Untalk command

This is accomplished by either of the following sequences:

```
SEND 7;CMD "?5X"&CHR$(24)   ! Configure the bus; send SPE.
ENTER 7 USING "#,B";A       ! Read Status Byte.
SEND 7;CMD CHR$(25)&"_"     ! Send Untalk and SPD.

SEND 7;UNL MLA TALK 24 CMD 24
ENTER 7 USING "#,B";A
SEND 7;CMD 25 UNT
```

The preceding secondary keywords provide the capability of sending various command messages through the bus. The activity that results on the bus when several other high-level commands are issued is summarized in "HP-IB Message Mnemonics".

Examples of Sending Data

Data messages can be sent by specifying the secondary keyword DATA. If the computer is the active controller, the data is sent immediately. However, if the computer is not the active controller, it waits to be addressed to talk before sending the data.

```
SEND 7;DATA "Message",13,10   ! Send with CR/LF.

SEND Bus;DATA "Data" END      ! Send with EOI.
```

The data list may contain any mixture of numeric or string expressions; if more than one expression is specified, they must be separated by commas. Each numeric expression is evaluated as an integer (MOD 256) and sent as a single byte. Each string item is evaluated and all resultant characters are sent serially. Each byte is **sent with ATN false** (sent as a data message). The last expression may be followed by the secondary keyword END, which causes the EOI terminator to be sent concurrently with the last data byte.

As another example, simulate this ENTER statement with a SEND statement.

```
ENTER 724;Number,String$
```

Any of the following pairs of statements can be used to accomplish the same operation.

```
SEND 7;UNL TALK 24 MLA
ENTER 7;Number,String$

SEND 7;UNL TALK 24 LISTEN 21
ENTER 7;Number,String$

SEND 7;CMD "?X5"
ENTER 7;Number, String$
```

HP-IB Message Mnemonics

This section contains the descriptions of several bus messages described by the IEEE 488-1978 standard. The following table describes message mnemonics, their meanings, and the secondary keywords used with the SEND statement. The HP-IB messages that require primary keywords are noted in the table.

All BASIC statements which send HP-IB messages (except SEND) always set ATN-true (command) messages with the most-significant bit set to zero. Using CMD (with SEND) allows you to send ATN-true messages with the most-significant bit set to one. This may be useful for non-standard IEEE-488 devices which require the most-significant bit to have a particular value.

The CMD and DATA secondary keywords of SEND statements allow string expressions as well as numeric expressions (e.g., CMD "?" is the same as CMD 63). All other secondary keywords which need data require **numeric** expressions. Keep this in mind while reading through this table.

Message Mnemonic	Message Description	SEND Clause Required (numeric values are decimal)
DAB	Data Byte	DATA 0 through 255
DCL	Device Clear	CMD 20 (or 148)
EOI	End or Identify	DATA data list END
GET	Group Execute Trigger	CMD 8 (or 136)
GTL	Go To Local	CMD 1 (or 129)
IFC	Interface Clear	Not possible with SEND; use the ABORT statement.
LAG	Listen Address	LISTEN 0 through 30; or CMD 32 through 62; or CMD 160 through 190
MLA	My Listen Address	MLA
MTA	My Talk Address	MTA
PPC	Parallel Poll Configure	CMD 5 (or 133)
PPD	Parallel Poll Disable	SEC 16; or CMD 112 (or 240) (Must be preceded by PPC.)
PPE	Parallel Poll Enable	SEC 0+Mask: SEC 0 through 15; or CMD 96 through 111; or CMD 224 through 239 (Must be preceded by PPC.)
PPU	Parallel Poll Unconfigure	CMD 21 (or 149)
PPOLL	Parallel Poll	Not possible with SEND; use the PPOLL function.
REN	Remote Enable	Not possible with SEND; use the REMOTE statement.
SDC	Selected Device Clear	CMD 4 (or 132)
SPD	Serial Poll Disable	CMD 25 (or 153)
SPE	Serial Poll Enable	CMD 24 (or 152)
TAD	Talk Address	TALK 0 through 30; or CMD 64 through 94; or CMD 192 through 222
TCT	Take Control	CMD 9 (or 137)
UNL	Unlisten	UNL; or LISTEN 31; or CMD 63 (or 191)
UNT	Untalk	UNT; or TALK 31; or CMD 95 (or 223)

Servicing Interrupts that Require Data Transfers

During the discussion on interrupts, three special types of interrupt conditions were described (which are enabled by setting bits in CONTROL register 4). These interrupts occur upon receiving: an unrecognized Universal command, an unrecognized addressed command, or a Secondary command. These situations all require the computer to read a byte of information from the bus and respond as desired by the programmer.

Status Register 4 **Interrupt Status**

Most Significant Bit — Least Significant Bit

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
Active Controller	Parallel Poll Configuration Change	My Talk Address Received	My Listen Address Received	EOI Received	SPAS	Remote/ Local Change	Talker/ Listener Address Change
Value = −32 768	Value = 16 384	Value = 8 192	Value = 4 096	Value = 2 048	Value = 1 024	Value = 512	Value = 256

Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Trigger Received	Handshake Error	Unrecognized Universal Command	Secondary Command While Addressed	Clear Received	Unrecognized Addressed Command	SRQ Received	IFC Received
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

As a reminder, these interrupt conditions occur under the following circumstances.

Bit 2 enables an interrupt upon receiving an unrecognized Addressed Command, if addressed to listen. This interrupt is used to detect and respond to commands that are undefined by the standard (but which may be recognized by the computer).

Bit 4 enables an interrupt upon receiving a Secondary Command, if addressed to either talk or listen during the command mode. Again, this allows the computer to detect and respond to special information from another controller.

Bit 5 enables an interrupt upon receiving an unrecognized Universal Command. This interrupt condition provides the computer with the ability to respond to new definitions that may be adopted by the IEEE standards committee.

Bit 14 enables an interrupt on any change in parallel poll configuration. If a Parallel Poll Configure command is received, the computer must set up its own parallel poll response designated by the controller. The response itself is set up by writing to CONTROL register 2 of the HP-IB interface.

Whenever any of the above interrupt conditions are enabled and occur, the computer logs the interrupt and then sets a **bus holdoff**. In other words, all bus activity is "frozen" until the program has released this holdoff. The holdoff is established to allow the program time to determine the current state of the bus.

The bus state is determined by reading HP-IB STATUS register 7, which returns the current logic state of the data and control lines as a 16-bit integer.

```
    STATUS 7,7;Bus_lines
```

After reading the state of the lines, it is necessary to release the bus holdoff by writing any value into HP-IB CONTROL register 4.

```
    CONTROL 7,4;Any_value
```

Control Register 4 **Release NDAC Holdoff**

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0 = Don't Accept Secondary Command All Non-zero Values Accept Secondary (Writing anything to this register releases NDAC holdoff)							
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

When a Secondary Command is received, two computer responses are possible. The first is to accept the address as a valid secondary address and consequently become an Extended Talker. The second is not to accept the address as valid and consequently remain in the primary addressed state.

If Secondary Command interrupts are enabled (while the computer is a non-active controller), the computer will not respond to its primary address alone; a valid secondary address is also required. Statements such as ENTER 7, OUTPUT 7, and LIST #7 should only be executed in the interrupt service routine after CONTROL has been used to indicate that a valid secondary address has been received but before interrupts are re-enabled.

When you no longer want the computer to respond as an Extended Talker/Listener, execute an ENABLE INTR with a mask which has bit 4 equal to zero.

Summary of HP-IB STATUS and CONTROL Registers

Status Register 0 — Card Identification

Most Significant Bit — Least Significant Bit

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

Control Register 0 — Interface Reset

Most Significant Bit — Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Any Bit Will Reset Interface							
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Status Register 1 — Interrupt and DMA Status

Most Significant Bit — Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Interrrupts Enabled	Interrupt Requested	Interrupt Level		0	0	DMA Channel 1 Enabled	DMA Channel 0 Enabled
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Control Register 1 — Serial Poll Response Byte

Most Significant Bit — Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Device Dependent Status	SRQ 1 = I did it 0 = I didn't	Device Dependent Status					
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Status Register 2 — Busy Bits

Most Significant Bit — Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved For Future Use					Handshake In Progress	Interrupts Enabled	Reserved For Future Use
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Control Register 2 — Parallel Poll Response Byte

Most Significant Bit — Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DIO8 1 = True	DIO7 1 = True	DIO6 1 = True	DIO5 1 = True	DIO4 1 = True	DIO3 1 = True	DIO2 1 = True	DIO1 1 = True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Status Register 3 — Controller Status and Address

Most Significant Bit — Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
System Controller	Active Controller	0	Primary Address of Interface				
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Control Register 3 — Set My Addrress

Most Significant Bit — Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not Used			Primary Address				
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Status Register 4 **Interrupt Status**

Most Significant Bit

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
Active Controller	Parallel Poll Configuration Change	My Talk Address Received	My Listen Address Received	EOI Received	SPAS	Remote/ Local Change	Talker/ Listener Address Change
Value = −32 768	Value = 16 384	Value = 8 192	Value = 4 096	Value = 2 048	Value = 1 024	Value = 512	Value = 256

Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Trigger Received	Handshake Error	Unrecognized Universal Command	Secondary Command While Addressed	Clear Received	Unrecognized Addressed Command	SRQ Received	IFC Received
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Control Register 4 **Release NDAC Holdoff**

Most Significant Bit — Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0 = Don't Accept Secondary Command All Non-zero Values Accept Secondary (Writing anything to this register releases NDAC holdoff)							
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Status Register 5 — Interrupt Enable Mask

Most Significant Bit

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
Active Controller	Parallel Poll Configuration Change	My Talk Address Received	My Listen Address Received	EOI Received	SPAS	Remote/ Local Change	Talker/ Listener Address Change
Value = −32 768	Value = 16 384	Value = 8 192	Value = 4 096	Value = 2 048	Value = 1 024	Value = 512	Value = 256

Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Trigger Received	Handshake Error	Unrecognized Universal Command	Secondary Command While Addressed	Clear Received	Unrecognized Addressed Command	SRQ Received	IFC Received
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Status Register 6 — Interface Status

Most Significant Bit

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
REM	LLO	ATN True	LPAS	TPAS	LADS	TADS	*
Value = −32 768	Value = 16 384	Value = 8 192	Value = 4 096	Value = 2 048	Value = 1 024	Value = 512	Value = 256

Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
System Controller	Active Controller	0	Primary Address of Interface				
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

* Least-significant bit of last address recognized

Status Register 7 — Bus Control and Data Lines

Most Significant Bit

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
ATN True	DAV True	NDAC* True	NRFD* True	EOI True	SRQ** True	IFC True	REN True
Value = −32 768	Value = 16 384	Value = 8 192	Value = 4 096	Value = 2 048	Value = 1 024	Value = 512	Value = 256

Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DIO8	DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

* Only if addressed to TALK, else not valid.
** Only if Active Controller, else not valid.

Summary of HP-IB READIO and WRITEIO Registers

READIO Registers

Register 1 — Card Identification
Register 3 — Interrupt and DMA Status
Register 5 — Controller Status and Address
Register 17 — Interrupt Status 0[1]
Register 19 — Interrupt Status 1[1]
Register 21 — Interface Status
Register 23 — Control-Line Status
Register 29 — Command Pass-Through
Register 31 — Data-Line Status[1]

HP-IB READIO Register 1 — Card Identification

Most Significant Bit — Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Future Use Jumper Installed	0	0	0	0	0	0	1
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Bit 7 is set (1) if the "future use" jumper is installed and clear (0) if not.

Bits 6 through 0 constitute a card identification code (= 1 for all HP-IB cards).

Note

This register is only implemented on external HP-IB cards. The internal HP-IB, at interface select code 7, "floats" this register (i.e., the states of all bits are indeterminate).

HP-IB READIO Register 3 — Interrupt and DMA Status

Most Significant Bit — Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Interrupt Enabled	Interrupt Request	Interrupt Level		X	X	DMA1	DMA0
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

1 Indicates that a READIO operation will change the state of the interface.

Bit 7 is set (1) if interrupts are currently enabled.

Bit 6 is set (1) when the card is currently requesting service.

Bits 5 and 4 constitute the card's hardware interrupt level (a switch setting on all external cards, but fixed at level 3 on the internal HP-IB).

Bit 5	Bit 4	Hardware Interrupt Level
0	0	3
0	1	4
1	0	5
1	1	6

Bits 3 and 2 are not used (indeterminate).

Bit 1 is set (1) if DMA channel one is currently enabled.

Bit 0 is set (1) if DMA channel zero is currently enabled.

Note

Bits 7, 5, 4, 3, 2, and 1 are not implemented on the internal HP-IB (interface select code 7).

HP-IB READIO Register 5 **Controller Status and Address**

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
System Controller	Not Active Controller	X	← HP-IB Primary Address of Interface (MSB)				(LSB) →
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Bit 7 is set (1) if the interface is the System Controller.

Bit 6 is set (1) if the interface is **not** the current Active Controller and clear (0) if it **is** the Active Controller.

Bit 5 is not used.

Bits 4 through 0 contain the card's Primary Address switch setting. The following bit patterns indicate the specified addresses.

Bit 4 3 2 1 0	Primary Address
0 0 0 0 0	0
0 0 0 0 1	1
.	.
.	.
.	.
1 1 1 0 1	29
1 1 1 1 0	30
1 1 1 1 1	(not allowed)

Note

Bits 5 through 0 are not implemented on the internal HP-IB.

HP-IB READIO Register 17 **MSB of Interrupt Status**

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit Ø
MSB Interrupt	LSB Interrupt	Byte Received	Ready for Next Byte	End Detected	SPAS	Remote/ Local Change	My Address Change
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Bit 7 set (1) indicates that an interrupt has occurred whose cause can be determined by reading the contents of this register.

Bit 6 set (1) indicates that an interrupt has occurred whose cause can be determined by reading Interrupt Status Register 1 (READIO Register 19).

Bit 5 set (1) indicates that a data byte has been received.

Bit 4 set (1) indicates that this interface is ready to accept the next data byte.

Bit 3 set (1) indicates that an End (EOI with ATN = 0) has been detected.

Bit 2 set (1) indicates that the Serial-Poll-Active State has been entered.

Bit 1 set (1) indicates that a Remote/Local State change has occurred.

Bit 0 set (1) indicates that a change in My Address has occurred.

HP-IB READIO Register 19 — LSB of Interrupt Status

Most Significant Bit — Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Trigger Received	Handshake Error	Unrecognized Universal Command	Secondary Command While Addressed	Clear Received	Unrecognized Addressed Command	SRQ Received	IFC Received
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Bit 7 set (1) indicates that a Group Execute Trigger command has been received.

Bit 6 set (1) indicates that an Incomplete-Source-Handshake error has occurred.

Bit 5 set (1) indicates that an unidentified command has been received.

Bit 4 set (1) indicates that a Secondary Address has been sent in while in the extended-addressing mode.

Bit 3 set (1) indicates that the interface has entered the Device-Clear-Active State.

Bit 2 set (1) indicates that My Address has been received.

Bit 1 set (1) indicates that a Service Request has been received.

Bit 0 set (1) indicates that the Inteface Clear message has been received.

HP-IB READIO Register 21 — Interface Status

Most Significant Bit — Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
REM	LLO	ATN True	LPAS	TPAS	LADS	TADS	LSB of Last Address
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Bit 7 set (1) indicates that this Interface is in the Remote State.

Bit 6 set (1) indicates that this interface is in the Local Lockout State.

Bit 5 set (1) indicates that the ATN signal line is true.

Bit 4 set (1) indicates that this interface is in the Listener-Primary-Addressed State.

Bit 3 set (1) indicates that this interface is in the Talker-Primary-Addressed State.

Bit 2 set (1) indicates that this interface is in the Listener-Addressed State.

Bit 1 set (1) indicates that this interface is in the Talker-Addressed State.

Bit 0 set (1) indicates that this is the least-significant bit of the last address recognized by this interface.

HP-IB READIO Register 23 Control-Line Status

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ATN True	DAV True	NDAC* True	NRFD* True	EOI True	SRQ** True	IFC True	REN True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

*Only if addressed to TALK, else not valid.

**Only if Active Controller, else not valid.

A set bit (1) indicates that the corresponding line is currently true; a 0 indicates that the line is currently false.

HP-IB READIO Register 29 Command Pass-Through

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DIO8	DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

This register can be read during a bus holdoff to determine which Secondary Command has been detected.

HP-IB READIO Register 31 Bus Data Lines

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DIO8	DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

A set bit (1) indicates that the corresponding HP-IB data line is currently true; a 0 indicates the line is currently false.

HP-IB WRITEIO Registers

Register 3 — Interrupt Enable
Register 17 — MSB of Interrupt Mask
Register 19 — LSB of Interrupt Mask
Register 23 — Auxiliary Command Register
Register 25 — Address Register
Register 27 — Serial Poll Response
Register 29 — Parallel Poll Response
Register 31 — Data Out Register

HP-IB WRITEIO Register 3 — Interrupt Enable

Most Significant Bit — Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Enable Interrupt	X	X	X	X	X	Enable Channel 1	Enable Channel 0
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Bit 7 enables interrupts from this interface if set (1) and disables interrupts if clear (0).

Bits 6 through 2 are "don't cares" (i.e., their values have no effect on the interface's operation).

Bit 1 enables DMA channel 1 if set (1) and disables if clear (0).

Bit 0 enables DMA channel 0 if set (1) and disables if clear (0).

Note

Bits 7 through 1 are not implemented on the internal HP-IB interface and thus have no effect on the interface's operation.

WRITEIO Register 17 — MSB of Interrupt Mask

Setting a bit of this register enables an interrupt for the specified condition. The bit assignments are the same as for the MSB of Interrupt Status Register (READIO Register 17), except that bits 7 and 6 are not used.

WRITEIO Register 19 — LSB of Interrupt Mask

Setting a bit of this register enables an interrupt for the specified condition. The bit assignments are the same as for the LSB of Interrupt Status Register (READIO Register 19).

HP-IB WRITEIO Register 23 **Auxiliary Command Register**

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Set	X	X	Auxiliary Command Function				
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Bit 7 is set (1) for a Set operation and clear (0) for a Clear operation.

Bits 6 and 5 are "don't cares".

Bits 4 through 0 are Auxiliary-Command-Function-Select bits. The following commands can be sent to the interface by sending the specified numeric values.

Decimal Value		Description of Auxiliary Command
0	—	Clear Chip Reset.
128	—	Set Chip Reset.
1	—	Release ACDS holdoff. If Address Pass Through is set, it indicates an invalid secondary has been received.
129	—	Release ACDS holdoff; If Address Pass Through is set, indicates a valid secondary has been received.
2	—	Release RFD holdoff.
130	—	Same command as decimal 2 (above).
3	—	Clear holdoff on all data.
131	—	Set holdoff on all data.
4	—	Clear holdoff on EOI only.
132	—	Set holdoff on EOI only.
5	—	Set New Byte Available (nba) false.
133	—	Same command as decimal 5 (above).
6	—	Pulse the Group Execute Trigger line, or clear the line if it was set by decimal command 134.
134	—	Set Group Execute Trigger line.
7	—	Clear Return To Local (rtl).
135	—	Set Return To Local (must be cleared before the device is able to enter the Remote state).
8	—	Causes EOI to be sent with the next data byte.
136	—	Same command as decimal 8 (above).
9	—	Clear Listener State (also cleared by decimal 138).
137	—	Set Listener State.
10	—	Clear Talker State (also cleared by decimal 137).
138	—	Set Talker State.

(Continued)

Decimal Value		Description of Auxiliary Command
11	—	Go To Standby (gts; controller sets ATN false).
139	—	Same command as decimal 11 (above).
12	—	Take Control Asynchronously (tca; ATN true).
140	—	Same command as decimal 12 (above).
13	—	Take Control Synchronously (tcs; ATN true).
141	—	Same command as decimal 13 (above).
14	—	Clear Parallel Poll.
142	—	Set Parallel Poll (read Command-Pass-Through register before clearing).
15	—	Clear the Interface Clear line (IFC).
143	—	Set Interface Clear (IFC maintained >100 μs).
16	—	Clear the Remote Enable (REN) line.
144	—	Set Remote Enable.
17	—	Request control (after TCT is decoded, issue this to wait for ATN to drop and receive control).
145	—	Same command as decimal 17 (above).
18	—	Release control (issued after sending TCT to complete a Pass Control and set ATN false).
146	—	Same command as decimal 18 (above).
19	—	Enable all interrupts.
147	—	Disable all interrupts.
20	—	Pass Through next Secondary Command.
148	—	Same command as decimal 20 (above).
21	—	Set T1 delay to 10 clock cycles (2 μs at 5 MHz).
149	—	Set T1 delay to 6 clock cycles (1.2 μs at 5 MHz).
22	—	Clear Shadow Handshake.
150	—	Set Shadow Handshake.

HP-IB WRITEIO Register 25 **Address Register**

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Enable Dual Addressing	Disable Listen	Disable Talker	Primary Address				
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Bit 7 set (1) enables the Dual-Primary-Addressing Mode.

Bit 6 set (1) invokes the Disable-Listen function.

Bit 5 set (1) invokes the Disable-Talker function

Bits 4 through 0 set the device's Primary Address (same address bit definitions as READIO Register 5).

HP-IB WRITEIO Register 27 **Serial Poll Response Byte**

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Device Dependent Status	Request Service	Device-Dependent Status					
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Bits 7 and 5—0 specify the Device-Dependent Status.

Bit 6 sends an SRQ if set (1).

Note

Given an unknown state of the Serial Poll Response Byte, it is necessary to write the byte with bit 6 set to zero followed by a write of the byte with bit 6 set to the desired final value. This will insure that a SRQ will be generated if one was desired.

HP-IB WRITEIO Register 29 **Parallel Poll Response**

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DIO8	DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

A 1 sets the appropriate bit true during a Parallel Poll; a 0 sets the corresponding bit false. Initially, and when Parallel Poll is not configured, this register must be set to all zeros.

HP-IB WRITEIO Register 31 **Data-Out Register**

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DIO8	DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Summary of Bus Sequences

The following tables show the bus activity invoked by executing HP-IB statements and functions. The mnemonics used in these tables were defined in the previous section of this chapter.

Note that the bus messages are sent by using single lines (such as the ATN line) and multi-line commands (such as DCL). The information shows the state of and changes in the state of the ATN line during these bus sequences. The tables implicitly show that these **changes in the state of ATN remain in effect unless another change is explicitly shown in the table**. For example, if a statement sets ATN (true) with a particular command, it remains true unless the table explicitly shows that it is set false (ATN). The ATN line is implememted in this manner to avoid unnecessary transitions in this signal whenever possible. It should not cause any dilemmas in most cases.

ABORT

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active Controller	IFC (duration ≥100μsec) REN $\overline{\text{ATN}}$		ATN MTA UNL $\overline{\text{ATN}}$	
Not Active Controller	IFC (duration ≥100 μsec)* REN $\overline{\text{ATN}}$		No Action	

* The IFC message allows a non-active controller (which is the system controller) to become the active controller.

CLEAR

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active Controller	ATN DCL	ATN MTA UNL LAG SDC	ATN DCL	ATN MTA UNL LAG SDC
Not Active Controller	Error			

LOCAL

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active Controller	$\overline{\text{REN}}$ $\overline{\text{ATN}}$	ATN MTA UNL LAG GTL	ATN GTL	ATN MTA UNL LAG GTL
Not Active Controller	$\overline{\text{REN}}$ $\overline{\text{ATN}}$	Error	Error	

LOCAL LOCKOUT

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active Controller	ATN LLO	Error	ATN LLO	Error
Not Active Controller	Error			

PPOLL

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active Controller	ATN & EOI (duration$\geq$25μs) Read byte $\overline{\text{EOI}}$ Restore ATN to previous state	Error	ATN & EOI (duration$\geq$25μs) Read byte $\overline{\text{EOI}}$ Restore ATN to previous state	Error
Not Active Controller	Error			

PPOLL CONFIGURE

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active Controller	Error	ATN MTA UNL LAG PPC PPE	Error	ATN MTA UNL LAG PPC PPE
Not Active Controller	Error			

PPOLL UNCONFIGURE

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active Controller	ATN PPU	ATN MTA UNL LAG PPC PPD	ATN PPU	ATN MTA UNL LAG PPC PPD
Not Active Controller	Error			

REMOTE

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active Controller	REN	REN ATN MTA UNL LAG	Error	
Not Active Controller	REN	Error	Error	

SPOLL

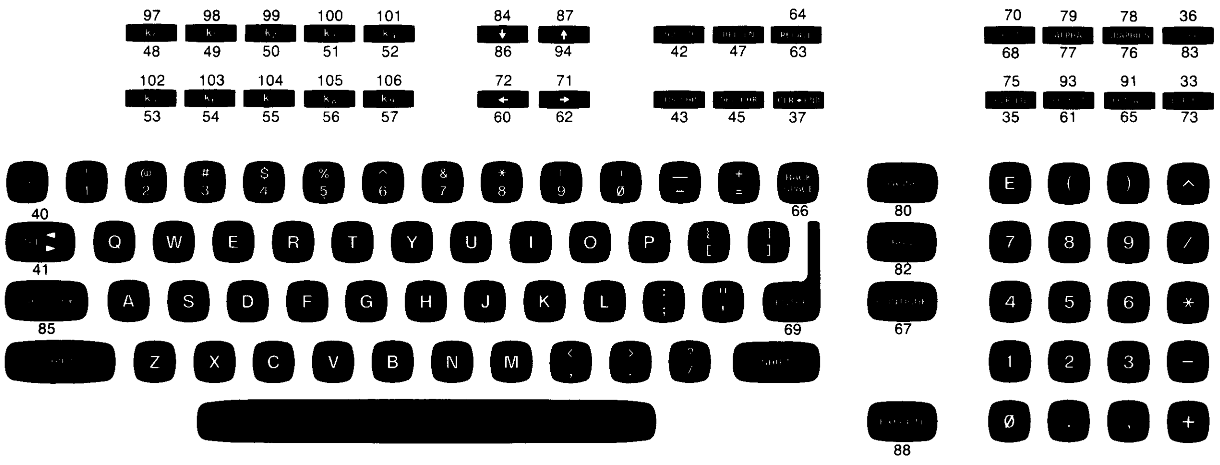
	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active Controller	Error	ATN UNL MLA TAD SPE $\overline{\text{ATN}}$ Read data ATN SPD UNT	Error	ATN UNL MLA TAD SPE $\overline{\text{ATN}}$ Read data ATN SPD UNT
Not Active Controller	Error			

TRIGGER

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active Controller	ATN GET	ATN MTA UNL LAG GET	ATN GET	ATN MTA UNL LAG GET
Not Active Controller	Error			

Appendix A

Non-ASCII Key Output Codes



To output non-ASCII keys to the keyboard, precede each key code with CHR$(255) in an OUTPUT statement (directed to interface select code 2).

US ASCII Character Codes

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Binary	Oct	Hex	Dec	
NULL	00000000	000	00	0	
SOH	00000001	001	01	1	GTL
STX	00000010	002	02	2	
ETX	00000011	003	03	3	
EOT	00000100	004	04	4	SDC
ENQ	00000101	005	05	5	PPC
ACK	00000110	006	06	6	
BELL	00000111	007	07	7	
BS	00001000	010	08	8	GET
HT	00001001	011	09	9	TCT
LF	00001010	012	0A	10	
VT	00001011	013	0B	11	
FF	00001100	014	0C	12	
CR	00001101	015	0D	13	
SO	00001110	016	0E	14	
SI	00001111	017	0F	15	
DLE	00010000	020	10	16	
DC1	00010001	021	11	17	LLO
DC2	00010010	022	12	18	
DC3	00010011	023	13	19	
DC4	00010100	024	14	20	DCL
NAK	00010101	025	15	21	PPU
SYNC	00010110	026	16	22	
ETB	00010111	027	17	23	
CAN	00011000	030	18	24	SPE
EM	00011001	031	19	25	SPD
SUB	00011010	032	1A	26	
ESC	00011011	033	1B	27	
FS	00011100	034	1C	28	
GS	00011101	035	1D	29	
RS	00011110	036	1E	30	
US	00011111	037	1F	31	

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Binary	Oct	Hex	Dec	
space	00100000	040	20	32	LA0
!	00100001	041	21	33	LA1
"	00100010	042	22	34	LA2
#	00100011	043	23	35	LA3
$	00100100	044	24	36	LA4
%	00100101	045	25	37	LA5
&	00100110	046	26	38	LA6
'	00100111	047	27	39	LA7
(	00101000	050	28	40	LA8
)	00101001	051	29	41	LA9
*	00101010	052	2A	42	LA10
+	00101011	053	2B	43	LA11
,	00101100	054	2C	44	LA12
−	00101101	055	2D	45	LA13
.	00101110	056	2E	46	LA14
/	00101111	057	2F	47	LA15
0	00110000	060	30	48	LA16
1	00110001	061	31	49	LA17
2	00110010	062	32	50	LA18
3	00110011	063	33	51	LA19
4	00110100	064	34	52	LA20
5	00110101	065	35	53	LA21
6	00110110	066	36	54	LA22
7	00110111	067	37	55	LA23
8	00111000	070	38	56	LA24
9	00111001	071	39	57	LA25
:	00111010	072	3A	58	LA26
;	00111011	073	3B	59	LA27
<	00111100	074	3C	60	LA28
=	00111101	075	3D	61	LA29
>	00111110	076	3E	62	LA30
?	00111111	077	3F	63	UNL

US ASCII Character Codes (Continued)

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Binary	Oct	Hex	Dec	
@	01000000	100	40	64	TA0
A	01000001	101	41	65	TA1
B	01000010	102	42	66	TA2
C	01000011	103	43	67	TA3
D	01000100	104	44	68	TA4
E	01000101	105	45	69	TA5
F	01000110	106	46	70	TA6
G	01000111	107	47	71	TA7
H	01001000	110	48	72	TA8
I	01001001	111	49	73	TA9
J	01001010	112	4A	74	TA10
K	01001011	113	4B	75	TA11
L	01001100	114	4C	76	TA12
M	01001101	115	4D	77	TA13
N	01001110	116	4E	78	TA14
O	01001111	117	4F	79	TA15
P	01010000	120	50	80	TA16
Q	01010001	121	51	81	TA17
R	01010010	122	52	82	TA18
S	01010011	123	53	83	TA19
T	01010100	124	54	84	TA20
U	01010101	125	55	85	TA21
V	01010110	126	56	86	TA22
W	01010111	127	57	87	TA23
X	01011000	130	58	88	TA24
Y	01011001	131	59	89	TA25
Z	01011010	132	5A	90	TA26
[	01011011	133	5B	91	TA27
\	01011100	134	5C	92	TA28
]	01011101	135	5D	93	TA29
^	01011110	136	5E	94	TA30
_	01011111	137	5F	95	UNT

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Binary	Oct	Hex	Dec	
`	01100000	140	60	96	SC0
a	01100001	141	61	97	SC1
b	01100010	142	62	98	SC2
c	01100011	143	63	99	SC3
d	01100100	144	64	100	SC4
e	01100101	145	65	101	SC5
f	01100110	146	66	102	SC6
g	01100111	147	67	103	SC7
h	01101000	150	68	104	SC8
i	01101001	151	69	105	SC9
j	01101010	152	6A	106	SC10
k	01101011	153	6B	107	SC11
l	01101100	154	6C	108	SC12
m	01101101	155	6D	109	SC13
n	01101110	156	6E	110	SC14
o	01101111	157	6F	111	SC15
p	01110000	160	70	112	SC16
q	01110001	161	71	113	SC17
r	01110010	162	72	114	SC18
s	01110011	163	73	115	SC19
t	01110100	164	74	116	SC20
u	01110101	165	75	117	SC21
v	01110110	166	76	118	SC22
w	01110111	167	77	119	SC23
x	01111000	170	78	120	SC24
y	01111001	171	79	121	SC25
z	01111010	172	7A	122	SC26
{	01111011	173	7B	123	SC27
|	01111100	174	7C	124	SC28
}	01111101	175	7D	125	SC29
~	01111110	176	7E	126	SC30
DEL	01111111	177	7F	127	SC31

European Display Characters

Character	Decimal Value	Character	Decimal Value	Character	Decimal Value
ʰꜰ	128	ʰꜰ	173	Ö	218
ʰꜰ	129	ʰꜰ	174	Ü	219
ʰꜰ	130	₤	175	É	220
ʰꜰ	131	‾	176	ï	221
ʰꜰ	132	ʰꜰ	177	ß	222
ʰꜰ	133	ʰꜰ	178	ʰꜰ	223
ʰꜰ	134	°	179	ʰꜰ	224
ʰꜰ	135	ʰꜰ	180	ʰꜰ	225
ʰꜰ	136	Ç	181	ʰꜰ	226
ʰꜰ	137	Ñ	182	ʰꜰ	227
ʰꜰ	138	ñ	183	ʰꜰ	228
ʰꜰ	139	¡	184	ʰꜰ	229
ʰꜰ	140	¿	185	ʰꜰ	230
ʰꜰ	141	¤	186	ʰꜰ	231
ʰꜰ	142	£	187	ʰꜰ	232
ʰꜰ	143	ʰꜰ	188	ʰꜰ	233
ʰꜰ	144	§	189	ʰꜰ	234
ʰꜰ	145	ʰꜰ	190	ʰꜰ	235
ʰꜰ	146	ʰꜰ	191	ʰꜰ	236
ʰꜰ	147	â	192	ʰꜰ	237
ʰꜰ	148	ê	193	ʰꜰ	238
ʰꜰ	149	ô	194	ʰꜰ	239
ʰꜰ	150	û	195	ʰꜰ	240
ʰꜰ	151	á	196	ʰꜰ	241
ʰꜰ	152	é	197	ʰꜰ	242
ʰꜰ	153	ó	198	ʰꜰ	243
ʰꜰ	154	ú	199	ʰꜰ	244
ʰꜰ	155	à	200	ʰꜰ	245
ʰꜰ	156	è	201	ʰꜰ	246
ʰꜰ	157	ò	202	ʰꜰ	247
ʰꜰ	158	ù	203	ʰꜰ	248
ʰꜰ	159	ä	204	ʰꜰ	249
ʰꜰ	160	ë	205	ʰꜰ	250
ʰꜰ	161	ö	206	ʰꜰ	251
ʰꜰ	162	ü	207	ʰꜰ	252
ʰꜰ	163	Å	208	ʰꜰ	253
ʰꜰ	164	î	209	ʰꜰ	254
ʰꜰ	165	Ø	210	🄺	255
ʰꜰ	166	Æ	211		
ʰꜰ	167	å	212		
´	168	í	213		
`	169	ø	214		
ˆ	170	æ	215		
¨	171	Ä	216		
˜	172	ì	217		

Katakana Display Characters

Character	Decimal Value
hp	128
hp	129
hp	130
hp	131
hp	132
hp	133
hp	134
hp	135
hp	136
hp	137
hp	138
hp	139
hp	140
hp	141
hp	142
hp	143
hp	144
hp	145
hp	146
hp	147
hp	148
hp	149
hp	150
hp	151
hp	152
hp	153
hp	154
hp	155
hp	156
hp	157
hp	158
hp	159
hp	160
｡	161
｢	162
｣	163
､	164
･	165
ｦ	166
ｧ	167
ｨ	168
ｩ	169
ｪ	170
ｫ	171
ｬ	172
ｭ	173
ｮ	174
ｯ	175
ｰ	176
ｱ	177
ｲ	178
ｳ	179
ｴ	180
ｵ	181
ｶ	182
ｷ	183
ｸ	184
ｹ	185
ｺ	186
ｻ	187
ｼ	188
ｽ	189
ｾ	190
ｿ	191
ﾀ	192
ﾁ	193
ﾂ	194
ﾃ	195
ﾄ	196
ﾅ	197
ﾆ	198
ﾇ	199
ﾈ	200
ﾉ	201
ﾊ	202
ﾋ	203
ﾌ	204
ﾍ	205
ﾎ	206
ﾏ	207
ﾐ	208
ﾑ	209
ﾒ	210
ﾓ	211
ﾔ	212
ﾕ	213
ﾖ	214
ﾗ	215
ﾘ	216
ﾙ	217
ﾚ	218
ﾛ	219
ﾜ	220
ﾝ	221
ﾞ	222
ﾟ	223
hp	224
hp	225
hp	226
hp	227
hp	228
hp	229
hp	230
hp	231
hp	232
hp	233
hp	234
hp	235
hp	236
hp	237
hp	238
hp	239
hp	240
hp	241
hp	242
hp	243
hp	244
hp	245
hp	246
hp	247
hp	248
hp	249
hp	250
hp	251
hp	252
hp	253
hp	254
$\boxed{K}$	255

HP 9826 BASIC Interfacing Techniques

HEWLETT PACKARD

Part No. 09826-90020
Microfiche No. 09826-99020

Printed in U.S.A.
First Edition, October 1981