HEWLETT PACKARD

# BASIC 3.0 Interfacing Techniques

BASIC 3.0 Interfacing Techniques
for the HP 9000 Series 200 Computers

Manual Part No. 98613-90020

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.

May 1984...First Edition
November 1984...Update
December 1984...Edition 1 with update

Table of Contents

Chapter 5: Entering Data

Chapter 6: Registers

Chapter 7: Interface Events

Chapter 8: The Internal CRT Interface

Chapter 9: The Internal Keyboard Interface

Chapter 10: I/O Path Attributes

Chapter 11: Advanced Transfer Techniques

Chapter 14: The RS-232 Serial Interface

Chapter 15: Powerfail Protection

Chapter 16: The GPIO Interface

Chapter 18: EPROM Programming

Figures

Tables

xviii

Manual Overview	Chapter
	1

Introduction

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

Note

It may be necessary, depending on your computer system, to add certain drivers or keywords which are provided by loading BIN files.

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 is designed for easy access by both beginners and experienced users. Experienced users may decide to go directly to the chapter that describes the interface to be used. If more background is required, the information in chapters 3 through 7 will provide further explanation. Less experienced users may want to begin with Chapter 2, "Interfacing Concepts", before reading about general or interface-specific techniques. It is recommended that less experienced users work through the *BASIC Programming Techniques* manual before using this manual.

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

Chapter Previews

Chapter 2 - Interfacing Concepts
This chapter presents a brief explanation of relevant interfacing concepts and terminology. This discussion is especially useful for beginners as it covers much of the why and how of 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.

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

Chapter 11 - Advanced Transfer Techniques
This chapter describes advanced I/O techniques which can be used when communicating with devices. These techniques are generally used with devices which have data-transfer rates either much faster or much slower than the computer's normal transfer rate(s).

Chapter 12 - The HP-IB Interface
This chapter describes programming techniques specific to the HP-IB interface. Details of HP-IB communications processes are also included to promote better overall understanding of how this interface may be used.

Chapter 13 - The Datacomm Interface
This chapter describes the HP 98628 Data Communications Interface and presents programming techniques for using the asynchronous or HP Data Link protocols provided by this interface.

Chapter 14 - RS-232 Serial Interface
This chapter describes programming techniques specific to using the asynchronous-protocol capabilities of the HP 98626 Serial Interface.

Chapter 15 - Powerfail Protection
This chapter describes programming techniques for achieving powerfail protection (Option 050 is required to use these capabilities).

Chapter 16 - The GPIO Interface
This chapter describes programming techniques specific to using the HP 98622 GPIO Interface.

Chapter 17 - The BCD Interface
This chapter describes programming techniques specific to using the HP 98623 BCD Interface.

Chapter 18 - EPROM Programming
This chapter describes how to program EPROMs (erasable programmable read only memory) using the HP 98255 EPROM Memory Card(s) and the HP 98253 EPROM Programmer Card.

Interfacing Concepts	Chapter
	2

Introduction

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

Terminology

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

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.
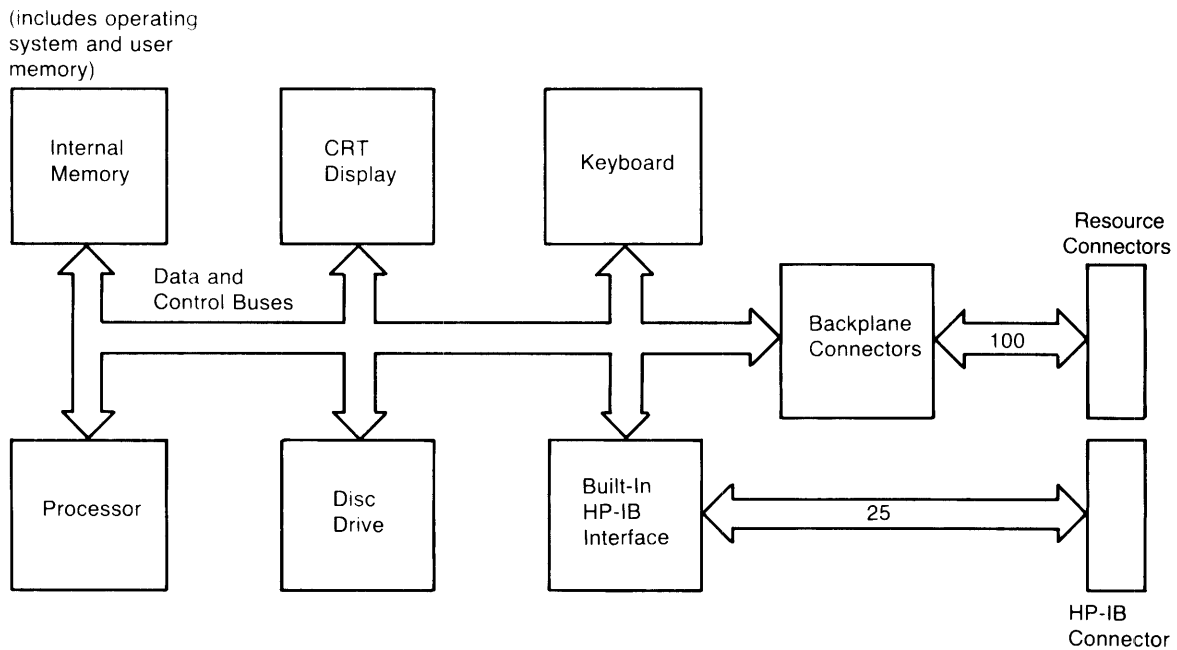
Block Diagram of the Computer

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 devices through interfaces connected to the backplane hardware.
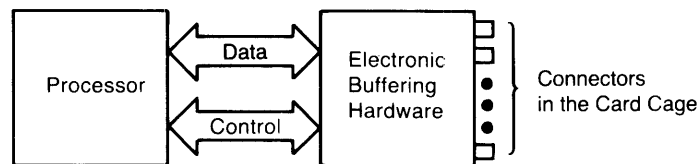
The Processor Communicates with the Interfaces through Backplane Hardware

Backplane Hardware

Why Do You Need an Interface?

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

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

CAUTION
EXCEEDING BACKPLANE HARDWARE RATINGS WILL DAMAGE THE 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.
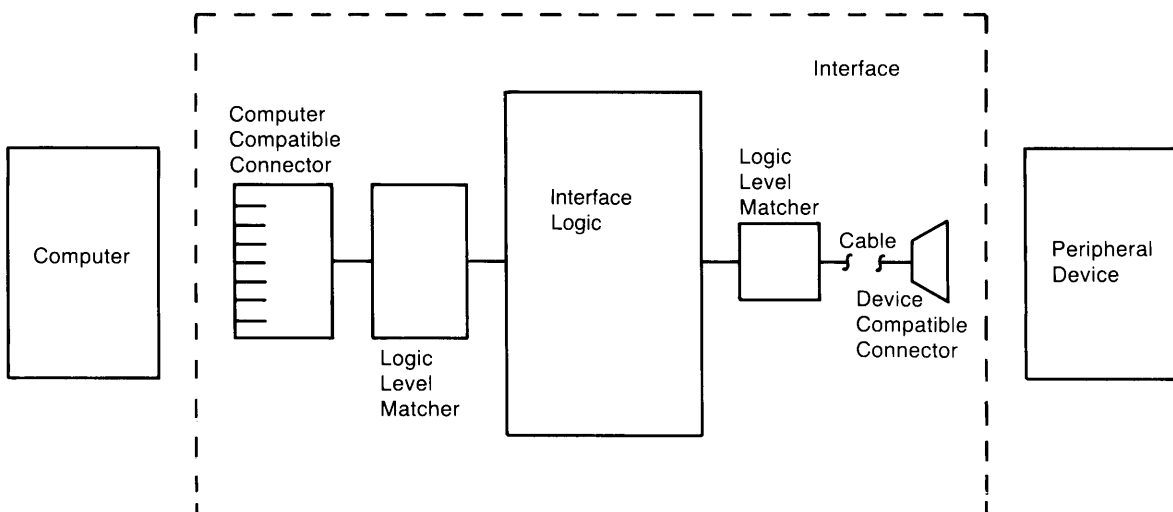
Functional Diagram of an Interface

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

Data Compatibility

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

Timing Compatibility

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

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

Additional Interface Functions

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

Interface Overview

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

The HP-IB Interface

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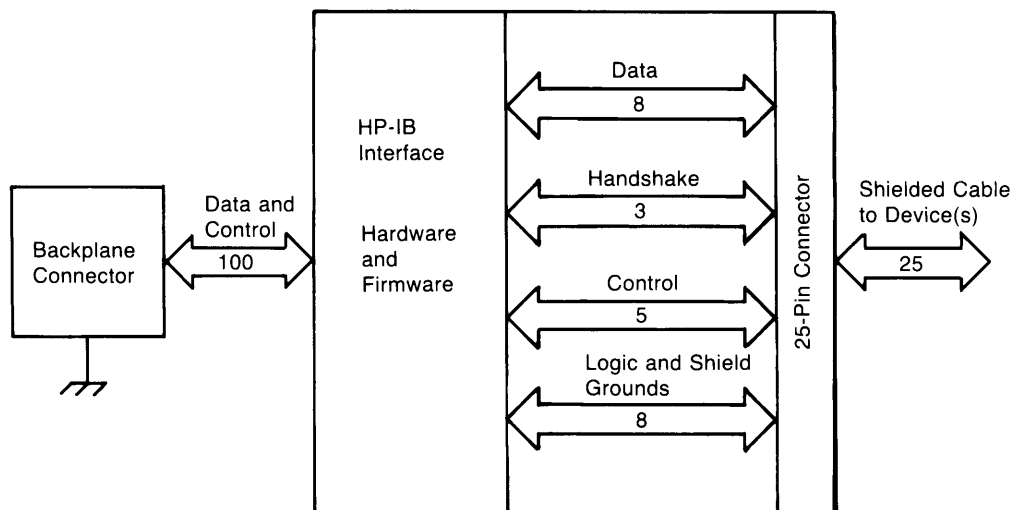
Block Diagram of the HP-IB Interface

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

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

The RS-232 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.
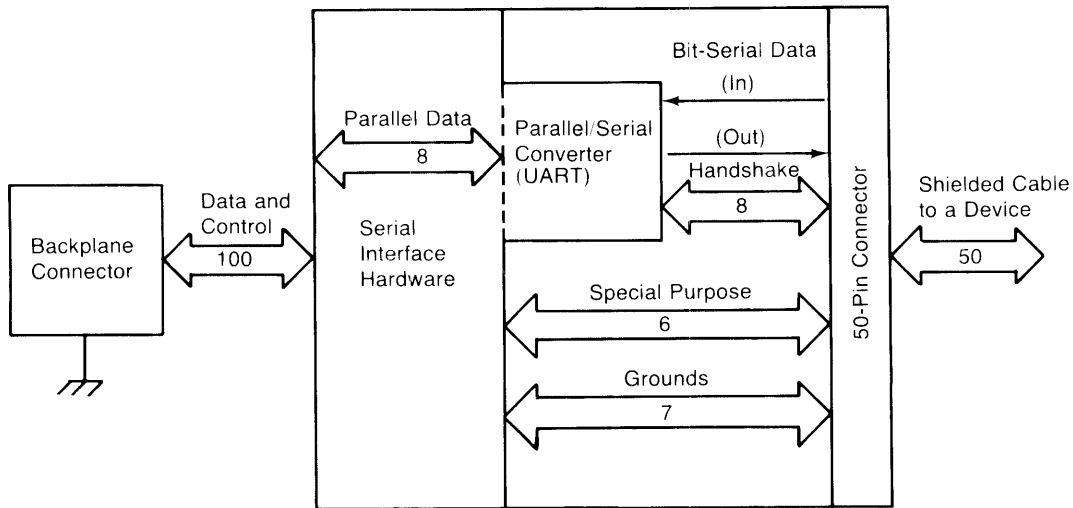
Block Diagram of the Serial Interface

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

The Datacomm Interface

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

The GPIO Interface

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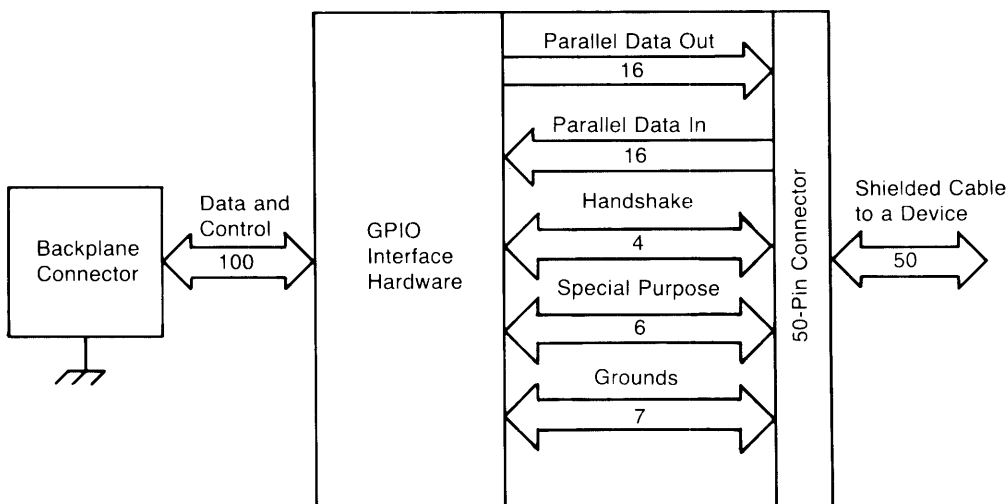
Block Diagram of the GPIO Interface

The BCD Interface

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

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 (**binary digits**), each of which can take on one of two logic levels (high or low). Depending on how the computer interprets these bits, they may mean on or not on (off), true or not true (false), one or zero, busy or not busy, or any other bi-state condition. These logic levels are actually voltage levels of hardware locations within the computer. The following diagram shows the voltage of a point versus time and relates the voltage levels to logic levels.
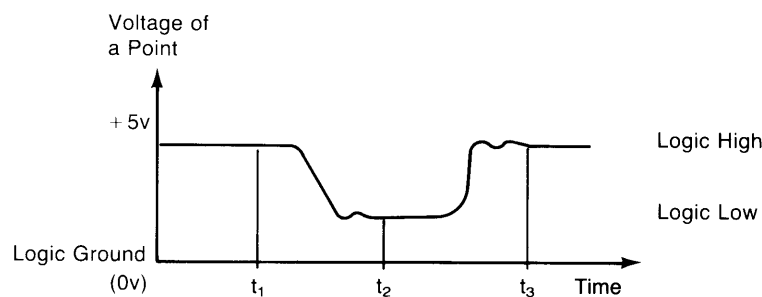
Voltage and Positive-True Logic

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 ($=2 \uparrow 16$) 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 ($=2 \uparrow 8$) 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 ($= 2 \uparrow 0$), a 1 in the 1st position has a value of 2 ($= 2 \uparrow 1$), and so forth. The number that the byte represents is then the total of all the individual bit's values.

Determining the Number Represented

$$
\begin{aligned}
0 * 2 \uparrow 0 &= \quad 0 \\
1 * 2 \uparrow 1 &= \quad 2 \\
1 * 2 \uparrow 2 &= \quad 4 \\
0 * 2 \uparrow 3 &= \quad 0 \\
1 * 2 \uparrow 4 &= \quad 16 \\
0 * 2 \uparrow 5 &= \quad 0 \\
0 * 2 \uparrow 6 &= \quad 0 \\
1 * 2 \uparrow 7 &= 128
\end{aligned}
$$

Number represented =

$2 + 4 + 16 + 128 = 150$

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

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

Printed Result

Number = 65

Representing Characters

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

[1] ASCII stands for "American Standard Code for Information Interchange". See the Useful Tables 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 ($=2 \uparrow 16$) unique integers can be represented.

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

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

sign bit ⎯ ... ⎿ $2 \uparrow 0$
$2 \uparrow 14$ ⎯ ⎿ $2 \uparrow 7$
$2 \uparrow 13$ ⎯
$2 \uparrow 8$ ⎯

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 computers. They are represented in a special binary mantissa-exponent notation within the computers for numerical calculations. During output and enter operations, they can also be represented with ASCII-decimal digits.

Internal Representation of Real Numbers

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

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

mantissa sign   exponent   mantissa

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

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 computer provides a means so that both types of representations can be used during I/O operations.

ASCII Representation of Real Numbers

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

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

---

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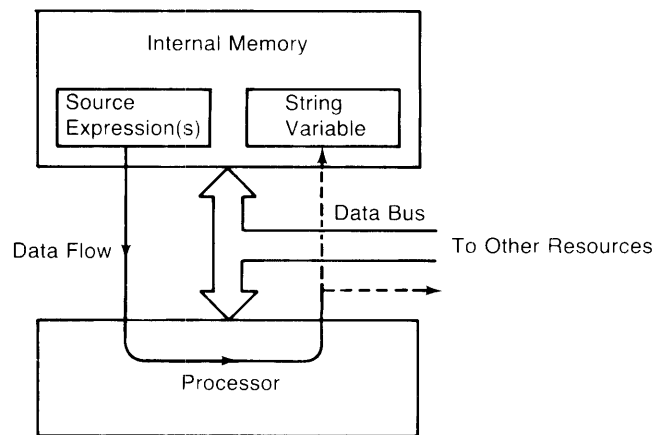
Data is Copied from Memory to a Resource During Output

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.
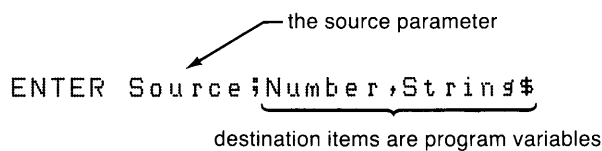
Copying Data to the Destination

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

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

Example Enter Statement

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

```
                              ┌─ the 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.
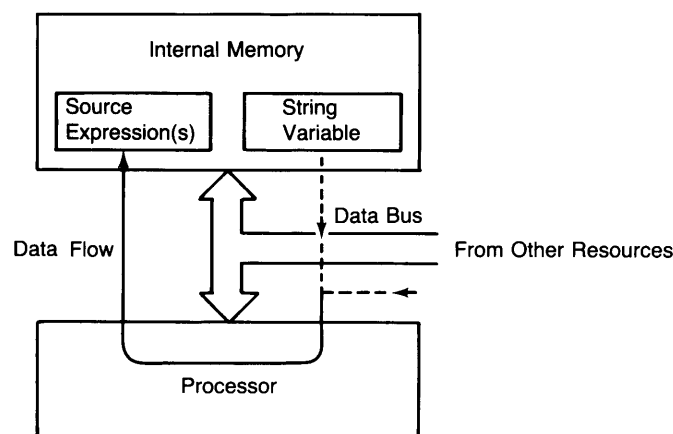
Data is Copied from a Resource to Memory During Enter

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.

Directing Data Flow	Chapter
	3

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
- Buffers

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 and in *BASIC Programming Techniques*. Buffers are described in Chapter 11.

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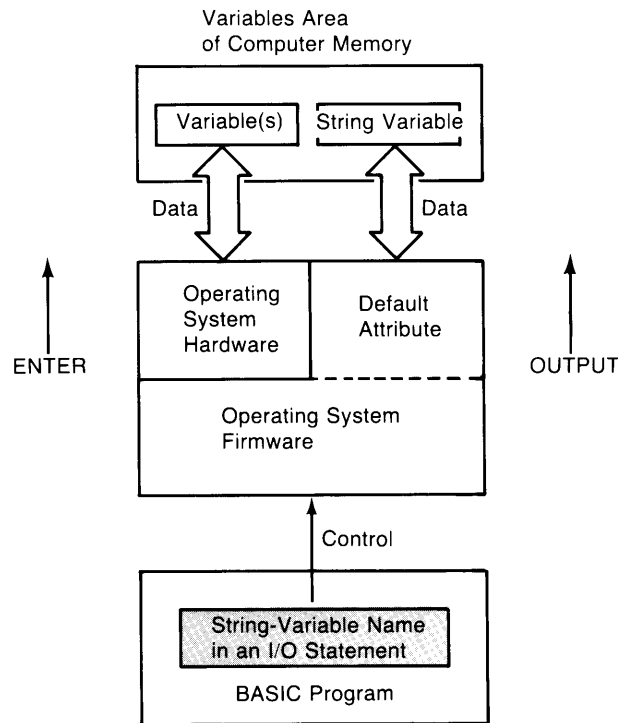
**Diagram of the Default I/O Path Used
for String-Variable I/O Operations**

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	Optional powerfail protection	5
Keyboard	2	Display (bit-mapped graphics)	6
Display (graphics)	3	Built-in HP-IB	7
Internal flexible disc drive	4		

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.

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

To access the device on:

interface select code 7 at primary address 01, use device selector 701
interface select code 10 at primary 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.
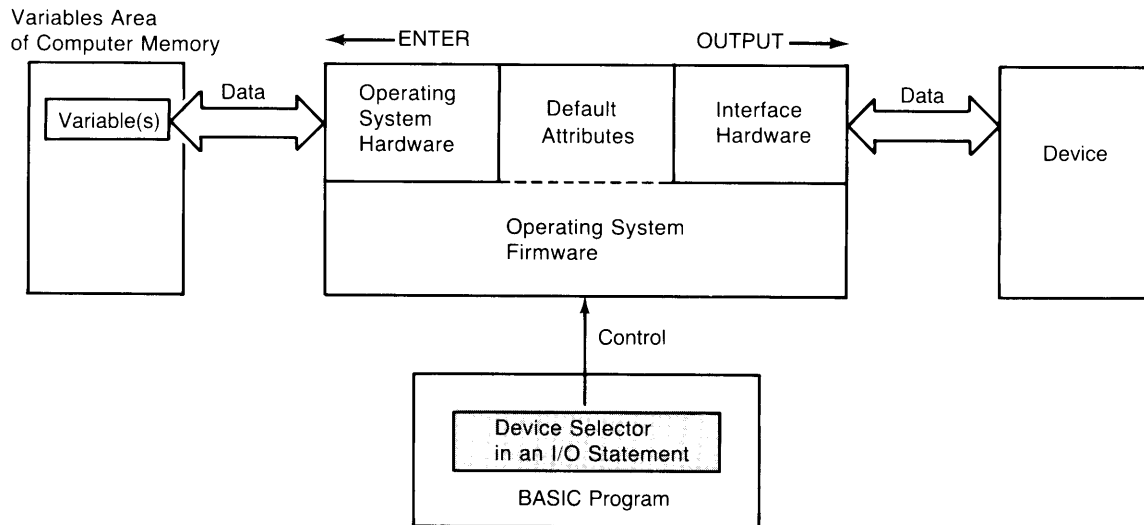
**Diagram of the Default I/O Path Used
when a Device Selector is Specified**

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

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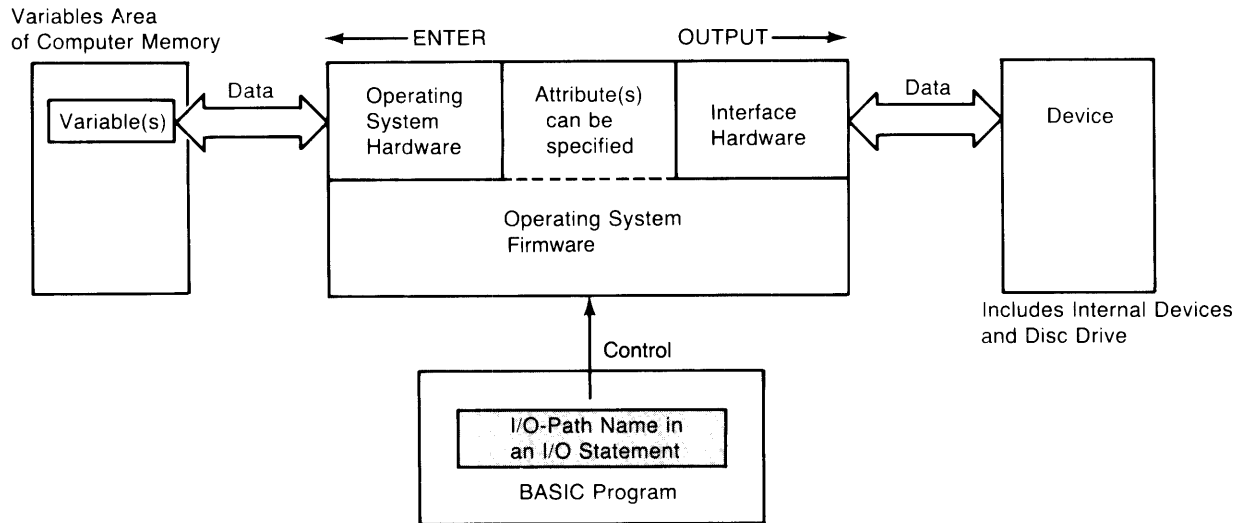
I/O Paths to Devices and Mass-Storage Files

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 See the *BASIC Language Reference* or *BASIC Programming Techniques* for further details.

Re-Assigning I/O Path Names

If an I/O path name already assigned to a resource is to be re-assigned to another resource, the preceding form of the ASSIGN statement is also used. The resultant action is that the validity flag is first set false, implicitly "closing" the I/O path name to the device[1]. A "new assignment" is then made just as if the first assignment never existed. Making this new assignment places information describing the specified device into the variable and sets the validity flag true. An example is shown below.

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

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

Closing I/O Path Names

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

Examples

```
ASSIGN @Printer TO *

ASSIGN @Serial_card TO *

ASSIGN @Gpio TO *
```

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

---

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

I/O Path Names in Subprograms

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

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

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

Assigning I/O Path Names Locally Within Subprograms

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

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

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

```
10  CALL Subprogram_x
11  OUTPUT @Log_device;"Main"  ◄──────────────────────  Insert into previous
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) ←───────────────────── Add formal parameter.
50  ASSIGN @Log TO 1  ! CRT.
60  OUTPUT @Log;"Subprogram"
70  SUBEND
```

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

Declaring I/O Path Names in Common

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

```
1    COM @Log_device ←──────────────────────────── Insert COM
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 "OUT-PUT 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 below.

Attribute Control

I/O paths have certain "attributes" which control how the system handles data sent through the I/O path. For example, the FORMAT attribute possessed by an I/O path determines which data representation will be used by the path during communications. If the path possesses the attribute of FORMAT ON, the ASCII data representation will be used. This is the default attribute automatically assigned by the computer when I/O path names are assigned to device selectors. If the I/O path possesses the attribute of FORMAT OFF, the internal data representation is used; this is the default format for BDAT files. Further details of these and additional attributes are discussed in Chapter 10.

The second additional factor that favors using I/O path names is that you can control which attribute(s) are to be assigned to the I/O path to devices (and also to the I/O paths to files and buffers). If device selectors are used, this control is not possible. Chapter 10 describes how to specify the attributes to be assigned to an I/O path and gives several useful techniques for using the available attributes.

Outputting Data	Chapter
	4

Introduction

The preceding chapter described how to identify a specific device as the destination of data in an OUTPUT statement. Even though a few example statements were shown, the details of how the data are sent were not discussed. This chapter describes the topic of outputting data to devices; outputting data to string variables, buffers, and mass storage files is described in Chapters 10 and 11 of this manual, in Chapter 7 of *BASIC Programming Techniques*, and in the *BASIC Language Reference*.

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

The OUTPUT and ASSIGN keywords are in mainframe BASIC. You may have to load some BIN files, depending on your system, to make all of the examples work.

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 15 (and INTEGERs can have up to 5) significant decimal digits of accuracy, not all of these digits are output with free-field OUTPUT statements. Instead, the following rules of the free-field convention are used when generating a number's ASCII representation.

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

Examples

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

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

Examples

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

Standard String Format

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. The end-of-line (EOL) sequence is discussed in the next section.

1st item	item terminator	2nd item	item terminator	...	last item	EOL sequence
	optional		optional			optional
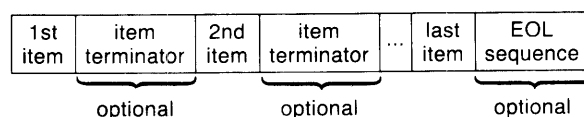
Using a **comma separator** after an item specifies that the **item terminator** (corresponding to the type of item) will be output after the last character of this item. A carriage-return, CHR$(13), and a line-feed, CHR$(10), terminate string items.

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

| I | t | e | m | CR | LF | – | 1 | 2 | 3 | 4 | EOL sequence |

The default EOL sequence is a CR/LF.

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",          OUTPUT Device;-1234,
```

| I | t | e | m | CR | LF |

| – | 1 | 2 | 3 | 4 | , |

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

```
OUTPUT 1;"Item1";"Item2"          OUTPUT 1;-12;-34
```

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

| – | 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	EOL sequence
	1	1	,		1	2	,		1	3	,		2	1	,		2	2	,		2	3	,
	1	1		1	2		1	3		2	1		2	2		2	3						

D	O	N	E	EOL sequence

Item separators cause similar action for string arrays.

```
100     OPTION BASE 1
110     DIM Array$(2,3)[2]
120     FOR Row=1 TO 2
130         FOR Column=1 TO 3
140             Array$(Row,Column)=VAL$(Row*10+Column)
150         NEXT Column
160     NEXT Row
170     !
180     OUTPUT 1;Array$(*)   ! 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	EOL sequence
1	1	CR	LF	1	2	CR	LF	1	3	CR	LF	2	1	CR	LF	2	2	CR	LF	2	3	EOL sequence
1	1	1	2	1	3	2	1	2	2	2	3											
D	O	N	E	EOL sequence																		

A pad byte may be sent following the last character of the EOL sequence when using an I/O path that possesses the WORD attribute. See Chapter 10 for further information.

Changing the EOL Sequence

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

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

```
ASSIGN @Device TO 12;EOL "LFLFCR"
```

If you cannot interpret the " >>< ", refer to the table for ASCII code and note that (CTRL)(J) is equivalent to a line feed, (CTRL)(N) to a carriage return.

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

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

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

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

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

```
ASSIGN @Device;EOL "CRLF" DELAY 0.1
```

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

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

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

Using END in Free-Field OUTPUT

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

Examples

```
ASSIGN @Gpio TO 12


OUTPUT @Gpio;-10,END
```

–	1	0	,

Item terminator, but no EOL sequence, is sent.
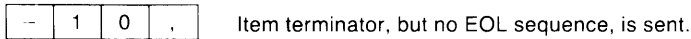
```
OUTPUT @Gpio;-10;END
OUTPUT @Gpio;-10 END
```

–	1	0

Neither item terminator nor EOL sequence is sent.

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

A	B	CR	LF

Item terminator, but no EOL sequence, is sent.
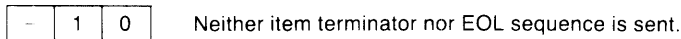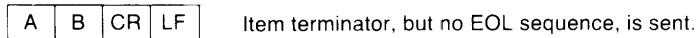
```
OUTPUT @Gpio;"AB";END
OUTPUT @Gpio;"AB" END
```

A	B

Neither item terminator nor EOL sequence is sent.

```
OUTPUT @Gpio
```

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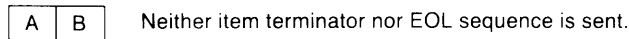
The EOL sequence is sent.
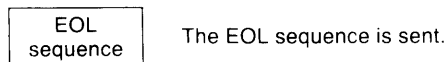
```
OUTPUT @Gpio;END            No EOL sequence is sent.
OUTPUT @Gpio;"" END
```

The END keyword has additional significance when the destination is a mass storage file. See Chapter 7 of *BASIC Programming Techniques* for further details.

Additional Definition

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

END with HP-IB Interfaces

With HP-IB interfaces, END has the additional function of sending the End-or-Identify signal (EOI) with the last data byte of the last source item; however, **if no data are sent from the last source item, EOI is not sent.** For further description of the EOI signal, see Chapter 12.

Examples

```
ASSIGN @Device TO 701
```

```
OUTPUT @Device;-10,END
```
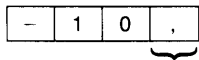
```
┌───┬───┬───┬───┐
│ - │ 1 │ 0 │ , │
└───┴───┴───┴───┘
            └─┬─┘
EOI sent with the last character
   (numeric item terminator).
```

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

```
┌───┬───┐
│ A │ B │
└───┴───┘
    └─┬─┘
EOI sent with the last character of the item.
```

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

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

END with the Data Communications Interface

With Data Communication interfaces, END has the additional function of sending an end-of-data indication to the interface. See Chapter 13 for further details.

Outputs that Use Images

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

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

The OUTPUT USING Statement

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

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

2.  100   Image_str$="6A,SDDD.DDD,3X"
    110   OUTPUT 1 USING Image_str$;" 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.
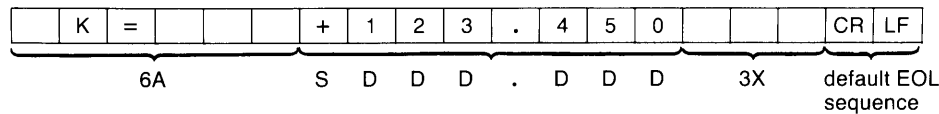
Step 1.   The computer evaluates the first image in the list. Generally, each group of specifiers separated by commas is an "image"; the commas tell the computer that the image is complete and that it can be "processed". In general, each group of specifiers is processed before going on to the next group. In this case, 6 alphanumeric characters taken from the first item in the source list are to be output.

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

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

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

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

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

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

Image Definitions During Outputs

This section describes the definitions of each of the image specifiers when referenced by OUTPUT statements. The specifiers have been categorized by data type. It is suggested that you scan through the description of each specifier and 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.

Sign, Digit, Radix and Exponent Specifiers

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

Numeric Examples

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

| − | 1 | 2 | 4 | EOL sequence |

`OUTPUT @Device USING "2D";-1.2`

| − | 1 | EOL sequence |

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

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

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

| 0 | . | 4 | EOL sequence |

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

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

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

| 2 | 0 | . | 0 | 0 | E | − | 0 | 5 | EOL sequence |

`OUTPUT @Device USING "K";12.400`

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

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

| − | 1 | 2 | . | 4 | 5 | EOL sequence |

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

| | | 2 | . | 0 | 9 | EOL sequence |

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

| + | 2 | . | 4 | EOL sequence |

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

| + | 0 | . | 4 | 9 | EOL sequence |

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

| − | 2 | 3 | . | 5 | 0 | E | − | 0 | 1 | EOL sequence |

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

*	2	.	6	EOL sequence

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

3	.	1	4	EOL sequence

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

3	.	1	4	1	6	EOL sequence

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 H, -H	Specifies that the string is to be output in "compact" format, similar to the standard string format; however, no item terminators are output as with the standard string format.

String Examples

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

C	h	a	r	a	c	t	e	EOL sequence

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

A	B	L	i	t	e	r	a	l	EOL sequence

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

			H	e	l	l	o			EOL sequence

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

			H	e	EOL sequence

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 corresponding numeric expression is evaluated, rounded to an integer, and interpreted MOD 256. If it is less than $-32\,768$, then CHR$(0) is output; if it is greater than 32 767, then CHR$(255) is output.
W	Specifies that one word (16 bits) of data is to be sent as a 16-bit, two's-complement integer. The corresponding numeric item is rounded to an integer. If it is greater than 32 767, then 32 767 is sent; if it is less than $-32\,768$, then $-32\,768$ is sent. If either an I/O path name with the BYTE attribute (see Chapter 10) or a device selector is used to access an 8-bit interface, two bytes will be output; the first byte output is most significant. If an I/O path name with the BYTE attribute is used to access a 16-bit interface, the BYTE attribute is overridden and one word is output in a single operation. If an I/O path name with the WORD attribute is used to access a 16-bit interface, a pad byte, CHR$(0), is output whenever necessary to achieve alignment on a word boundary. If the destination is a BDAT file, string variable, or buffer, the WORD attribute is ignored and all data are sent as bytes; however, pad byte(s) will also be output when necessary to achieve alignment on a word boundary. The pad byte may be changed by using the CONVERT attribute, if desired (see Chapter 10).
Y	Like W, except that no pad bytes are output to achieve word alignment. If an I/O path with the BYTE attribute is used to access a 16-bit interface, the attribute is not overridden (as with the W specifier).

Binary Examples

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

A	B	C	EOL sequence

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

CR

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

A	B	EOL sequence

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

```
OUTPUT @Device USING "K,W";"Odd",256*65+66
```
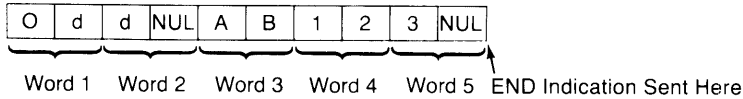
O	d	d	NUL	A	B	1	2	3	NUL

Word 1   Word 2   Word 3   Word 4   Word 5   END Indication Sent Here

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

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

O	d	d	A	B	CR	LF	NUL

Word 1   Word 2   Word 3   Word 4
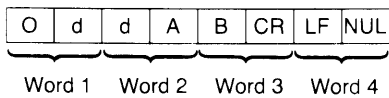
Special-Character Images

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	EOL sequence

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

←(50 spaces)→	EOL sequence

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

FF	CR	LF	EOL sequence

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

CR	LF	EOL sequence
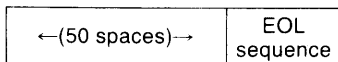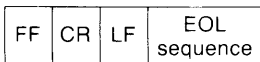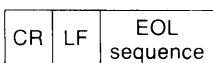
Termination Images

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

Termination Examples

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

D	a	t	a	EOL sequence	EOL sequence

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

D	a	t	a

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

FF

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

D	a	t	a	CR

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

EOL sequence	D	a	t	a	LF

Additional Image Features

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

Repeat Factors

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

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

Examples

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

0	3	2	8	.	0	3	0	EOL sequence

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

D	a	t	a		b	EOL sequence

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

					D	a	EOL sequence

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

EOL sequence	EOL sequence	D	a	t	a	EOL sequence

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

T	h	e		E	n	d		FF	FF	EOL sequence

OUTPUT @Device USING "2/"

CR	LF	CR	LF	EOL sequence

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	EOL sequence

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

END with OUTPUTs that Use Images

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

Examples

```
Device=12
```

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

A	B	C	EOL sequence

The EOL sequence is not suppressed.

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

EOL sequence	CR	LF	L	i	t	e	r	a	l		FF	EOL sequence

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

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

EOL sequence	CR	LF	L	i	t	e	r	a	l		FF

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

Additional Definition

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

END with HP-IB Interfaces

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

Examples

```
ASSIGN @Device TO 701
```

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

```
| D | a | t | a | EOL
              sequence |
```
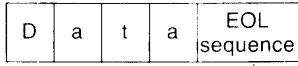
EOI sent with last character
of the EOL sequence.

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

```
| D | a | t | a |
```
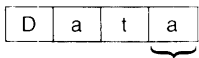
EOI sent with this character.

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

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

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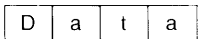
The EOI was not sent in either case, since no data were sent from the last source item **and** the EOL sequence was suppressed.

END with Data Communications Interfaces

With Data Communications interfaces, END has the additional definition of sending an end-of-data indication to the interface in the same instances in which EOI would be sent on HP-IB interfaces. See Chapter 13 for further details.

	Chapter
Entering Data	**5**

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

When using an I/O path that possesses the WORD attribute, an additional byte may be entered (but ignored). See Chapter 10 for further information.

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.

Mantissa sign	Mantissa digit(s)	E	Exponent sign	Exponent digit(s)	Terminator (character or END indication)

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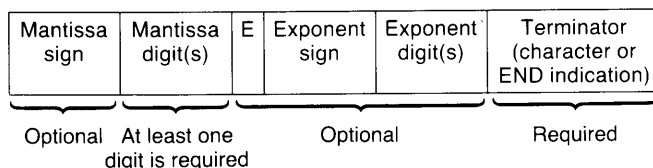
Numeric characters include decimal digits "0" through "9" and the characters ".", "+", "−", "E", and "e". These last five characters must occur in meaningful positions in the data stream to be considered numeric characters; if any of them occurs in a position in which it cannot be considered part of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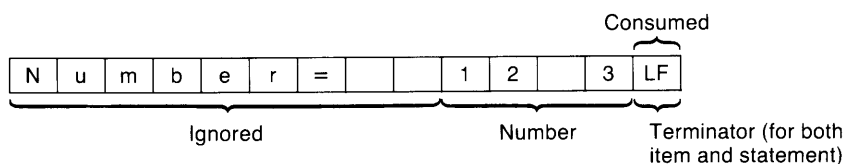
Consumed

N	u	m	b	e	r	=			1	2		3	LF

Ignored	Number	Terminator (for both item and statement)

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

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

Example

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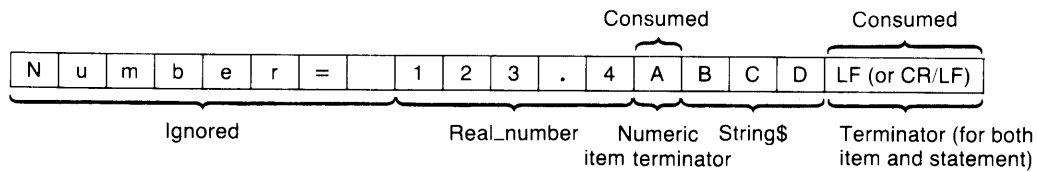
Consumed   Consumed

| N | u | m | b | e | r | = | | 1 | 2 | 3 | . | 4 | A | B | C | D | LF (or CR/LF) |

Ignored   Real_number   Numeric String$ Terminator (for both
                                          item terminator   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
```
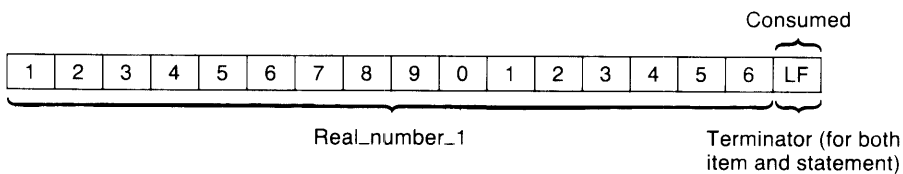
Consumed

| 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
```
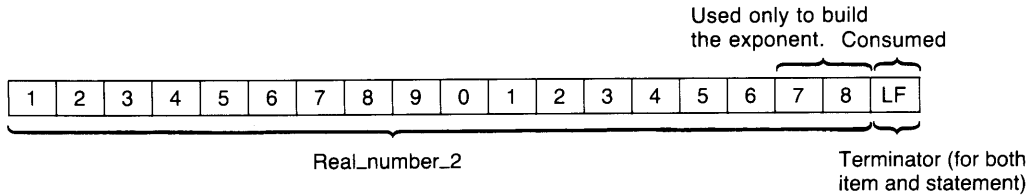
Used only to build
the exponent. Consumed

| 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
```
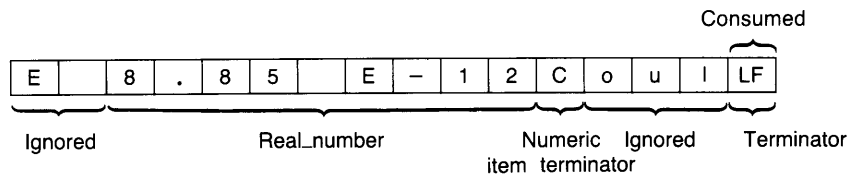
Consumed

| E | | 8 | . | 8 | 5 | | E | – | 1 | 2 | C | o | u | l | LF |

Ignored — Real_number — Numeric item terminator — Ignored — Terminator

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
```

Consumed

| 1 | 2 | 3 | . | 4 | E | + | 3 | 0 | 7 | LF |

Evaluates to 1.234 E + 309.

The resultant value cannot   Terminator (for both items
be stored in Real_number.   and statement)
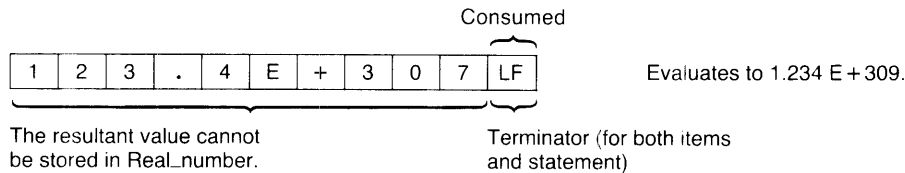
The data is entered but evaluates to a number outside the range of REAL numbers. Consequently, error 19 is reported, and the variable **Real_number retains its former value**.

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 END indication (such as EOI signal with a character). The topic of terminating free-field ENTER statements is described later in this chapter in the section of the same name.

Entering String Data

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

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

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

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

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

Example

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

```
                                        Consumed
                                       ⌐‾⌐
  ┌───┬───┬───┬───┬───┬───┬───┬───┬───┬───┐
  │ A │ B │ C │ D │ E │ F │ G │ H │CR │LF │
  └───┴───┴───┴───┴───┴───┴───┴───┴───┴───┘
  └─────────┬─────────┘ └───┬───┘ └───┬───┘
        Five_char$      Ignored   Terminator (for both
                                  item and statement)
```
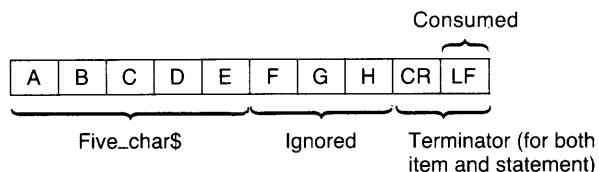
The variable Five_char$ only receives the characters "ABCDE", but the characters "FGH" are entered (and ignored) in search of the terminating carriage-return/line-feed (or line-feed).

Example

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

```
                 Consumed                                        Consumed
                ⌐‾⌐                                             ⌐‾⌐
┌───┬───┬───┬───┬───┬───┬───┬───┐              ┌───┬───┬───┬───┬───┬───┬───┬───┬───┐
│ A │ B │ C │ D │ E │ F │ G │LF │      or      │ A │ B │ C │ D │ E │ F │ G │CR │LF │
└───┴───┴───┴───┴───┴───┴───┴───┘              └───┴───┴───┴───┴───┴───┴───┴───┴───┘
└───────────┬───────────┘ └─┬─┘                └───────────┬───────────┘ └───┬───┘
      Ten_char$         Terminator (for              Ten_char$         Terminator (for both
                       both item and statement)                        item and statement)
```
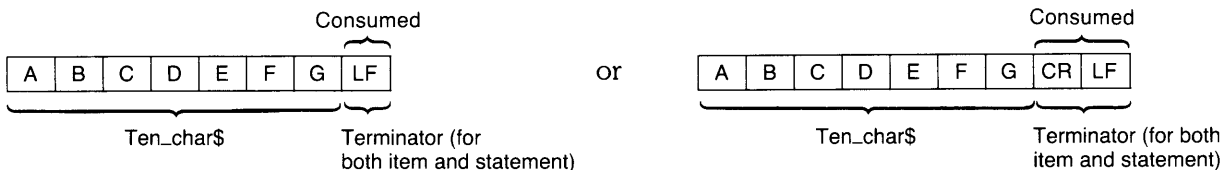
The result of entering the preceding data with the given ENTER statement is that Ten_char$ receives the characters "ABCDEFG" and the terminating LF (or CR/LF) is lost.

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 an END indication is received while entering data into the last item, the statement is properly terminated. Examples of END indications are encountering the last character of a string variable while entering data from the variable, receiving EOI with a character, and receiving a control block while entering data through the Data Communications interface.

3 If one of the preceding **statement-termination** conditions has **not** occurred **but** entry into the **last item** has been terminated, up to 256 **additional** characters are entered in search of a termination condition. If one is not found, an error occurs.

One case in which this termination condition may not be obvious can occur while entering string data. If the last variable in the destination list is a string **and** the dimensioned length of the string has been reached **before** a termination condition occurs, additional characters are enterd (but ignored) until a termination condition occurs. 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, an end-of-data, or a character accompanied by EOI. Unless a 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$
```

| A | B | C | D | E | F | or | A | B | C | D | E | F | LF | or | A | B | C | D | E | F | CR | LF |

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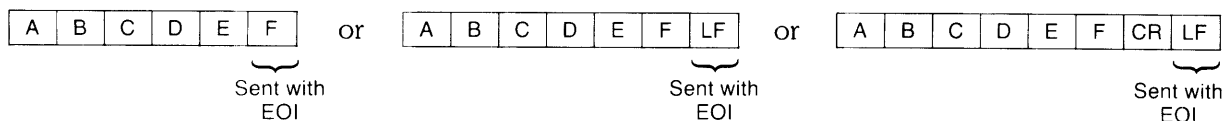
The result of entering the preceding data with the given ENTER statement is that String$ receives the characters "ABCDEF". The EOI signal being received with either the last character or with the terminator character properly terminates the ENTER statement. If the character accompanied by EOI is a string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

Example

ENTER @Device;Number

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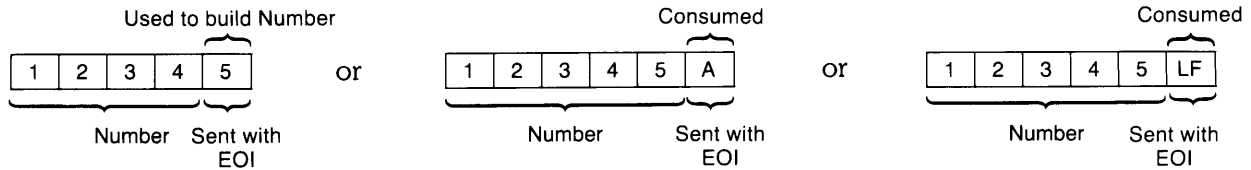
The result of entering any of the above data streams with the given ENTER statement is that Number receives the value 12345. If the EOI signal accompanies a numeric character, it is entered and used to build the number; if the EOI is received with a numeric terminator, the terminator is lost as usual.

Example

ENTER @Device;Number,String$

An error is reported
(Error 153 Insufficient data for ENTER).
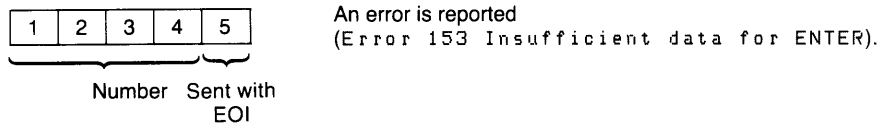
The result of entering the preceding data with the given statement is that an **error is reported** when the character "5" accompanied by EOI is received. However, Number receives the value 12345, but String$ retains its previous value. An error is reported because **all** variables in the destination 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 — "Temp. =   " (first group)

Degrees — "+98.3"

Units$ — "F"

Ignored — "ahrenheit"

Assume EOI is sent with this character
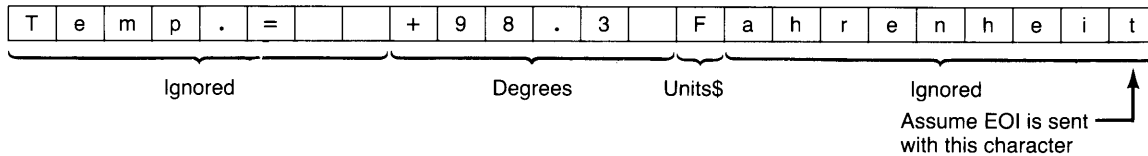
Given the above conditions, let's look at how the computer executes the following ENTER statement that uses the specified IMAGE statement.

```
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	Specifies that one byte is to be entered and interpreted as a numeric character. If the character is non-numeric (including leading spaces and item terminators), it will still "consume" one digit of the image item.
Z, *	Same action as D. Keep in mind that Z and * can only appear to the left of the radix indicator (decimal point or R) in a numeric image item.
S, M	Same action as D in that one byte is to be entered and interpreted as a numeric character. At least one digit specifier must follow either of these specifiers in an image item.
.	Same action as D in that one byte is to be entered and interpreted as a numeric character. At least one digit must accompany this specifier in the image item.
R	Same action as D in that one byte is to be entered and interpreted as a numeric character; however, when R is used in a numeric image, it directs the number builder to use the comma as a radix indicator and the period as a terminator for the numeric item. At least one digit specifier must accompany this specifier in the image item.
E	Equivalent to 4D, if preceded by at least one digit specifier (Z, *, or D) in the image item. The following specifiers must also be preceded by at least one digit specifier.
ESZ	Equivalent to 3D.
ESZZ	Equivalent to 4D.
ESZZZ	Equivalent to 5D.
K, −K	Specifies that a variable number of characters are to be entered and interpreted according to the rules of the number builder.
H, −H	Like K, except that a comma is used as the radix indicator, and a period is used as a terminator for the numeric item.

Examples of Numeric Images

```
ENTER @Device USING "SDD,D";Number
ENTER @Device USING "3D,D";Number
ENTER @Device USING "5D";Number
ENTER @Device USING "DESZZ";Number
ENTER @Device USING "**,DD";Number
```

These five images are equivalent.

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

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

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

String Images

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

String Specifiers

Image Specifier	Meaning
A	Specifies that one byte is to be entered and interpreted as a string character. Any terminators are entered into the string when this specifier is used.
K, H	Specifies that 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 is received (CR/LF, LF, or an END indication).
−K, −H	Like K, except that line-feeds do not terminate entry into the string; instead, they are treated as string characters and placed in the variable. Receiving an END indication (for instance, receiving EOI with a character, encountering an end-of-data, or reaching the string variable's dimensioned length) terminates the image 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. Note that it may be necessary to skip characters (with specifiers such as "X" or "/") when ENTERing data which has been sent by including these specifiers in an OUTPUT statement. Even greater care must be given to cases in which pad bytes may be sent; see "The BYTE and WORD Attributes" in Chapter 10 for further explanation.

Examples of String Images

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

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

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

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

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

Ignoring Characters

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

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 and interpreted as an integer in the range of 0 through 255.
W	Specifies that one 16-bit word is to be entered and interpreted as a 16-bit, two's-complement integer. If either an I/O path name with the BYTE attribute (see Chapter 10) or a device selector is used to access an 8-bit interface, two bytes will be entered; the first byte entered is most significant. If an I/O path name with the BYTE attribute is used to access a 16-bit interface, the BYTE attribute is overridden and one word is entered in a single operation. If an I/O path name with the WORD attribute is used to access a 16-bit interface, one byte is entered and ignored when necessary to achieve alignment on a word boundary. If the source is a file, string variable, or buffer, the WORD attribute is ignored and all data are entered as bytes; however, one byte may still be entered and ignored when necessary to achieve alignment on a word boundary.
Y	Like W, except that pad bytes are never entered to achieve word alignment. If an I/O path name with the BYTE attribute is used to access a 16-bit interface, the BYTE attribute is not overridden (as with the W specifier).

Examples of Binary Images

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

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

Assume that @Device possesses the WORD attribute.

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

@Device may possess either BYTE or WORD attribute.

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

Terminating Enters that Use Images

This section describes the **default statement-termination conditions** for **enters that use images** (for devices). The effects of numeric-item and string-item terminators and the 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. An END indication (such as the EOI signal or end-of-data) is received **with** the byte that satisfies the last **image item** or **within 256 bytes after** the byte that satisfied the last image item.

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.

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

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 terminator or statement terminator	Item terminator or statement terminator	Immediate statement terminator
Statement terminator required?	Yes	Yes	No	No
Early termination allowed?	No	No	No	Yes

Statement-Termination Modifiers

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

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.

Image Specifier	Meaning
+	Specifies that an END indication is required to terminate the ENTER statement. Line-feeds are ignored as statement terminators; however, they will still terminate items (unless a – K or – H image is used for strings).
–	Specifies that a line-feed is required to terminate the statement. EOI is ignored, and other END indications (such as EOF or end-of-data) cause an error if encountered before the line-feed.

Examples of Modifying Termination Conditions

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

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

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

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

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

Additional Image Features

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

Registers	Chapter
	6

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 computer 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  !  Parm 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.
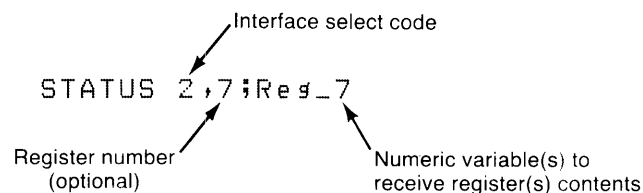
Example

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
```
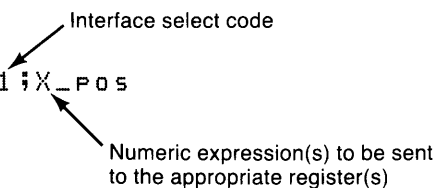
The CONTROL Statement

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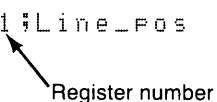
Register 0 of interface select code 1 is modified with the following statement. This register determines the "X" screen coordinate at which subsequent characters output to the CRT display will appear.

```
                          Interface select code
100   CONTROL 1;X_pos
                          Numeric expression(s) to be sent
                          to the appropriate register(s)
```

Example

Register 1 of interface select code 1 is modified with the following statement. This register's contents determine the "Y" coordinate at which subsequent characters' output to the screen appear.

```
100   CONTROL 1,1;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.

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

```
100   ! Example of determining select code
110   ! to which an I/O path name is assigned.
120   !
130 Show_sc: IMAGE "'@Io_path' assigned to ",K,"; Select code = ",D,L
140        !
150   ASSIGN @Io_path TO 701   ! Device selector.
160   Device_selector=FNSc(@Io_path)
170   OUTPUT 1 USING Show_sc;"device 701",Device_selector
180   !
190   ASSIGN @Io_path TO "Data1" ! ASCII file.
200   Device_selector=FNSc(@Io_path)
210   OUTPUT 1 USING Show_sc;"ASCII file",Device_selector
220   !
230   ASSIGN @Io_path TO "Chap1" ! BDAT file.
240   Device_selector=FNSc(@Io_path)
250   OUTPUT 1 USING Show_sc;"BDAT file",Device_selector
260   !
270   ASSIGN @Io_path TO BUFFER [1024] ! Buffer.
280   Device_selector=FNSc(@Io_path)
290   OUTPUT 1 USING Show_sc;"BUFFER",Device_selector
300   !
310   END
320   !
330   DEF FNSc(@Io_path) ! ***************************************
340      ! Read I/O path register 0.
350      STATUS @Io_path;Resource_code
360      SELECT Resource_code
370      CASE 0   ! Not assigned.
380         RETURN 32  ! Return a select code out of range.
390      CASE 1   ! Assigned to a device selector.
400         STATUS @Io_path,1;Select_code
410         RETURN Select_code
420      CASE 2   ! Assigned to a file specifier.
430         STATUS @Io_path,2;Device_selector
440         WHILE Device_selector>100
450            Device_selector=Device_selector/100   ! Remove addresses.
460         END WHILE
470         RETURN Device_selector
480      CASE 3 ! Assigned to a buffer.
490         RETURN 0   ! No error, but cannot determine source
500                    ! or destination of transfer to/from buffer.
510      END SELECT
520      !
530      FNEND ! ***************************************
```

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

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

'@Io_path' assigned to ASCII file; Select code = 4

'@Io_path' assigned to BDAT file; Select code = 4

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

The user-defined function called FNSc interrogates I/O path registers to find the select code. If the I/O path name is currently not assigned, the function returns an arbitrary value of 32 (an invalid value of select code). Since STATUS Register 2 of I/O path names assigned to files contains the entire device selector, which may include addressing information, the function must remove any addressing information to extract the select code.

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

With AP2.0, this function is a feature of the BASIC language system. The following statements show examples of using this function.

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

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

Interface Registers

I/O Path Status and Control Registers

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
	3	I/O path name assigned to a buffer

I/O Path Names Assigned to a Device:

Status Register 1	Interface select code
Status Register 2	Number of devices
Status Register 3	1st device selector

If assigned to more than one device, the other device selectors 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 (in bytes)
Status Register 5	Current record
Control Register 5	Set current record
Status Register 6	Current byte within record
Control Register 6	Set current byte within record
Status Register 7	EOF record
Control Register 7	Set EOF record
Status Register 8	Byte within EOF record
Control Register 8	Set byte within EOF record

I/O Path Names Assigned to a Buffer:

Status Register 1	Buffer type (1 = named, 2 = unnamed)
Status Register 2	Buffer size in bytes
Status Register 3	Current fill pointer
Control Register 3	Set fill pointer
Status Register 4	Current number of bytes in buffer
Control Register 4	Set number of bytes
Status Register 5	Current empty pointer
Control Register 5	Set empty pointer
Status Register 6	Interface select code of inbound TRANSFER
Status Register 7	Interface select code of outbound TRANSFER
Status Register 8	If non-zero, inbound TRANSFER is continuous
Control Register 8	Cancel continuous mode inbound TRANSFER if zero
Status Register 9	If non-zero, outbound TRANSFER is continuous
Control Register 9	Cancel continuous mode outbound TRANSFER if zero
Status Register 10	Termination status for inbound TRANSFER

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	TRANSFER Active	TRANSFER Aborted	TRANSFER Error	Device Termination	Byte Count	Record Count	Match Character
Value = 0	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Status Register 11   Termination status for outbound TRANSFER

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	TRANSFER Active	TRANSFER Aborted	TRANSFER Error	Device Termination	Byte Count	Record Count	0
Value = 0	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 0

Status Register 12	Total number of bytes transferred by last inbound TRANSFER
Status Register 13	Total number of bytes transferred by last outbound TRANSFER

	Chapter
Interface Events	7

Introduction

The 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 *BASIC Programming Techniques* and 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 and KBD events are described in Chapter 9, "The Internal Keyboard Interface"; the END, ERROR, and KEY events are described in *BASIC Programming Techniques* and 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.

KBD — occurs when an enabled key is pressed.

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_1" and "Key_2" are the service routines for the events of **k1** and **k2** 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. The HP 46020 keyboard has no softkeys that correspond with **k0** or **k9**. The softkeys which correspond with **k1** through **k8** are labled **f1** through **f8**. More on this in the keyboard chapter.

```
100 ON KEY 1,3 CALL Key_1   ! Set up events and
110 ON KEY 2,4 CALL Key_2   ! 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_1
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_2
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 **k2** is pressed **after k1 but while** the Key_1 routine is being executed, execution of Key_1 is **temporarily interrupted** and the Key_2 routine is executed. When Key_2 is finished, execution of Key_1 is resumed at the point where it was temporarily interrupted. This occurs because **k2** was assigned a **higher software priority** than k1.
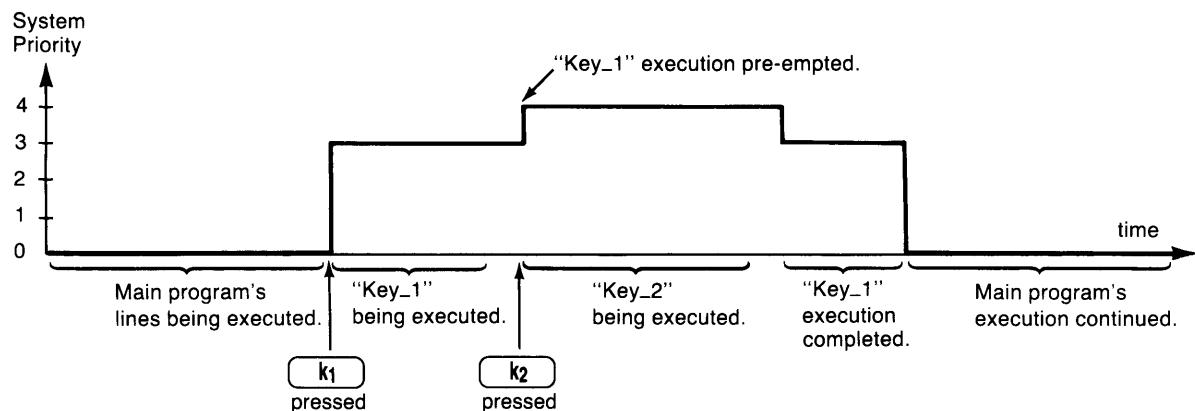
Events with Higher Software Priority Take Precedence

On the other hand, if **k1** is pressed **while k2** is being serviced, the computer finishes executing Key_2 **before** executing Key_1. The event of pressing **k1** 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.
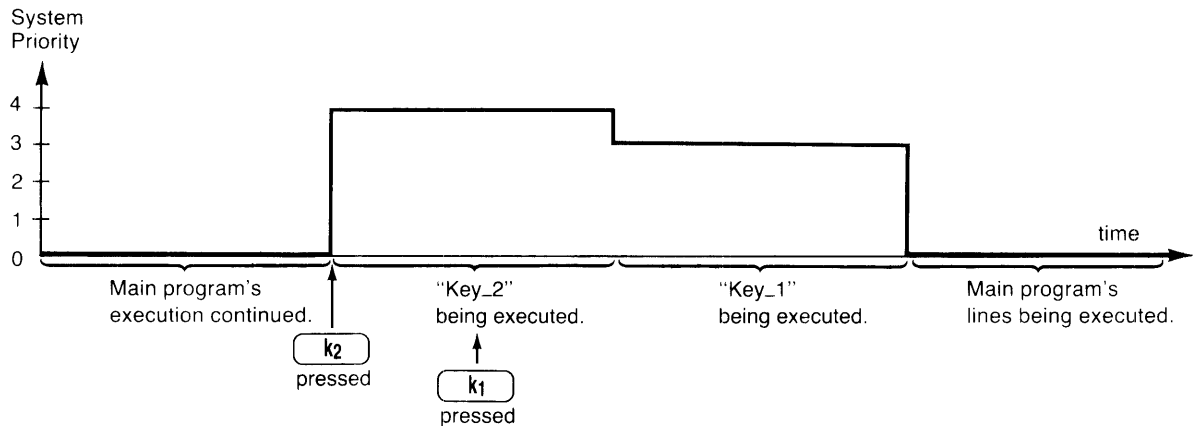
An Event with Lower Software Priority Must Wait

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 ocurrences 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 . . . . . . Effective software priority of END, ERROR, and TIME-OUT 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 **k1** was pressed, the system priority became 3. When **k2** was subsequently pressed, the system first logged the event and then checked its priority against the current system priority. Since **k2** had been assigned a priority of 4, it pre-empted **k1**'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 sections further describe logging and servicing events.

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

If **k1** was pressed **again** while its own service routine was being executed, execution of the first service routine was finished before the service routine was executed again. Thus, if an event occurs that has the **same** software priority as the system priority, its service routine will **not** interrupt the current routine. The service routine will **only** be executed if the event's software priority becomes the highest priority of any event which has been logged (i.e., **after all** other events of **higher** software priority have been serviced).

Changing System Priority

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

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

```
100     GINIT   ! Use default plotter is CRT.
110     GRAPHICS ON
120     VIEWPORT 0,131,30,100
130     WINDOW 0,2000,0,7
140     !
150     ON KEY 1 LABEL "Prior.1",1 GOSUB Key_1
160     ON KEY 2 LABEL "Prior.2",2 GOSUB Key_2
170     ON KEY 3 LABEL "Prior.2",3 GOSUB Key_3
180     !
190     Sys_prior$="SYSTEM PRIORITY" ! Define string for SYSTEM$.
200     !
210 Main_program: !
220     DISP "Quiescent system priority level = 0."
230     X=X+1
240     Sys_prior=VAL(SYSTEM$(Sys_prior$))
250     GOSUB Plot_priority
260     GOTO Main_program
270     !
280 Key_1:   FOR Iota=1 TO 100
290              DISP "Key 1; priority 1."
300              X=X+1
310              Sys_prior=VAL(SYSTEM$(Sys_prior$))
320              GOSUB Plot_priority
330          NEXT Iota
340          RETURN
350          !
360 Key_2:   FOR Twinkle=1 TO 100
370              DISP "Key 2; priority 2."
380              X=X+1
390              Sys_prior=VAL(SYSTEM$(Sys_prior$))
400              GOSUB Plot_priority
410          NEXT Twinkle
420          !
430          ! Critical routine raise system priority.
440          SYSTEM PRIORITY 3
450          FOR Split_second=1 TO 100
460              DISP "Subroutine set system priority to 3."
470              X=X+1
480              Sys_prior=VAL(SYSTEM$(Sys_prior$))
490              GOSUB Plot_priority
500          NEXT Split_second
510          !
520          ! System priority lowered when finished.
530          SYSTEM PRIORITY 0
540          RETURN
550          !
560 Key_3:   FOR Jiffy=1 TO 100
570              DISP "Key 3; priority 3."
```

```
580                X=X+1
590                Sys_Prior=VAL(SYSTEM$(Sys_Prior$))
600                GOSUB Plot_Priority
610          NEXT Jiffy
620          RETURN
630             !
640 Plot_Priority:  !
650                IF X>2000 THEN ! Draw new plot,
660                   GCLEAR
670                   MOVE 0,0
680                   X=0
690                END IF
700                PLOT X,Sys_Prior
710                RETURN
720                !
730                !
740     END
```

The subroutine called Key_2 raised the system priority from its current level, 2, to level 3 during the time that the second FOR-NEXT loop was being executed. During this time, pressing (**k3**) will not interrupt the routine, since a priority of 4 or greater is required to interrupt the Key_2 routine.

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

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

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

Hardware Priority

There is a second event priority, hardware priority, that also influences the order in which the computer responds to events. Hardware priority determines the order in which events are **logged** by the system, 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.** (This statement requires the IO BIN).

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 Model 226. 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 Interfaces

Hardware Priority	Interface(s) and Event(s) at This Priority
0	(Quiescent level; no interface is currently interrupting)
1	Internal Keyboard (KBD, KEY and KNOB events)
2	Internal Disc Drive
3	Internal HP-IB (INTR events)
3 – 6	Optional Interface Cards (INTR 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.

Pending Events

The ON INTR, OFF INTR, ENABLE INTR, and DISABLE INTR keywords let you set up means, within software, to service pending events. In particular, it is possible to accommodate interface cards, such as the HP 98630 Breadboard card, for which no driver is installed. You can allow interrupts to be detected from BASIC.

On INTR and OFF INTR statements may be executed for any I/O card plugged into the computer. If the I/O card does not have a driver loaded, all other I/O statements (CONTROL, STATUS, ENABLE INTR, OUTPUT, ASSIGN, etc.) will generate an "I/O interface not present" error 163. Before an interrupt can be generated by the driver-less I/O card, you must emulate the ENABLE INTR statement using WRITEIO. For example, if an HP98630 Breadboard card is at select code 17, the statements:

```
ON INTR 17 CALL My_card_fsr
WRITEIO 17,Card_mask_reg;Card_mask   ! Set the mask
WRITEIO 17,3;128        !  Enable the card to interrupt
```

Set up the card to call My–card–isr when the Breadboard card interrupts. The two WRITEIO statements perform the function of ENABLE INTR.

When the Breadboard card interrupts, BASIC clears bit 7 of register 3 on the Breadboard card (the interrupt enable bit) and logs the interrupt so that the user's BASIC routine will be called at the next end of line. No other actions are taken during the hardware ISR.

To perform a DISABLE INTR function, execute

```
WRITEIO 17,3;0          !  Disable the card
```

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

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 1,5 CALL Serv_K1     ! 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.
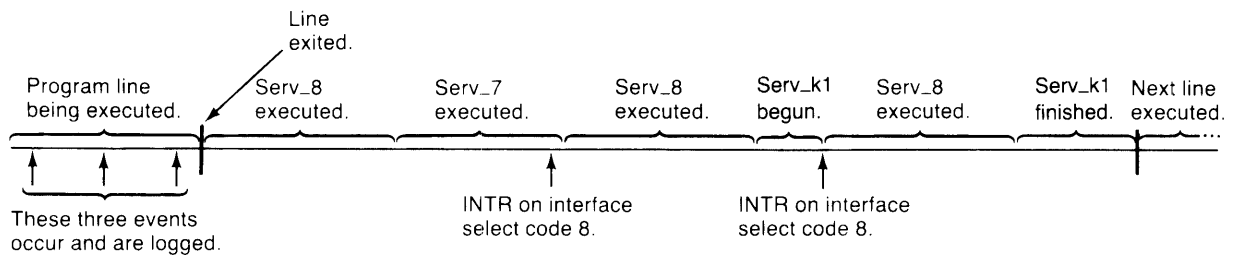
Software priority's **main function** is to pre-empt events of lower software priority so that more "urgent" events can be serviced quickly. Decreasing the system's response time to these urgent events may also **increase overall system throughput**.

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. *BASIC Programming Techniques* and the *BASIC Language Reference* fully describe 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[1]

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 ———→   Both types of interrupts are
from Device   signalled to the computer in
                       the same manner.

Interrupt
from
Interface
Hardware
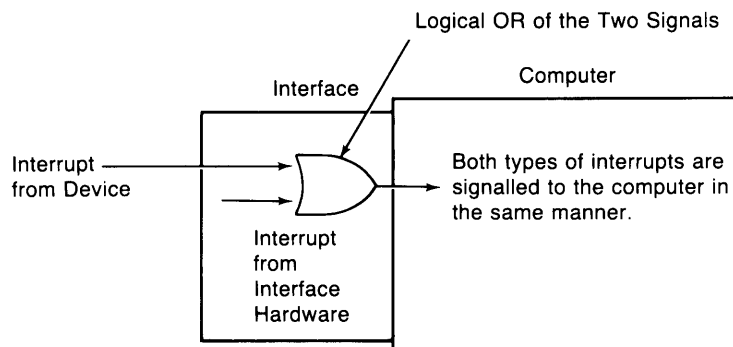
There are **two general types of interrupt events**. The **first** type of event occurs when a **device** determines that it requires the computer to execute a special procedure. The second type occurs when the **interface itself** determines that a condition exists or has occurred that requires the computer's attention.

---

[1] The IO BIN file must be loaded.

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
```

Most Significant Bit   Least Significant Bit

Bit 15	Bit 14			Bit 3	Bit 2	Bit 1	Bit 0
←	Other interrupt causes described in subsquent sections		→			Service Request	See Subsequent Sections
Value = −32 768	Value = 16 384			Value = 8	Value = 4	Value = 2	Value = 1
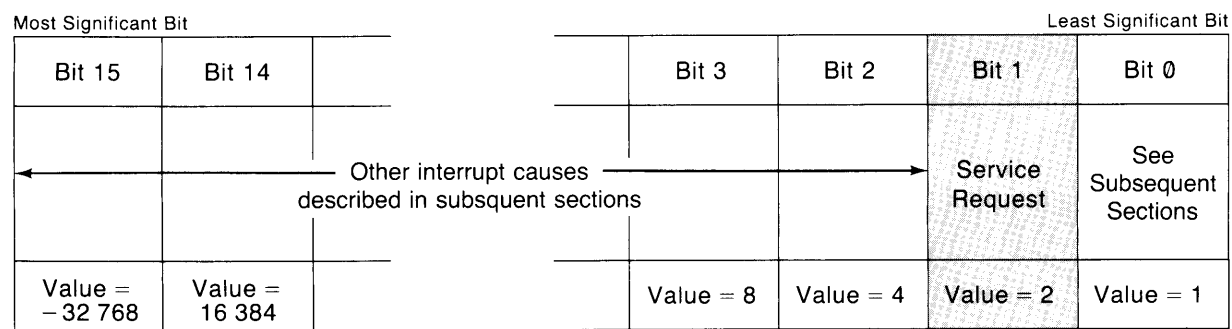
The mask parameter is **optional**. If it is included, the specified value is written into the appropriate register of the specified interface. If this parameter is omitted, the mask specified in the **last** ENABLE INTR is used. If **no** ENABLE INTR statement has been executed for the specified interface, 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          ! 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...has been executed. Consequently, the computer may wait up to 25% longer than the specified time to detect a timeout event; however, it will always **at least** the specified time before generating the interrupt.

There is **no default** timeout time parameter. Thus, if no ON TIMEOUT is executed for a specific interface, the computer will wait **indefinitely** on the device to respond. The only way that the computer can continue executing the program is for the operator to use the (CLR I/O) (or (Break)) 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.

The Internal CRT Interface	Chapter 8

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. The number of lines and columns vary according to which CRT you have.
- Characters OUTPUT to the CRT appear in top area (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.

CRT Display

Series 200 computers can interface with several monitors. Whatever monitor you use, BASIC partitions the CRT display into several regions. The following areas occur in a top to bottom of screen sequence.

- An output area displays output to the CRT (select code 1). The number of lines or columns varies, depending on which monitor you have. See the *BASIC User's Guide* for details.
- A blank line separates the output area from other areas.
- One display line is available. Output to the CRT goes to this line when you use a DISP statement.
- Two keyboard area lines are available. What you type is echoed on the CRT in this two-line area.
- A message/results line is available. The line is truncated to allow display of annunciators for case mode and active menu on systems that have an HP 46020 keyboard. (The menus are SYSTEM, USER 1, USER 2, or USER 3.)
- A two line area at the bottom of the screen permits display of softkey labels. Your system defaults to: labels displayed if you have an HP 46020 keyboard, or labels not displayed if you have a different keyboard.

While the number of lines in the output area can vary, BASIC provides these areas on any CRT you may have.

The Output Area and the Display line.

Characters sent to the CRT with PRINT statements are displayed in output area. The number of lines available can vary depending on what monitor you have. Model 226 and Model 236 computers provide 18 lines. The HP 98781 monitor (bit-mapped) provides 41 lines. Besides some hardware dependency, the number of lines in the output area can be altered by CONTROL Register 13 (of select code 1). You can experiment with the parameters to find the limits for your CRT. The lower limit is 9. Usual upper limits are 25 or 48. Characters sent to the CRT with the DISP statement appear in the display line.

Type in an run the following program to see these areas. Experiment with different parameters for the CONTROL statement in line 10. Try integers in the range 9 to 41. Error messages alert you to out of range situations.

```
10      CONTROL 1,13;25
20      FOR K=1 TO 18
30      PRINT "This is line #";K;" in the output area."
40      WAIT .5
50      DISP "This is line #";K;" in the display line."
60      WAIT .5
70      NEXT K
80      END
```

You should experiment with the loop upper limit in line 20. After the program executes, scroll the displayed text up and down to see what happens. Try values such as 10, 18, 25, and 26 if you have a CRT which displays 25 lines by 80 columns. If you have a bit-mapped CRT, try values such as 10, 30, 48, and 49. Use CONTROL Register 13 to manipulate the size of the output area. Use:

```
STATUS 1,13;Height
```

to return the height of the display screen.

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 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
returnse
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.

Here is another short example program. Typeit and watch the execution. Notice how you can use CONTROL statements and selected registers to control CRT display.

```
10     CONTROL 1,1;10                       ! Go to 10th line
20     PRINT "let's back up a little.";      ! Print a sentence
30     WAIT .6                              ! Time to see it.
40     FOR K=23 TO 1 STEP -1                ! Loop for backspacing
50         CONTROL 1,1;10      ! Stay on 10th line.  Could do other ways
60         CONTROL 1,0;K       ! Move cursor back one space
70         PRINT CHR$(8);" ";  ! Issue backspace
80         WAIT .3             ! Let you see backspacing
90     NEXT K                  ! Continue loop
100    END
```

Display-Enhancement Characters

Some Series 200 CRTs have ability to display underlined, blinking, and inverse-video characters. These features are accessed by displaying special characters. Both the Output Area and the Display Line have these abilities.

There are eight special bit patterns that control the use of these features. CHR$(128) through CHR$(135) cause the following display actions.

Character Code	Action Resulting from Displaying the Character
128	All enhancements off.
129	Inverse mode on.
130	Blinking mode on.
131	Inverse and Blinking modes on.
132	Underline mode on.
133	Underline and Inverse modes on.
134	Underline and Blinking modes on.
135	Underline, Inverse, and Blinking modes on.

No blinking on bit-mapped CRTs.

When one of these characters is sent to the CRT, it turns on the corresponding enhancement(s). All subsequent characters on the CRT are also displayed in the specified enhancement mode; if only a few characters are to be enhanced, a CHR$(128) must be sent to the display after the last character to be enhanced, which turns off **all** enhancements.

From the preceding table, you may have deduced that certain bits within the character bytes turn on these display modes. The following bit pattern and individual bits control these features.

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	0	0	Underline On	Blinking On	Inverse On
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Notice that the upper five bits (7 through 3) must be in the pattern shown (numeric value = 128). Thus, adding the values 4, 2, or 1 enable the Underline, Blinking, and Inverse features. Several examples follow.

```
100     PRINTER IS 1
110     Off=128
120     Underline=4
130     Blinking=2
140     Inverse=1
150     !
160     PRINT CHR$(Off);"Normal"
170     PRINT
180     PRINT CHR$(128+Inverse);"Inverse"
190     PRINT "carries over onto"
200     PRINT "subsequent lines"
210     PRINT
220     PRINT CHR$(128+Underline);"Underline"
230     PRINT "also remains on until turned off"
240     PRINT
250     PRINT CHR$(128+Blinking);"Blinking"
260     PRINT "is the same"
270     PRINT
280     PRINT CHR$(Off);"Back to normal"
290     PRINT
300     END
```

These same features can also be placed in strings by using the (ANY CHAR) key while initially assigning the string variable its value. Keep in mind that, even though these characters are not shown on the screen, they are counted in the length of the string. Dimension string variables accordingly.

The Display Functions Mode

The preceding program showed the control characters which are defined to invoke a special display function when sent to some CRTs. 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 (or softkey). 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. An asterisk is visible in the Display fctns softkey label when the Display functions mode is on. (Only true for the HP 46020 keyboard).

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 (or softkey) 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.

The following program uses the CRT mapping register (lines 50 and 210) and the display functions register (lines 60, 110, 130, 150, 170, and 200) to display the logical and physical CRT character sets.

```
1       ! Output available character sets to CRT
10      PRINT "If you want to see the ROM contents, THEN ENTER any character"
20      PRINT "ELSE, to see the legitimate character set, ENTER a null string"
30      INPUT A$
40      PRINT CHR$(12)        ! Issue a form feed
50      CONTROL 1,11;LEN(A$)   ! Identify which character set
60      CONTROL 1,4;1          ! Set display functions mode
70      ! Provide loop to display cahracter set
80      FOR I=0 TO 255
90         PRINT USING "#,DDD,X,A,X";I,CHR$(I)
100        SELECT 1
110        CASE I=13
120           CONTROL 1,4;0
130           PRINT "";
140           CONTROL 1,4;1
150        CASE NOT (I MOD 13)
160           CONTROL 1,4;0
170           PRINT
180           CONTROL 1,4;1
190        END SELECT
200     NEXT I
210     CONTROL 1,4;0
220     CONTROL 1,11;0
230     PRINT
240     END
```

Output-Area Memory

In addition to the visible display lines in the output area, there may be additional lines 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. The **visible** lines of output-area memory will hereafter be called the **screen**. Parenthetically, bit-mapped monitors (HP 98781) initially have no scrolling area.

```
100    ! Example to show scrolling.
110    !
120    PRINTER IS 1              ! PRINT on CRT.
130    !
140    FOR Line=1 TO 48         ! Write 48 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 portions of the lines within output-area memory. If the display is scrolled **down** as far as possible, there are no lines "above screen". Similarly, if the display is scrolled **up** as far as possible, there are no lines "below screen". The following drawing illustrates the screen of a Model 236 with 6 lines above screen.
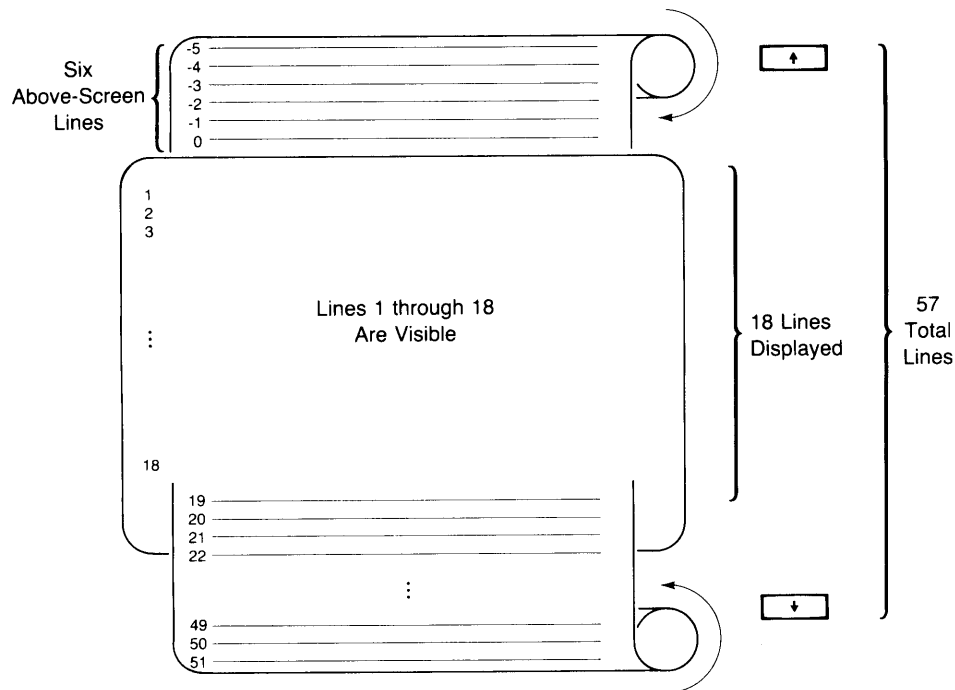
Line Positions of the Output Area

The values could vary if you have a different CRT.

The values could vary if you have a different CRT.

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) or (Shift)-(Clear line)), 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 on a non-bitmapped 25 line CRT.

```
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 varies depending on what monitor you have. 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 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 last 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 48
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 48th 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.

Determining Screenwidth

All programs written and stored on the Model 226 can be run on the Model 236, and vice versa. Programs that use the display extensively have probably been written with the 50-character screenwidth in mind. Since all programs are transportable between these computers, the program should have the ability to distinguish in which computer it is being executed. The BASIC language system provides this capability.

Interface select code 1 is used to access the CRT from BASIC programs. Several registers are associated with this interface which allow interrogating and controlling the CRT through its interface. In particular, STATUS register 9 of the CRT interface is dedicated to storing the current screenwidth. The following statement determines the current screenwidth.

```
STATUS 1,9;Screenwidth
```

The resultant value of Screenwidth differs for each computer: the value of 80 is returned in the Model 216, 217, and 236. Fifty is returned in the Model 226. Other values may be returned for external monitors.

Scrolling the Display

A 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. Assuming a 25 line CRT, 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 1 TO 48
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. Display-enhancement characters, CHR$(128) through CHR$(143), cannot be entered from the CRT memory.

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

Again assuming a 25 line CRT, 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$(48)[50] ! To read 48 lines.
120     !
130     FOR Screen_line=1 TO 48
140         OUTPUT 1;"Line";Screen_line
150     NEXT Screen_line
160     WAIT 1
170     !
180     STATUS 1,3;Lines_above
190     CONTROL 1;1,-Lines_above+1    ! Scroll to read
200     ENTER 1 USING "K";Memory$(*) ! entire memory.
210     !
220     FOR Screen_line=1 TO 48        ! 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 last column (50, 80, or 128 depending on your CRT), 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 the last column.

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

The display-enhancement characters produce the same effects in the DISP Line as in the Output Area.

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
```

Softkey Labels

Softkeys can be defined as: SYSTEM or USER labels on systems which use the HP 46020 keyboards, typing-aid keys, or as keys that initiate program (ON KEY) branches. In any usage, two display lines (near the bottom of the CRT) can be used for key labels. The topic of typing-aid keys is discussed in Chapter 2 of *BASIC Programming Techniques*; using softkeys to initiate program branches is discussed in Chapter 3 of the same manual. Softkey labels for systems which use the HP 46020 keyboard are discussed in the *BASIC User's Guide*.

Softkey labels can be turned off and on by writing to CRT CONTROL Register 12. The values written to the register have the following effects:

Value of CRT Register 12	Effect on Key Labels
0	Typing-aid key labels are *displayed until the program is run*, at which time they are turned off (until at least one ON KEY is executed). Annunciators, if present, stay on. SYSTEM menu softkeys are displayed even when a program is running.
1	Typing-aid and softkey labels are *not displayed at any time*.
2	Typing-aid and softkey labels are *displayed at all times* This is the default for systems with the HP 46020 keyboard..

The default value of this register is 0 (2 for systems using HP 46020 keyboard), which is restored at power-on and when SCRATCH A is executed. The register's current contents can be determined by reading STATUS Register 12.

Here is an illustrative program which shows cycles through Register 12 with the values 0, 1, and 2.

```
1       ! Set up clear screen mechanism and subroutine
3       C$=CHR$(255)&"K"
4       GOSUB 1000
10      ! Toggle key displays
20      PRINT "Softkey labels are toggled."
21      WAIT 2
30      ! Set up toggle loop.
31      FOR J=1 TO 3
40        FOR Toggle=0 TO 2
41          GOSUB 1000
50          PRINT "Flag value =";Toggle
60          CONTROL 1,12;Toggle
70          WAIT 1.5
80        NEXT Toggle
81      NEXT J
90      !
100     STOP
1000    OUTPUT 2 USING "#,K";C$
1010    RETURN
1020    END
```

If you have an HP 46020 keyboard, try running the program with the USER menu up and with the SYSTEM menu up.

You can use CONTROL register 2 (of select code 1) to cycle the menus; 0 = SYSTEM, 1 = USER 1, 2 = USER 2, and 3 = USER 3. For example,

```
    CONTROL 1,2;3
```

displays the menu for USER 3.

CONTROL resister 14 (of select code 1) can be used to invoke function key bases 0 or 1 (i.e. f1 to f0, f2 to f1, etc.). The default is 0, which means the function keys start at 1. Changing the flag value to 1 starts the function keys at 0. You might also need to deal with this because the function keys on some keyboards are labeled from 0 to 9, and from 1 to 8 on other keyboards.

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	Set default alpha color:
	For Alpha Displays:

Value	Result
< 128	Error
128-135	Ignored
136	White
137	Red
138	Yellow
139	Green
140	Cyan
141	Blue
142	Magenta
143	Black
144-159	Ignored
> 159	Error

For Bit-Mapped Displays:
Values 0 thru 255 which correspond to the graphics pens.

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

CRT Status and Control Registers (cont.)

Status Register 11   CRT character mapping flag
Control Register 11   Disable CRT character mapping if non-0

Status Register 12   Key labels display mode
Control Register 12   Set key labels display mode: 0 = typing-aid key labels displayed unless program is in the RUN state
                  1 = key labels always off
                  2 = key labels displayed at all times

Status Register 13   CRT height
Control Register 13   CRT height: number of lines in Alpha display must be greater than 8.

Status Register 14   Display replacement

Control Register 14   Display replacement

0 – 0
1 – source AND old
2 – source AND NOT old
3 – source:default
4 – NOT source AND old
5 – old
6 – source EXOR old
7 – source OR old
8 – source NOR old
9 – source EXNOR old
10 – NOT old
11 – source OR NOT old
12 – NOT source
13 – NOT source OR old
14 – source NAND old
15 – 1

The Internal Keyboard Interface	Chapter 9

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 computer 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 2^1 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.
- If present, 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.

Note

Not all Series 200 computers use the same keyboards. There are substantial differences between the HP 46020 keyboard and any other keyboard. You should be very familiar with using your keyboard before you read this chapter. An attempt to generalize techniques was made, but certain examples relate to certain keyboards.

---

1 BASIC provides a numeric function, KBD, which returns a value of 2.

ASCII and Non-ASCII Keys

The keys of the Series 200 computer keyboards 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 (Return), (ENTER), 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 (Extend char) key, when present, is held down while you press other keys from the main typewriter section to generate the rest of the available 256 ASCII characters.

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 keys(s). This is particularly useful when you need to include a control character in a string.

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)	Nu
1	SOH	Start of Header	(A)	SH
2	STX	Start of Text	(B)	SX
3	ETX	End of Text	(C)	EX
4	EOT	End of Transmission	(D)	ET
5	ENQ	Enquiry	(E)	EQ
6	ACK	Acknowledgement	(F)	AK
7	BEL	Bell	(G)	⌂
8	BS	Backspace	(H)	BS
9	HT	Horizontal Tab	(I)	HT
10	LF	Line-Feed	(J)	LF
11	VT	Vertical Tab	(K)	VT
12	FF	Form-Feed	(L)	FF
13	CR	Carriage-Return	(M)	CR
14	SO	Shift Out	(N)	SO
15	SI	Shift In	(O)	SI
16	DLE	Data Link Escape	(P)	DL
17	DC1	Device Control	(Q)	D1
18	DC2	Device Control	(R)	D2
19	DC3	Device Control	(S)	D3
20	DC4	Device Control	(T)	D4
21	NAK	Neg. Acknowlegdement	(U)	NK
22	SYN	Synchronous Idle	(V)	SY
23	ETB	End of Text Block	(W)	EB
24	CAN	Cancel	(X)	CN
25	EM	End of Media	(Y)	EM
26	SUB	Substitute	(Z)	SB
27	ESC	Escape	([ )	EC
28	FS	File Separator	(SHIFT)-([ )	FS
29	GS	Group Separator	( ])	GS
30	RS	Record Separator	(↑)	RS
31	US	Unit Separator	(SHIFT)-(/)	US

Pressing the (ESC) key on the HP 46020 keyboard is an alternative to (CTRL)([ ). 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.

(CTRL)-(J)

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 within a 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 (or the softkey in the SYSTEM menu) 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.
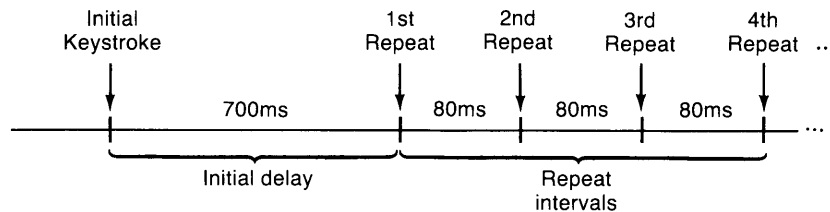
These intervals can be changed from the program, if desired, by writing different values into CONTROL registers 3 and 4 (of interface select code 2). Register 3 contains the parameter that controls the auto-repeat interval, and register 4 contains the parameter that controls the initial delay. The values 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 1, 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.

Note

Softkey keyboard labels can range from **k0, k1, ..., k9** or **f1, f2, ..., f8** (depending on what keyboard you have). You may want to shift softkey labels. Use Control Register 14 (of interface select code 2). CONTROL 2,14;1 makes f1 = k0, f2 = k1, and so on. CONTROL 2,14;0 is the default (f1 = k1, f2 = k2, and so on). Applies only to HP 46020 keyboards.

```
100   ON KEY 1 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 Data 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 (or softkey).

Pressing the (ENTER), (Return), (STEP), or (CONTINUE) key or softkey 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 keys or softkeys in the above paragraph 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.

(Return) or (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 (Return), (ENTER), or (STEP) keys). Pressing the (Return), (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) (or (Shift) (•) on the numeric keypad) 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 (Return), (ENTER), (STEP), or (CONTINUE) key (or softkey).

(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.

Sending Data to the Keyboard

Characters output to the keyboard are indistinguishable from characters typed in from the keyboard. All charactes output to the keyboard, **including control characters**, are displayed in the keyboard area. The following program outputs the BEEP statement to the keyboard. Read on to see how it works.

```
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 (ENTER) or (Return) key after the program has ended executes the statement. Modify the program to "press" the (ENTER) or (Return) key by typing in (CTRL) (ENTER) (or (Return)) following the BEEP. Sending this special two-character sequence to the keyboard is equivalent to the operator pressing the (ENTER) or (Return) 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$(255) does not generate the same character on most printers as it does on the Model 226 display, it is recommended that some explicit means of documenting these character sequences be employed. For instance, string variables can be defined to contain these sequences; then when the program is listed on an external printer, it will be much easier to determine which non-typing keys are being represented. The (CTRL) key is still used with the non-ASCII key to generate the two-character sequence, but the special character should be changed to a CHR$(255).

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

Second Byte of Non-ASCII Key Sequences (String)

Holding the CTRL key and pressing a non-ASCII key generates a two-character sequence on the CRT. The first character is an "inverse-video" K. This table can be used to look up the key that corresponds to the second character of the sequence. (On the small keyboard of the Model 216, some non-ASCII keys generate ASCII characters when they are pressed while holding the CTRL key.)

Character	Value	Key
space		[1]
!	33	STOP
"		[1]
#	35	CLR LN
$	36	ANY CHAR
%	37	CLR→END
&	38	Select
'	39	Prev
(	40	SHIFT-TAB
)	41	TAB
*	42	INS LN
+	43	INS CHR
,	44	Next
-	45	DEL CHR
.	46	Ignored
/	47	DEL LN
0	48	k0
1	49	k1
2	50	k2
3	51	k3
4	52	k4
5	53	k5
6	54	k6
7	55	k7
8	56	k8
9	57	k9
:	58	SHIFT-system f6 [2]
;	59	SHIFT-system f7 [2]
<	60	←
=	61	RESULT
>	62	→
?	63	RECALL
@	64	SHIFT-RECALL
A	65	PRT ALL
B	66	BACK SPACE
C	67	CONTINUE
D	68	EDIT
E	69	ENTER
F	70	DISPLAY FCTNS
G	71	SHIFT-→
H	72	SHIFT-←
I	73	CLR I/O
J	74	Katakana Mode
K	75	CLR SCR
L	76	GRAPHICS
M	77	ALPHA
N	78	DUMP GRAPHICS
O	79	DUMP ALPHA

Character	Value	Key
P	80	PAUSE
Q		[1]
R	82	RUN
S	83	STEP
T	84	SHIFT-↓
U	85	CAPS LOCK
V	86	⊣
W	87	SHIFT-↑
X	88	EXECUTE
Y	89	Roman Mode
Z		[1]
[	91	CLR TAB
\	92	▼
]	93	SET TAB
..	94	↑
_	95	SHIFT-▼
`		[1]
a	97	k10
b	98	k11
c	99	k12
d	100	k13
e	101	k14
f	102	k15
g	103	k16
h	104	k17
i	105	k18
j	106	k19
k	107	k20
l	108	k21
m	109	k22
n	110	k23
o	111	SHIFT-system f1 [2]
p	112	SHIFT-system f2 [2]
q	113	SHIFT-system f3 [2]
r	114	SHIFT-system f4 [2]
s	115	SHIFT-user f1 [2]
t	116	SHIFT-user f2 [2]
u	117	SHIFT-user f3 [2]
v	118	SHIFT-user f4 [2]
w	119	SHIFT-user f5 [2]
x	120	SHIFT-user f6 [2]
y	121	SHIFT-user f7 [2]
z	122	SHIFT-user f8 [2]
}	123	System
|	124	Menu
{	125	User
~	126	SHIFT-Menu
▓		[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] System and user refer to the softkey menu which is currently active.

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    En$=CHR$(255)&"E"   ! ENTER or Return key.
120    Up$=CHR$(255)&"^"   ! Up arrow key.
130    Prt$=CHR$(255)&"A"  ! PRT ALL key or softkey.
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""";En$;Up$;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




               (blank)




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.

Softkeys

The keys on the upper-left portion of the keyboard are called "softkeys." These keys can be defined by BASIC programs to initiate program branches. These keys can be defined as typing-aid keys, which produce keystrokes just as if you had typed them in yourself.

Brief examples of using the softkeys have already been presented in Chapter 7 and in earlier in this chapter (see "Modifying the Repeat and Delay Intervals"). Typing-aid keys are discussed in Chapter 2 of *BASIC Programming Techniques*; softkeys are also briefly described in Chapter 3 of the same manual.

Sensing Knob Rotation

Your computer system may, or may not, have a knob or a mouse (HP 46060). In any event, the programs below are illustrative. It is assumed that you will use the techniques and apply them to your programming situation.

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 400
130          WAIT .2
140          DISP Iteration
150     NEXT Iteration
160     !
170     STOP
180     !
190 Knob:    STATUS 2,10;Key_with_Knob
200          PRINT KNOBX;" pulses "; KNOBY;" 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
```

Parenthetically, the .25 parameter in line 100 takes no action with a mouse. A mouse always acts as if the parameter is .02. Otherwise, the interval parameter of 0.25 seconds was specified in the preceding program so 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 and KNOBY functions 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.

The next program illustrates use of an ON KNOB statement with a mouse. Note changes in iteration as you move the mouse.

```
10      COM /Knob/ Kx,Ky
20      Kx=0
30      Ky=0
40      ON KNOB 1 CALL Knob
50      PRINT TABXY(1,1);" "
60      FOR I=1 TO 1.E+6
70         DISP I
80         PRINT TABXY(1,2);Kx;Ky;"                    "
90      NEXT I
100     END
110     SUB Knob
120        COM /Knob/ Kx,Ky
130        INTEGER Knx,Kny
140        Knx=KNOBX
150        Kny=KNOBY
160        Kx=Kx+Knx
170        Ky=Ky+Kny
180        PRINT TABXY(1,5);Knx;Kny;"                    "
190     SUBEND
```

Note

If you have programs written in BASIC 2.0/2.1, refer to the Knob section in Chapter 15 of the *BASIC 3.0 Programming Techniques* manual for more information.

Enhanced Keyboard Control

Normally, the BASIC operating system handles all keyboard inputs. Several BASIC statements allow programs to handle inputs from the keyboard; examples are the INPUT, LINPUT, ENTER, ON KEY, and ON KNOB statements. Additional keyboard statements provide BASIC programs with a means of intercepting both ASCII and non-ASCII keystrokes for processing by the program. The statements are:

ON KBD	sets up and enables keystrokes to be trapped.
ON KBD,ALL	includes (PAUSE), (STOP), (CLR I/O), and softkeys.
KBD$	returns keystrokes trapped in the buffer.
OFF KBD	resumes normal keystroke processing.

ON KBD allows terminal emulation, keyboard masking, and special data inputs. Each keystroke produces unique code(s) that allow the program to differentiate between different keys being pressed. The program can also determine whether the (SHIFT) or (CTRL) keys are being pressed with a particular key, but these keystrokes cannot be detected by themselves. Also, the (RESET) key cannot be trapped by ON KBD.

Trapping Keystrokes

The ON KBD statement sets up a branch that is initiated when the keyboard buffer becomes "non-empty". The service routine may then interrogate the buffer as desired, processing the keystrokes as determined by the program. The keyboard buffer of Series 200 computers contains up to two CRT lines. Calling the KBD$ function does two things: it returns all keystrokes trapped since the last time the buffer was read, and it then clears the keyboard buffer.

The following program uses ON KBD, KBD$, and OFF KBD to trap and process keystrokes, rather than allowing the operating system to do the same. The program defines each keystroke to print a complete word.

```
100    OPTION BASE 1
110    DIM String$(26)[6]
120    READ String$(*)
130    !
140    DATA A,BROWN,CAT,DOG,EXIT,FOX,GOT
150    DATA HI,IN,JUMPS,KICKED,LAZY,MY
160    DATA NO,OVER,PUSHED,QUICK,RED,SMART
170    DATA THE,UNDER,VERY,WHERE,XRAY,YES,ZOO
180    !
190    PRINTER IS 1
200    PRINT "Many ASCII keys have been"
210    PRINT "defined to produce words."
220    PRINT
```

```
230    PRINT "Press the following keys."
240    PRINT "T Q B F J O T L D ."
250    !
260    ON KBD GOSUB Process_Keys
270    !
280    LOOP
290      EXIT IF Word$="EXIT"
300    END LOOP
310    !
320    STOP
330    !
340 Process_Keys: Key$=KBD$   ! Read buffer.
350    !
360    REPEAT ! Process ALL keys trapped.
370      Key_code=NUM(Key$[1;1])! Calculate code.
380      !
390      SELECT Key_code            ! Choose response.
400      !
410        CASE 65 TO 90            ! CASE "A" TO "Z".
420          Word$=String$(Key_code-64)
430          Key$=Key$[2]           ! Remove processed key.
440          !
450        CASE 97 TO 122           ! CASE "a" TO "z".
460          Word$=String$(Key_code-96)
470          Key$=Key$[2]           ! Remove processed key.
480          !
490        CASE 255                 ! CASE non-ASCII key.
500          IF Key$[2;1]<>CHR$(255) THEN
510            Word$=Key$[1,2]      ! Non-ASCII key alone,
520            Key$=Key$[3]         ! so take 2 codes.
530          ELSE
540            Word$=Key$[1,3]      ! Non-ASCII w/ CTRL,
550            Key$=Key$[4]         ! so take 3 codes.
560          END IF
570        CASE ELSE                ! CASE all others.
580          Word$=""
590          Key$=Key$[2]           ! Remove processed key.
600          !
610      END SELECT
620      !
630                                 ! Execute response.
640      Defined=LEN(Word$)<>0
650      IF Defined THEN
660        PRINT Word$;" ";
670        DISP
680      ELSE
690        BEEP 100,.05
700        DISP "Key undefined."
710      END IF
```

```
720    !
730    UNTIL LEN(Key$)=0 ! Until ALL keys processed,
740    !
750    RETURN
760    !
770 Quit:   END
```

Notice that all non-ASCII keys produce two-character sequences: CHR$(255) followed by an ASCII character. Pressing the (CTRL) key with non-ASCII keys produce three-character sequences: another CHR$(255) character preceding the two-character sequence produced by pressing the non-ASCII key by itself. See the tables in "Outputs to the Keyboard" for a listing of the sequences produced by non-ASCII keys.

BASIC programs can output ASCII keystrokes to the keyboard, via OUTPUT 2, without initiating an ON KBD branch; however, outputting non-ASCII "closure" keys will initiate the ON KBD branch. For example, executing the following statement (in a program line):

    OUTPUT 2;"32*2";CHR$(255);"E";"KBD";

causes the characters K B D which follow the closure key to be placed in the KBD$ buffer, which also initiates the ON KBD branch. The (EXECUTE)-key (or equivalent such as (Return)) sequence which was sent to the keyboard executes the numeric expression 32*2 before the branch is initiated. This type of operation may result in unpredictable results and is therefore not recommended while ON KBD is in effect.

ON KBD branching is disabled by DISABLE, deactivated by OFF KBD, and temporarily deactivated when the program is executing LINPUT, INPUT, or ENTER 2 statements.

Softkeys and Knob Rotation
When ON KNOB is not in effect, knob rotation is also trapped by ON KBD. Rotation will produce the "cursor" keystrokes; clockwise rotation produces CHR$(255)followed by "^", while counter-clockwise rotation produces CHR$(255) followed by "V".

ON KBD ,ALL allows softkey trapping ("overrides" ON KEY) but does not change the softkey labels.

Disabling Interactive Keyboard
Another group of statements is used to disable the interactive keyboard functions:

SUSPEND INTERACTIVE   ignores the (PAUSE), (STOP), (STEP), and.(CLR I/O)
                             keys and disables live keyboard execution.

SUSPEND INTERACTIVE ,RESET   ignores (RESET) too.

RESUME INTERACTIVE   returns to normal operation

SUSPEND INTERACTIVE can be used to prevent interruption of programs which gather data or which control other systems.

Special care should be taken when using SUSPEND INTERACTIVE,RESET. If an "infinite loop" is executed while interactive keyboard functions are disabled, only the power switch will stop execution of the program.

```
110     ! This program cannot be stopped by
120     ! PAUSE, STOP, OR RESET
130     ! before its normal completion
140     !
150     !
160     SUSPEND INTERACTIVE,RESET ! ignore keyboard
170     !
180     PRINT "COUNTDOWN IS "
190     PRINT
200     I=10                        ! Initial value,
210       REPEAT
220         PRINT " T minus ";I   ! Print count,
230         I=I-1                  ! Decrement count,
240         WAIT 1                 ! Wait one second,
250       UNTIL I<0
260     !
270     PRINT
280     BEEP 100,1
290     PRINT "Done"
300     RESUME INTERACTIVE         ! Return to normal,
310     !
320     END
```

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. Then 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 computer, 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.

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   Function key menu.
Control Register 2   Function key menu.   0 = System menu
                                             1-3 = User menu 1 thru 3

Status Register 3   Undefined

Control Register 3   Set auto-repeat interval. If 1 thru 255, repeat rate in milliseconds is 10 times this value. 256 = turn off auto-repeat. (Default at power-on or SCRATCH A is 80 ms.)

Status Register 4   Undefined

Control Register 4   Set delay before auto-repeat. If 1 thru 256, delay in milliseconds is 10 times this value. (Default at power-on or SCRATCH A is 700 ms.)

Status Register 5   KBD$ buffer overflow register.   1 = overflow. Register is reset when read.
Control Register 5   Undefined

Status Register 6   Typing aid expansion overflow register. 1 = overflow. Register is reset when read.

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

Keyboard Status and Control Registers (cont.)

Status Register 8   Keyboard language jumper

0 = US ASCII	6 – Canadian English	12 – Dutch
1 = French	7 – United Kingdom	13 – Swiss German
2 = German	8 – Canadian French	14 – Latin(Spanish)
3 = Swedish	9 – Swiss French	15 – Danish
4 = Spanish	10 – Italian	16 – Finnish
5 = Katakana	11 – Belgian	17 – Norwegian

Control Register 8   Undefined

Status Register 9

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Internal Use	Internal Use	1 = HP46020A Keyboard 0 = Other Keyboard	1 = No Keyboard 0 = Keyboard Present	1 = n-Key Rollover 0 = 2 or less Key Rollover	0	0	1 = HP98203A Keyboard 0 = Other Keyboard
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Control Register 9   Undefined

Status Register 10   **State at Last Knob Interrupt**

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   0 = horizontal-pulse mode; 1 = all-pulse mode
Control Register 11   (default is 0 without KNB2_0 loaded, 1 with KNB2_0 loaded)
Refer to the Knob section in Chapter 15 of the *BASIC 3.0 Programming Techniques* manual for more information.

Status Register 12   "Pseudo-EOI forCTRL-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

Status Register 14   Function keys on HP 46020A software key numbers shifted.
Control Register 14   Function keys on HP 46020A software key numbers shifted.
0 = (f1) is Key 1; default
1 = (f1) is Key 0

I/O Path Attributes	Chapter
	10

This chapter contains two major topics, both of which involve additional features provided by I/O path names. The first topic is that I/O path names can be given attributes which control the way that the system handles the data sent and received through the I/O path. Attributes are available for such purposes as controlling data representations, generating and checking parity, and defining special end-of-line (EOL) sequences.

The second topic is that the same I/O statements can be used to access most system resources, including the CRT, keyboard, mass storage files, and buffers (rather than using a separate set of BASIC statements for each type of resource). This second topic, herein called "unified I/O," may be considered an implicit attribute of I/O paths.

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.[1] If the I/O path possesses the FORMAT OFF attribute, the computer'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".

---

[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 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**; this is also the default for buffers. 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 interpreted as ASCII and then converted back into its internal representation. The following diagram pictorially describes these operations.
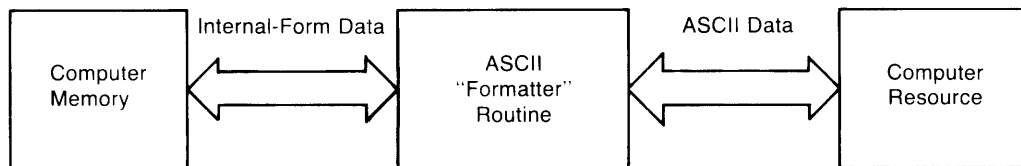
The FORMAT ON Attribute Requires Data To Be Formatted

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.
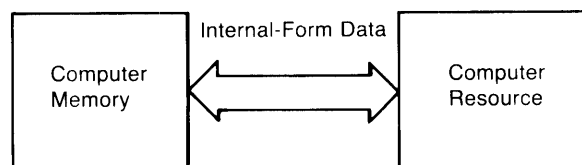
The Internal Data Representation Is Maintained with FORMAT OFF

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.

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 of Initially Assigning an Attribute

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

Example of Changing an Attribute

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.

```
200    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 of Restoring the Default Format Attribute

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

```
340    ASSIGN @Device ! Restores the default attributes,
```

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**.
- no non-default CONVERT or PARITY attribute may be assigned to the I/O path (see subsequent sections).

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 bytes, 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 two-byte, binary length header followed by the actual string characters. When a string is output, a four-byte length header (the first two bytes of which are zeros) 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. If the string length is odd, the source must send an extra trailing byte to make an even number of characters.

---

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 I/O) or (Break)) key is pressed before all characters have been received), the variable contains the characters that have been received.

Additional Attributes

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

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

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

The BYTE and WORD Attributes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
┌───┬───┬───┬────┬────┬────┐
│ O │ d │ d │ CR │ LF │NUL │
└───┴───┴───┴────┴────┴────┘
  └──┬──┘ └──┬──┘ └──┬──┘
  Word 1  Word 2  Word 3
```

```
┌───┬───┬───┬────┬────┬────┬───┬───┬───┬───┬────┬────┐
│ O │ d │ d │ CR │ LF │NUL │ E │ v │ e │ n │ CR │ LF │
└───┴───┴───┴────┴────┴────┴───┴───┴───┴───┴────┴────┘
  └──┬──┘ └──┬──┘ └──┬──┘ └──┬──┘ └──┬──┘ └──┬──┘
  Word 1  Word 2  Word 3  Word 4  Word 5  Word 6
```
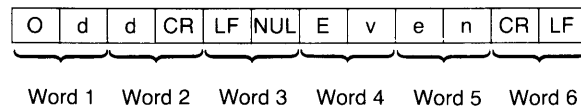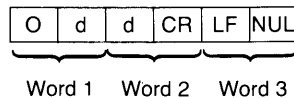
In the first statement, a Null was sent after the EOL characters to flush the buffer and force word alignment for a subsequent OUTPUT. The second statement shows that a pad byte will be sent after any EOL sequence when required to achieve word alignment; the Null pad byte was not needed after the second EOL sequence. In addition, if a buffer or file pointer currently has an odd value, a leading pad byte will be output to force word alignment before any data are sent by the OUTPUT statement.

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

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

The variables receive the following values:

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

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

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

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

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

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

A	NUL	B	C	CR	LF

Word 1   Word 2   Word 3
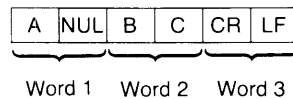
The Null pad byte was sent before the "W" image data to align the INTEGER specified by the "W" on a word boundary.

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

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

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

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

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

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

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

```
ASSIGN @Printer TO 12
```

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

```
ASSIGN @Printer;WORD
```

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

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

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

```
ASSIGN @Device
```

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

Converting Characters

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

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

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

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

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

The example program's output is as follows.

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

To perform the lowercase-to-uppercase conversion, it was not necessary to include characters with codes 123 through 255 in the conversion string, since these characters are not to be converted. They were included to emphasize that the 256th character must be included in the string if CHR$(0) is to be converted with this method. The CONVERT attribute is then assigned to the I/O path, and all subsequent data sent through the I/O path (while CONVERT is in effect) will be converted.

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

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

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

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

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

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

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

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

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

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

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

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

Changing the EOL Sequence

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

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

```
ASSIGN @Device TO 12;EOL "LFLFCR"
```

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

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

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

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

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

```
ASSIGN @Device;EOL "CRLF" DELAY 0.1
```

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

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

Parity Generation and Checking

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Determining the Outcome of ASSIGN Statements

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

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

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

The following statement shows a method of determining the Open/Closed status of the I/O path.

```
ASSIGN @Device;RETURN Closed_status
```

If @Device is currently Open, then 0 is returned; if it is Closed, then 177 is returned (Undefined I/O path name). When RETURN is used in this manner, the default attributes are not restored.

When RETURN is used in this manner, ON ERROR is normally disabled during the ASSIGN statement; however, there are certain errors which cannot be trapped by using RETURN in the ASSIGN statement.

If more than one error occurred during the ASSIGN, there is no assurance that the error number returned is either the first or the last error.

Concepts of Unified I/O

The computer's BASIC language system 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 (through the CRT and keyboard) by using these I/O statements. The next section of this chapter describes how data can be moved to and from string variables with OUTPUT and ENTER statements. Chapter 11 describes how to use buffers and TRANSFER, which can also be used to communicate with several system resources.

Chapter 12 describes how these I/O statements are used to communicate with HP-IB peripheral devices. And, if you have read about mass storage operations (Chapter 7 of *BASIC Programming Techniques*), you know that the ENTER and OUTPUT statements are also used to move data between the computer and mass storage files. This ability to move data between the computer and all of its resources with the same statements is a very powerful capability of the computer's BASIC language.

Before briefly discussing I/O paths to mass storage files, the following discussion will present some background information that will help you understand the rationale behind implementing the two data representations used by the computer. The remainder of this chapter then presents several uses of this language structure.

Data-Representation Design Criteria

As you know, the computer supports two general data representations — the ASCII and the internal representations. This discussion presents the rationale of their design.

The data representations used by the computer were chosen according to the following criteria.

- to maximize the rate at which computations can be made
- to maximize the rate at which the computer can move the data between its resources
- to minimize the amount of storage space required to store a given amount of data
- to be compatible with the data representation used by the resources with which the computer is to communicate

The **internal representations** implemented in the computer are designed according to the **first three of the above criteria**. However, the last criterion must always be met if communication is to be achieved. If the resource uses the ASCII representation, this compatibility requirement takes precedence over the other design criteria. The **ASCII representation** fulfills this **last criterion** for most devices and for the computer operator. The first three criteria are further discussed in the following description of data representations used for mass storage files.

I/O Paths to Files

There are 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 computer 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. Both numeric and string **computations are much faster.** 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) and numeric data can be represented with fewer bits.

The **transfer time** required for each data item is also **decreased.** Numeric output operations are always much faster because the data are not re-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.

```
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
Buffers	FORMAT ON	Yes

Applications of Unified I/O

This section describes two uses of the powerful unified-I/O scheme of the computer. The first application contains further details and uses of I/O operations with string variables. The second application involves using a disc file to simulate a device.

I/O Operations with String Variables

Chapter 3 briefly described how string variables may be specified as the source or destination of data in I/O statements, but it described neither the details nor many uses of these operations. This section describes both the details of and several uses of outputting data to and entering data from string variables.

Outputting Data to String Variables

When a string variable is specified as the destination of data in an OUTPUT statement, source items are evaluated individually and placed into the variable according to the free-field rules or the specified image, depending on which type of OUTPUT statement is used. Thus, item terminators may or may not be placed into the variable. The ASCII data representation is always used during outputs to string variables; in fact, **data output to string variables is exactly like that sent to devices through I/O paths with the FORMAT ON attribute.**

Characters are always placed into the variable beginning at the first position; no other position can be specified as the beginning position at which data will be placed. Thus, **random access of the information in string variables is not allowed** from OUTPUT and ENTER statements; all data must be accessed serially. For instance, if the characters "1234" are output to a string variable by one OUTPUT statement, and a subsequent OUTPUT statement outputs the characters "5678" to the same variable, the second output **does not** begin where the first one left off (i.e., at string position five). The second OUTPUT statement begins placing characters in position one, just as the first OUTPUT statement did, overwriting the data initially output to the variable by the first OUTPUT statement.

---

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 if all characters are read from the string; the ENTER statement automatically terminates when the last character is read from the variable. However, **item** terminators are still required **if** the items are to be separated **and** the lengths of the items are not known. If the length of each item is known, an image can be used to separate the items.

Example

The following program shows an example of the need for **either** item terminators **or** length of each item. The first item was not properly terminated and caused the second item to not be recognized.

```
100    OUTPUT String$;"ABC123";  ! OUTPUT w/o CR/LF.
110    !
120    ! Now enter the data.
130    ON ERROR GOTO Try_again
140    !
150 First_try: !
160    ENTER String$;Str$,Num
170    OUTPUT 1;"First try results:"
180    OUTPUT 1;"Str$=  ";Str$,"Num=";Num
190    BEEP      ! Report getting this far.
200    STOP
210    !
220 Try_again: OUTPUT 1;"Error";ERRN;" on 1st try"
230               OUTPUT 1;"STR$=";Str$,"Num=";Num
240               OUTPUT 1
250               OFF ERROR  ! The next one will work.
260               !
270    ENTER String$ USING "3A,3D";Str$,Num
280    OUTPUT 1;"Second try results:"
290    OUTPUT 1;"Str$=  ";Str$,"Num=";Num
300    !
310    END
```

This technique is convenient when attempting to enter an unknown amount of data or when numeric and string items within incoming data are not terminated. The data can be entered into a string variable and then searched by using images.

Example

ENTERs from string variables can also be used to generate a number from ASCII numeric characters (a recognizable collection of decimal digits, decimal point, and exponent information), rather than using the VAL function. As with outputs to string variables, images can be used to interpret the data being entered.

```
30    Number$="Value= 43.5879E-13"
40    !
50    ENTER Number$;Value
60    PRINT "VALUE=";Value
70    END
```

Taking a Top-Down Approach

This application shows how the computer's BASIC-language structure may help simplify using a "top-down" programming approach. In this example, a simple algorithm is first designed and then expanded into a program in a general-to-specific, stepwise manner. The top-down approach shown here begins with the general steps and works toward the specific details of each step in an orderly fashion.

One of the first things you **should** do when programming computers is to **plan the procedure before actually coding any software**. At this point of the design process, you need to have a good understanding of both the problem and the requirements of the program. The general tasks that the program is to accomplish must be described before the order of the steps can be chosen. The following simple example goes through the steps of taking this top-down approach to solving the problem.

Problem: write a program to monitor the temperature of an experimental oven for one hour.

*Step 1. Verbally describe what the program must do in the **most general** terms. You may want to make a chart or draw a picture to help visualize what is required of the program.*

Initialize the monitoring equipment. Start the timer and turn the oven on. Begin monitoring oven temperature and measure it every minute thereafter for one hour. Display the current oven temperature, and plot the temperatures vs. time on the CRT.

Step 2. Verbally describe the algorithm. Again, try to keep the steps as general as possible.

This process is often termed writing the "pseudo code". Pseudo code is merely a written description of the procedure that the computer will execute. The pseudo code can later be translated into BASIC-language code.

> Setup the equipment.
>
> Set the oven temperature and turn it on.
>
> Initialize the timer.
>
> Perform the following tasks every minute for one hour.
>
>> Read the oven temperature.
>>
>> Display the current temperature and elapsed time.
>>
>> Plot the temperature on the CRT.
>
> Turn the oven and equipment off.
>
> Signal that the experiment is done.

Step 3. Begin translating the algorithm into a BASIC-language program.

The following program follows the general flow of the algorithm. As you become more fluent in a computer language, you may be able to write pseudo code that will translate more directly into the language. However, avoid the temptation to write the initial algorithm in the computer language, because writing the pseudo code is a **very important** step of this design approach!

```
100    ! This program: sets up measuring equipment,
110    ! turns an oven on, and initializes a timer.
120    ! The oven's temperature is measured every
130    ! minute thereafter for one hour. The temp.
140    ! readings are displayed and plotted on the
150    ! CRT.
160    !
170    Rdgs_interval=60   ! 60 seconds between readings.
180    Test_length=60     ! Run test for 60 minutes.
190    !
200    CALL Equip_setup
210    CALL Set_temp
220    GOSUB Start_timer
230    !
240 Keep_monitoring: ! Main loop.
250                        !
260    GOSUB Timer
270    !
280    IF Seconds<=Rdgs_interval THEN
290        GOTO Keep_monitoring
300    ELSE
310        Minutes=Minutes+1
320        CALL Read_temp
330        CALL Plot_temp
340    END IF
350    !
360    !
370    IF Minutes<Test_length THEN
380        GOTO Keep_monitoring
390    ELSE
400        CALL Off_equip
410        PRINT "End of experiment"
420    END IF
430    !
440    STOP
450    !
460    !
470    ! First the subroutines.
480    !
490 Start_timer: Init_time=TIMEDATE
500              PRINT "Timer initialized."
510              PRINT
520              PRINT
530              RETURN
540                !
550 Timer: !
560        Seconds=TIMEDATE-Minutes*60-Init_time
570        DISP USING Time_image;Minutes,Seconds
580 Time_image: IMAGE "Time: ",DD," min  ",DD.D," sec"
```

```
590          RETURN
600            !
610     END
620     !
630     !
640     ! Now the subprograms.
650     !
660     SUB Equip_setup
670          PRINT "Equipment setup."
680          SUBEND
690            !
700     SUB Set_temp
710          PRINT "Oven temperature set."
720          SUBEND
730            !
740     SUB Read_temp
750          PRINT "Temp.= xx degrees F   ";
760          SUBEND
770            !
780     SUB Plot_temp
790          PRINT "(Plotted)."
800          PRINT
810          SUBEND
820            !
830     SUB Off_equip
840          PRINT
850          PRINT "Equipment shut down."
860          PRINT
870          SUBEND
```

At this point, you should run the program to verify that the general program steps are being executed in the desired sequence. If not, keep refining the program flow until all steps are executed in the proper sequence. This is also a very important step of your design process; the sooner you can verify the flow of the main program the better. This approach also relieves you of having to set up and perform the actual experiment as the first test of the program.

Notice also that some of the program steps use CALLs while others use GOSUBs. The general convention used in this example is that subprograms are used only when a program step is to be expanded later. GOSUBs are used when the routine called will probably not need further refinement. As the subprograms are expanded and refined, each can be separately stored and loaded from disc files, as shown in the next step.

Step 4. After the correct order of the steps has been verified, you can begin programming and verifying the details of each step (known as stepwise refinement).

The computer features a mechanism by which the process of expanding each step can be simplified. With it, each subprogram can be expanded and refined individually and then stored separately in a disc file. This facilitates the use of the top-down approach. Each subprogram can also be tested separately, if desired.

In order to use this mechanism, first save or store the main program; for instance, execute SAVE "MAIN1". Then, isolate the subprogram by deleting all other program lines in memory. In this case, executing DEL 10,650 and DEL 700,900 would delete the lines which are not part of the "Equip_setup" subprogram. 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 computer'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 computer. 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 computer 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.

Advanced Transfer Techniques	Chapter
	11

Introduction

While many applications will not need the specialized techniques presented here, these techniques aid in communicating with very slow and very fast devices. The TRANS BIN file must be present in memory.

When using OUTPUT and ENTER to communicate with peripheral devices, special problems can arise. Normally, program execution does not leave the statement until all data items are satisfied; therefore, a very slow device will keep the computer waiting between each byte or word. A great amount of time may be wasted while the computer waits for the device to be ready for the next item.

Another problem exists when communicating with a very fast device. The device may attempt to send data faster than the computer can accept it. To overcome both problems, an alternate method of communication has been implemented — the TRANSFER statement.

The TRANSFER statement allows you to exchange information with a device or file through I/O paths. The most important difference between using TRANSFER and the regular methods of communication (OUTPUT and ENTER) is that a transfer can take place concurrently with continued program execution. Thus a transfer can be thought of as a "background" process or an "overlapped" operation. This has far-reaching consequences that affect the behavior of the computer.

Before any transfer takes place, an area of memory is reserved to hold the data being transferred. This area of memory is called a **buffer**. Defining a buffer is somewhat analogous to creating a high-speed device inside the computer. Two advantages are gained by simulating a device in memory: the buffer is fast enough to accept incoming data from almost any device and the actual transfer operation can be handled concurrently with continued program execution.

Every transfer will use a buffer as either its source or its destination. From the buffer's point of view, there are two types of transfers. An **inbound** transfer moves data from a device or file into the buffer. An **outbound** transfer moves data from the buffer to a device or file.

In addition to the TRANSFER statement, the OUTPUT statement can be used to place data in the buffer and the ENTER statement can be used to remove data from the buffer.

The actual method of transfer is device dependent and is chosen by the computer. The three possible transfer methods are: DMA (direct memory access), FHS (fast handshake), and INT (interrupt).

The ON EOT statement allows you to define a branch to be taken upon the completion of a transfer. When the data being transferred has been divided into records, the ON EOR statement can be used to define a branch to be taken after each record is transferred.

Note

An active TRANSFER will not be terminated by stopping or pausing a program. You may use (RESET) or ABORTIO to terminate a TRANSFER prematurely. The (CLR I/O) (or (Reset)) key will not terminate a TRANSFER.

If a TRANSFER is active while a program is paused, the I/O ($^I{}_□$) indicator is displayed in the lower-right corner of the CRT instead of the pause (-) indicator.

Buffers

A buffer is a section of computer memory reserved to hold the data being transferred. Two types of buffers can be created and assigned to I/O path names. A named buffer is a string scalar, an INTEGER array, or a REAL array. An unnamed buffer is a section of memory which has no associated variable name. Assigning an I/O path name to a buffer creates a control table. This control table defines STATUS and CONTROL registers which can monitor and interact with the operation of the buffer.

All I/O path names, including I/O path names assigned to buffers, use register 0 to indicate the path type.

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

Register 0 returns a 3 when the I/O path is associated with a buffer. Register 1 indicates whether the buffer is named or unnamed.

Status Register 1 – Buffer type (1 = named, 2 = unnamed)

Using Buffers

Creating buffers

Named buffers are buffers which use variables declared in DIM, COM, REAL, or INTEGER statements. Note that a buffer cannot be allocated by an ALLOCATE statement. Named buffers are declared by placing the keyword BUFFER after the variable name. For instance:

```
DIM A$[256],B$[256] BUFFER,C$
```

```
COM Block(1000),Temp(100) BUFFER,INTEGER X(10,10) BUFFER,Y,Z
```

```
REAL Fools_buff(1000), Real_buff(10) BUFFER, No_buff(10)
```

Only the variable name immediately preceding the keyword BUFFER becomes a buffer. In the first example statement, B$ is a buffer while A$ and C$ are not buffers. Declaring a variable as a buffer does not prevent it from being used in its normal manner, but care must be taken not to corrupt the information in the buffer.

Assigning Buffers

Once a named buffer has been declared, an I/O path name can be assigned to it by an ASSIGN statement. For instance:

```
ASSIGN @Path TO BUFFER B$
```

```
ASSIGN @Buff TO BUFFER X(*)
```

```
ASSIGN @Buffer TO BUFFER Real_buff(*)
```

The I/O path name can now be used to access the buffer. The keyword BUFFER must appear in both the variable declaration statement and the ASSIGN statement for named buffers.

Unnamed buffers are created in ASSIGN statements and can only be accessed by their I/O path names. Using unnamed buffers ensures data integrity since the buffer cannot be accessed by a variable name. Closing (ASSIGN @Path TO *) an I/O path assigned to an unnamed buffer releases the memory reserved for the buffer. This is similar to the behavior of allocated variables. The following statement shows a typical unnamed buffer assignment.

```
ASSIGN @Buff to BUFFER [65536]
```

The value in brackets indicates the number of bytes of memory to be reserved for the buffer. This allows a buffer to be larger than the maximum length of 32 767 bytes for a string variable. Named buffers using REAL and INTEGER arrays can also be larger than 32 767 bytes.

Once a buffer has been assigned an I/O path name, Status register 2 returns the buffer's capacity (maximum size, in bytes).

Status Register 2 — buffer size in bytes

When I/O path names are assigned to buffers, the buffer must exist as long as the I/O path name is valid. Consider the example of a buffer created locally in a context and then assigned an I/O path name declared in COM. When execution leaves the local context, the I/O path name would still be valid but the buffer would no longer exist. If this happens, an error is reported:

```
ERROR 602 Improper BUFFER lifetime.
```

This error also occurs if the buffer and the I/O path name being assigned are in different COM areas.

Buffer Pointers

In order to understand I/O involving buffers, it is essential to understand how a buffer is set up and maintained.

When an ASSIGN statement associates an I/O path name with a buffer, it also creates and initializes a buffer control table. Among the entries in the control table are two pointers and a counter which are used to monitor and control all data transfer to and from the buffer through the I/O path. The buffer **fill pointer** points to the next byte of the buffer which can accept data. The **empty pointer** points to the next byte of data which can be read from the buffer. When the ASSIGN is performed, both of these pointers are set to the first byte of buffer storage, and the counter is set to 0 to signify an empty buffer.

The current values of the pointers can be checked by using the STATUS statement with the following registers.

Status Register 3 Current fill pointer

Status Register 4 Current number of bytes in buffer

Status Register 5 Current empty pointer

As data is written into the buffer (OUTPUT or TRANSFER), the fill pointer is advanced as necessary to point to the next available byte of buffer storage, and the counter is incremented by the number of bytes added to the buffer. Similarly, when data is read from the buffer (ENTER or TRANSFER), the empty pointer is advanced to point to the first unread byte, and the counter is decremented by the number of bytes which have been read.

(inbound)

```
TRANSFER @Device TO @Buffer
```

```
                                        ↓ fill pointer
_____
....ata   data   data   data   data   data   data   data   data   d...............
_____
 ↑  empty pointer
```
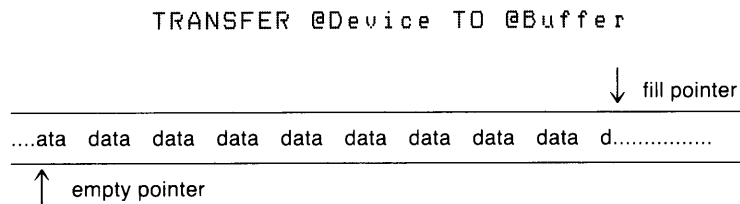
(outbound)

```
TRANSFER @Buffer to @File
```

It is also important to realize that the buffers used with the TRANSFER statement are **circular**. This means that when the last byte of buffer storage has been accessed, the system will wrap around and access the first byte of buffer storage. The only thing which prevents writing more data into the buffer is the byte count (Register 4) to become equal to the buffer capacity (Register 2). Similarly, once the system has read the data from the last byte of buffer storage, it will next read from the first byte, but reading must cease when the byte count reaches zero.

Both full and empty buffers have the fill pointer and the empty pointer referencing the same byte of buffer storage. The system distinguishes between full and empty by examining the byte count. If it is zero, the buffer is empty. If it is equal to the buffer's capacity, the buffer is full.

It is impossible to perform any operation which would cause the byte count to take on a value less than zero or greater than the buffer capacity. Attempting to OUTPUT more data into a full buffer or ENTER data from an empty buffer produces:

```
ERROR 59 End of file or buffer found
```

Since fill and empty pointers are updated independently of each other and a TRANSFER can execute concurrently with other statements, it is possible for one TRANSFER to be putting data into the buffer while another TRANSFER is removing data.

The amount of data which can be moved by a single transfer operation is not limited by the buffer's capacity. When two TRANSFER statements involving the same buffer are of comparable speed and execute concurrently, the buffer's fill and empty pointers may never reach the empty or full state. If the two TRANSFER statements execute at different speeds because of the transfer mode which must be used or because of the throughput capacity of the devices involved, it is still possible to keep two TRANSFER statements running concurrently by specifying the CONT parameter on both. CONT directs a transfer not to terminate when the buffer becomes full or empty. Instead, the transfer "goes to sleep" until the buffer is again ready for the transfer process to continue.

Transfers

Once a buffer has been created and an I/O path name assigned to it, data can be transferred into or out of the buffer by a TRANSFER statement. Every TRANSFER will need a buffer as either its source or destination. For example:

```
TRANSFER @Source TO @Buffer
```

or

```
TRANSFER @Buffer TO @Destination
```

From the buffer's point of view, there are two types of transfers; inbound and outbound. An inbound transfer will move data from a device or file into the buffer, updating a fill pointer and byte count as it proceeds. An outbound transfer will remove data from the buffer, updating an empty pointer and byte count as necessary. For a complete explanation, see the Anatomy of a Buffer section at the end of this chapter.

The OUTPUT and ENTER statements may be used to interact with the data sent through the buffer. If the I/O path name of the buffer is used as the source for an ENTER or the destination for an OUTPUT, the control table will be updated automatically. Accessing the data in a named buffer by using the variable name will not update the buffer pointers. This could easily lead to corruption of the data in the buffer.

I/O path names used with TRANSFER are restricted to external devices, BDAT files, or buffers. A transfer cannot involve the CRT, the keyboard, the BCD interface, or the tape backup of the CS80 disc drives. One and only one buffer can be specified in a TRANSFER statement. Transfers from buffer to buffer or from device to device are not allowed.

OUTPUT and ENTER statements can format data according to a given IMAGE list and transform the data according to the attributes specified in the ASSIGN statement. No data formatting or transformation occurs, however, when data are transferred by a TRANSFER statement.

Using Transfers

TRANSFER statements are generally used in one of three catagories: sending data to a file or device (outbound), receiving data from a file or device (inbound), and data logging (inbound and outbound).
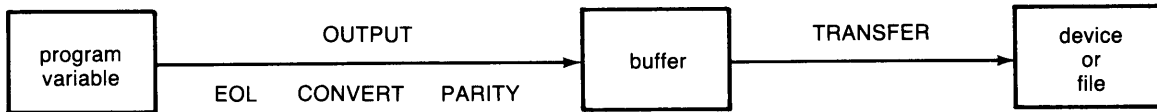
Outbound Transfer
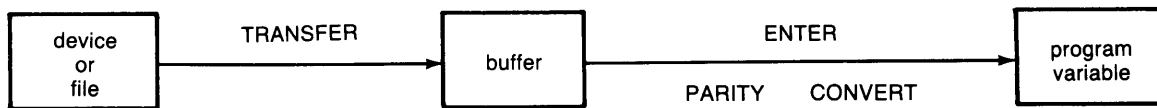
Inbound Transfer
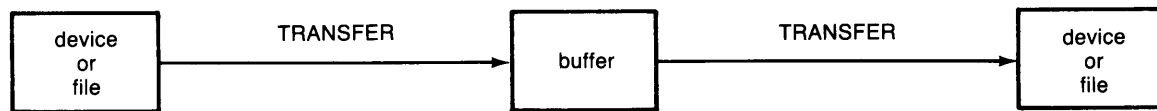
Data Logging

TRANSFER operations are allowed for all supported mass storage devices.

Initiating a Transfer

Several steps are necessary before a transfer can take place. For instance, a named buffer would need a declaration and an assignment to an I/O path name. A string variable is used here:

```
10    DIM Text$[1025] BUFFER
20    ASSIGN @Buff TO BUFFER Text$
```

Note that the keyword BUFFER must appear in both declaration and ASSIGN statements. For an unnamed buffer only an ASSIGN statement would be necessary. For example:

```
ASSIGN @Buff TO BUFFER[1025]
```

The next step is to assign path names to the devices used in the transfer. For example, the following statement will assign an I/O path name to the printer (PRT is a function, which returns device selector 701).

```
30    ASSIGN @Print TO PRT
```

The printer can now be used as the destination device. For the purposes of this example, it is convenient to use an OUTPUT statement to fill the buffer instead of assigning a device as the source.

```
50    FOR I=1 TO 25
60      OUTPUT @Buff;"How many times do I need to print this?"
70    NEXT I
```

By using the I/O path name with OUTPUT, the buffer is filled and the fill pointer is updated. This is not the same as modifying the variable, which does not change the buffer pointers.

Now that the necessary steps have been taken, the TRANSFER can begin.

```
90    TRANSFER @Buff TO @Print
```

Here is the complete listing of the program.

```
10     DIM Text$[1025] BUFFER
20     ASSIGN @Buff TO BUFFER Text$
30     ASSIGN @Print TO PRT              ! 'PRT' returns 701 for printer
40     !
50     FOR I=1 TO 25
60       OUTPUT @Buff;"How many times do I need to print this?"
70     NEXT I
80     !
90     TRANSFER @Buff TO @Print          ! Start the transfer
100    !
110    FOR I=1 TO 450
120      PRINT TABXY(I MOD 15,0);"As many times as it takes."
130    NEXT I
140    END
```

In the program, lines 10 and 20 create a named buffer. Line 30 assigns a printer that will be used as the destination for the transfer. The OUTPUT statement in line 60 fills the buffer with data. Line 90 contains the TRANSFER statement that sends the data in the buffer to the printer. Running the program shows the overlapped operation of transfers. Buffered data is being printed on the printer while the program prints on the CRT.

A similar technique can be used for inbound transfers.

```
10      DIM Text$[256] BUFFER,A$(100)[80]
20      ASSIGN @Buff TO BUFFER Text$
30      ASSIGN @Device TO 12              ! Some device at select code 12
40      !
50      TRANSFER @Device TO @Buff;CONT    ! Start the transfer
60      !
70      FOR I=1 TO 100
80         ENTER @Buff;A$(I)              ! Enter the items
90      NEXT I
100     ABORTIO @Device                   ! Terminate TRANSFER
110     !
120     END
```

A named buffer is created in lines 10 and 20. A device is assigned in line 30 that will be used as the source for the transfer. The buffer is filled by the TRANSFER in line 50 and the ENTER statement in line 80 empties the buffer.

Choosing Transfer Parameters

For a standard inbound transfer, data from the device (or file) is placed in the buffer and the TRANSFER is deactivated when the buffer is full. For an outbound transfer, all data is removed from the buffer and the TRANSFER is deactivated when the buffer is empty.

To allow a TRANSFER to continue indefinitely, the CONT parameter can be specified.

```
TRANSFER @Source TO @Buffer;CONT
```

Several interesting things happen when a continuous TRANSFER is specified. Execution cannot leave the current program context unless the buffer and I/O path name are in COM (or passed as parameters), and you will not be able to LOAD, GET, or EDIT a program. During program development, you can terminate a transfer by (RESET) (or (Shift) (Break)) or ABORTIO @Non_buff (use the I/O path name assigned to either the device or file). ABORTIO can be used in a program or executed from the keyboard.

A continuous TRANSFER can also be canceled by writing to a CONTROL register (use the I/O path name assigned to the buffer). Note that the CONTROL register only cancels the continuous mode. The TRANSFER is still active until the buffer is full or empty.

```
CONTROL @Buff,8;0    for inbound transfers

CONTROL @Buff,9;0    for outbound transfers
```

When the CONT parameter is specified for an inbound transfer, the transfer fills the buffer and is then suspended while program execution continues. The suspended transfer "sleeps" until another operation removes some data from the buffer. The transfer then "wakes up" and continues the transfer operation. When the CONT parameter is specified for an outbound transfer, the transfer empties the buffer and is then suspended. As soon as more data are available, the transfer "wakes up" and continues the transfer operation. This process proceeds until the transfer is completed or the CONT mode is canceled.

By default, transfers take place concurrently with continued program execution. To defer program execution until a transfer is complete, use the WAIT parameter. This allows transfers to take place serially (non-overlapped).

```
TRANSFER @Source TO @Buffer;WAIT
```

When the WAIT parameter is specified, the program statement following the TRANSFER will not be executed until the transfer has completed. By combining both the CONT and WAIT parameters, a continuous serial TRANSFER can be defined. However, this is only legal if you already have an active TRANSFER for the buffer in the opposite direction.

```
TRANSFER @Source TO @Buffer;WAIT,CONT
```

The COUNT parameter tells a transfer how many bytes are to be transferred. The following TRANSFER specifies 32 bytes to be transferred. The transfer will terminate after 32 bytes have been transferred (or when the buffer becomes full for non-continuous transfers).

```
TRANSFER @Source TO @Buffer;COUNT 32
```

The DELIM parameter can be used to terminate an inbound transfer when a specified character is received. The following TRANSFER will terminate when the delimiter (comma) is sent or when the buffer is full (unless the CONT parameter is specified). The DELIM parameter is not allowed on outbound transfers or WORD transfers. If the DELIM string is the null string, the DELIM clause is ignored. This allows programmatic disabling of DELIM checking. An error results if the DELIM string contains more than one character.

```
TRANSFER @Source TO @Buffer;DELIM ","
```

The END parameter can also be used to terminate a TRANSFER. The END parameter is discussed in detail following the introduction of the RECORDS parameter.

```
TRANSFER @Source TO @Buffer;END
```

It is often desirable to divide the data into records. The RECORDS parameter is then used to indicate the size of each record.

Whenever RECORDS is used, there must be a parameter which signals the end of a record. The EOR (End-Of-Record) parameter can use COUNT, DELIM, or END (discussed later) to signify the end of a record. For example, the following statement specifies 4 records of 15 bytes per record are to be transferred.

```
TRANSFER @Source TO @Buffer;RECORDS 4,EOR(COUNT 15)
```

When multiple termination conditions are specified, the transfer will terminate when any one of the conditions occurs.

```
TRANSFER @Source TO @Buffer;COUNT 128,DELIM ";",END
TRANSFER @Source TO @Buffer;RECORDS 100,EOR(COUNT 15,END)
```

As in all transfer operations, unless the CONT parameter is specified, the TRANSFER will also terminate when the buffer is full or empty.

The END parameter specifies an inbound transfer will be terminated by receiving an interface-dependent signal (for devices) or by encountering the current end-of-file (for files). Some devices on the HP-IB send an EOI concurrently with the last byte of data. Unless the END parameter is specified, receiving an EOI will generate an error. For files, encountering the end-of-file will generate an error unless the END parameter is specified.

Using the END parameter with an outbound transfer on the HP-IB will result in the EOI signal being sent concurrently with the last byte of the transfer. If EOR(END) is specified, EOI will be sent with the last byte of each record. For files, END will cause the end-of-file pointer to be updated at the end of the transfer. Using EOR(END) will cause the pointer to be updated at the end of each record.

The following tables show the different system responses to the END and EOR(END) parameters.

Inbound TRANSFER

	File	Device
No END	Terminate prematurely. Bit 3 of Register 10 is set. Error 59 waiting.	Terminate prematurely. Bit 3 of Register 10 is set. Error 59 waiting.
END	Terminate normally. Bit 3 of Register 10 is set.	Terminate normally. Bit 3 of Register 10 is set.
EOR(END)	Terminate normally. Bit 3 of Register 10 is set.	Finish current record. ON EOR triggered. Start new record.
END,EOR(END)	Terminate normally. Bit 3 of Register 10 is set.	Terminate normally. Bit 3 of Register 10 is set.

An error is logged when a transfer terminates prematurely. For overlapped transfers, this error is "waiting" and will be reported the next time the non-buffer I/O path name is referenced. At that time, any ON ERROR or ON TIMEOUT branches will be triggered. (If the WAIT parameter is specified, the error is reported immediately.) See "Error Reporting" for further explanation.

An ON END branch will be triggered only if the END parameter is not specified.

Outbound TRANSFER

	File	Device
No END	No special action.	No special action.
END	Update EOF pointer after TRANSFER is finished.	Send an EOI with the last byte of TRANSFER.
EOR(END)	Update EOF pointer after each record.	Send an EOI with the last byte of each record.
END,EOR(END)	Update EOF pointer after each record and when the TRANSFER is finished.	Send an EOI with the last byte of each record and with the last byte of the TRANSFER.

For an outbound transfer to a device, no special action is taken if the device does not support EOI. The Serial, Datacomm and GPIO interfaces do not support EOI.

Branching

Two types of event-initiated branches can be defined for a transfer. The ON EOT statement defines and enables a branch to be taken upon completion of a transfer. The ON EOR statement defines and enables a branch to be taken every time a record is transferred.

```
ON EOT @Device CALL Process
ON EOR @File GOTO Parse
```

No ON EOR branches will be triggered unless the EOR parameter is specified in the TRANSFER statement and an item is transferred which satisfies one of the end-of-record conditions (COUNT, DELIM, or END).

To ensure that a branch receives service, the transfer must complete before attempting to leave the context in which the branches are defined. If the I/O path names are local to a program context, encountering SUBEND, SUBEXIT, or RETURN before the transfer has completed will cause the context switch to be deferred until completion of the transfer. If this happens, any ON EOR or ON EOT branch will not be serviced.

Certain statements wait until a transfer is completed before they are executed. A complete list of these statements is provided later in this chapter. These statements can be used to prevent overlapped operation or defer a context switch until completion of the transfer. For example, if the following I/O path names were used in a TRANSFER, either of the following statements will cause program execution to wait until the transfer is finished.

```
ASSIGN @Path TO *        (can be a device, file, or buffer)

WAIT FOR EOT @Non_buff (can be a device or file)
```

When a TRANSFER is used inside a loop, the entire loop may execute before the transfer has completed. If this happens, the second execution of the TRANSFER statement will wait until the completion of the first. Any event-initiated branch defined for the TRANSFER (ON EOT or ON EOR) will be serviced.

While the WAIT parameter can be specified to ensure completion of a transfer before proceeding with the next statement (thus ensuring a branch can be serviced), this defeats any advantage of overlapped operation.

The WAIT FOR statement can be used to allow overlapped operation up to the point where the WAIT FOR statement is encountered. The WAIT FOR statement ensures the servicing of an event-initiated branch defined for the end-of-transfer or end-of-record.

Terminating a Transfer

A transfer is usually terminated by satisfying the conditions specified by the transfer parameters. There are times, especially during program development, when you may wish to prematurely terminate (abort) a transfer.

A transfer can be aborted by pressing the (**RESET**) (or (**Shift**) (**Break**)) key. Pressing (**RESET**) will stop the program, close all I/O paths, and destroy all buffer pointers. To abort a transfer without stopping the program, the ABORTIO statement can be used from the program or the keyboard. For example:

```
ABORTIO @Non_buff
```

This statement will terminate any active transfer associated with the I/O path. ABORTIO has no effect if a transfer is not in progress. Using ABORTIO does not ensure all data in the buffer is transferred, but it does leave the buffer pointers and byte count in their correct state.

Note

If the destination of a TRANSFER is a mass storage file, aborting a TRANSFER with ABORTIO will not cause data already placed in the disc buffer to be written to the disc. Up to 255 bytes of data could be lost.

While most transfers are terminated by fulfilling the conditions specified by the parameters, a continuous TRANSFER (using the CONT parameter) requires a bit more effort to terminate.

To terminate a continuous TRANSFER without leaving data in the buffer, first cancel the continuous mode (with CONTROL), then wait for the transfer to complete. Use register 8 for inbound transfers and register 9 for outbound transfers. The following two methods are the safest ways of terminating a continuous TRANSFER.

```
CONTROL @Buff,8;0
WAIT FOR EOT @Path

CONTROL @Buff,8;0
ASSIGN @Path TO *
```

Remember that the buffer pointers are not reset to the beginning of the buffer when the transfer is finished. The RESET statement (RESET @Buff) can be used to reset the buffer pointers to the beginning of the buffer and the byte count to zero.

Transfers are not terminated by pausing the program. The I/O indicator in the lower-right corner of the CRT will indicate when a transfer is in progress.

While transfers may continue when the computer is in the paused state, all transfers must terminate before entering the stopped state. Pressing (ENTER), after editing or adding a program line, will attempt to put the computer in the stopped state. If a transfer is still in progress, the computer will "hang" until the transfer is completed. To abort the transfer without performing a hardware reset, press (CLR I/O) (or (Break)) to clear the (ENTER) and then ABORTIO the non-buffer I/O path name for each active TRANSFER. If a hardware reset can be tolerated, press (RESET) (or (Shift) (Break)) to terminate the transfer.

Transfer Examples

Here is a short program which sets up a continuous transfer from a device through the buffer to a BDAT file. A program of this type is useful when the data being received must be saved for later analysis.

```
10    ! Data Logging Example
20    !
30    ! Buffer size should be a multiple of disc sector (256)
40    ASSIGN @Device TO 717              ! Assign source device on HPIB
50    ASSIGN @Buf TO BUFFER [512]        ! Assign BUFFER
60    ASSIGN @File TO "LOG_FILE"         ! Assign destination file
70    !
80    TRANSFER @Device TO @Buf;CONT      ! Continuous TRANSFER
90    TRANSFER @Buf TO @File;CONT        ! Continuous TRANSFER
100   !
110   ! Program execution continues ...
120   ! Data logging continues as a "background" task ...
130   !
140   PAUSE                    ! TRANSFER continues in paused state
150   END
```

The following program creates a BDAT file and then sends it to a printer. Notice that the OUTPUT statement used to fill the file placed a CR/LF at the end of each record. The TRANSFER statement (line 90) looks for the carriage-return as a record delimiter.

```
10    ON ERROR CALL Makefile
20    ASSIGN @File TO "BDAT_FILE"      ! Test for file's existence
30    OFF ERROR
40    ASSIGN @Buff TO BUFFER [2046]    ! Assign buffer
50    ASSIGN @Print TO PRT             ! Assign destination
60    !
70    Cr$=CHR$(13)                     ! ASCII character for carriage return
80    PRINT "Start"
90    TRANSFER @File TO @Buff;RECORDS 10,END,EOR (DELIM Cr$)
100   !
110   TRANSFER @Buff TO @Print
120   FOR I=1 TO 10000
130     PRINT "TRANSFERS RUNNING",I
140     STATUS @Buff,11;Stat
150     IF NOT BIT(Stat,6) THEN 180
160   NEXT I
170   !
180   OUTPUT @Print;CHR$(12)           ! ASCII character for formfeed
190   PRINT "File is printed"
200   END
210   !
220   SUB Makefile
230     OFF ERROR
240     CREATE BDAT "BDAT_FILE",10,12
250     ASSIGN @File TO "BDAT_FILE";FORMAT ON
260     FOR I=1 TO 10
270       DISP "Writing";I
280       READ Word$
290       OUTPUT @File;Word$
300     NEXT I
310     DISP
320     DATA ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,TEN
330   SUBEND
```

The next program continually shows the activity of the buffer. Note that a continuous TRANS-FER is used (line 90). Data is placed in the buffer a few bytes at a time (line 130) and the status is displayed by the SUB called from line 140. After a few hundred bytes are transferred, the continuous mode is canceled (line 180), the program waits for the transfer to finish (line 190), and the final status is displayed.

```
10    !                      "SHOW_BUFF"
20    PRINTER IS CRT
30    PRINT USING "@"              ! Clear Screen
40    COM @Buff,@Print,B$[47] BUFFER    ! Declare variables
50    INTEGER Characters
60    ASSIGN @Buff TO BUFFER B$         ! Assign I/O path name to buffer
70    ASSIGN @Print TO PRT              ! Assign I/O path name to 701
80    DISP "printer is off line"        ! Transfer hangs if no printer
90    TRANSFER @Buff TO @Print;CONT     ! Continuous transfer
100   DISP                             ! Clear display line
110   !
120   REPEAT                           ! Fill buffer with data
130      OUTPUT @Buff;"AB ";
140      CALL Buff_status
150      Times=Times+1
160   UNTIL Times>100
170   !
180   CONTROL @Buff,9;0                ! Cancel continuous mode
190   WAIT FOR EOT @Print              ! Wait for buffer empty
200   CALL Buff_status                 ! Show final status
210   END
220   !---------------------------------------------------
230   SUB Buff_status
240      COM @Buff,@Print,B$ BUFFER
250      STATUS @Buff;R0
260      PRINT TABXY(1,1);"Buffer Status: ";
270      STATUS @Buff,1;R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13
280      IF R1=1 THEN PRINT "Named ";
290      IF R1=2 THEN PRINT "Unnamed ";
300      PRINT "Buffer[";VAL$(R2);"]"
310      PRINT TABXY(1,3);RPT$(" ",55)
320      PRINT TABXY(R3,3);"v"       ! Show fill pointer position
330      PRINT TABXY(1,4);"""";B$;""""  ! Show buffer contents
340      PRINT TABXY(1,5);RPT$(" ",55)
350      PRINT TABXY(R5,5);"^"       ! Show empty pointer position
360      PRINT
370      PRINT "Fill pointer:   ";R3
380      PRINT "Bytes in use:   ";R4
390      PRINT "Empty pointer:  ";R5
400      PRINT
410      PRINT "           inbound/outbound"
420      PRINT "Select code:    ";R6;"/";R7
430      PRINT "Continuous? :   ";R8;"/";R9
440      PRINT "Term. status:   ";R10;"/";R11
450      PRINT "Total bytes:    ";R12;"/";R13
460   SUBEND
```

Data currently in the buffer can be reused or ignored by manipulating the pointers (with CON-TROL). When it is neccessary to move data through the buffer without using I/O path names, the CONTROL statement can be used to modify the pointers, thus allowing a TRANSFER to take place. The next program uses this technique. The array size used in the next program is for the Model 236; change the array size in lines 50 and 60 for the Model 226 and Model 216. You must load the GRAPH BIN file to create and execute the program.

```
10     GINIT                               ! Uses graphics
20     GCLEAR
30     GRAPHICS ON
40     PRINT CHR$(12)                      ! Clear the screen
50     INTEGER I,Graph(1:12480) BUFFER     ! (1:7500) FOR Model 226,Model 216
60     Gbytes=2*12480                      ! 2 * 7500 FOR Model 226,Model 216
70     ASSIGN @Buff TO BUFFER Graph(*)
80     ON ERROR GOTO Record                ! Enable ERROR trap
90     ASSIGN @Read TO "PHOTOS"            ! Test if file exists
100    ASSIGN @Read TO *                   ! Close file
110    GOTO Playback                       ! If file exists then Playback
120                                        !
130 Record:OFF ERROR
140    CREATE BDAT "PHOTOS",5,Gbytes       ! Five "PHOTOS" of graphics screen
150    ASSIGN @Write TO "PHOTOS"           !    to be written to the BDAT file
160    FOR I=1 TO 5
170       GRID I*4,I*4
180       GSTORE Graph(*)                  ! Fill buffer with GSTORE
190       GCLEAR
200       DISP "SAVING #";I
210       CONTROL @Buff,4;Gbytes           ! Tell TRANSFER "The buffer is full"
220       TRANSFER @Buff TO @Write;WAIT
230    NEXT I
240    ASSIGN @Write TO *
250    !
260 Playback:OFF ERROR
270    ASSIGN @Read TO "PHOTOS"
280    FOR I=1 TO 5
290       DISP "LOADING #";I
300       TRANSFER @Read TO @Buff;WAIT
310       GLOAD Graph(*)
320       CONTROL @Buff,4;0                ! Tell TRANSFER "The buffer is empty"
330    NEXT I
340    DISP "DONE"
350    END
```

The program creates five "photos" of the graphics raster and writes them to a disc file. The file is then read and each picture is loaded back into the graphics raster.

Special Considerations

Transfer with Care

Whenever possible, a transfer will take place concurrently with continued program execution. You must carefully construct a program using transfers. A poorly designed transfer may take longer to execute than using OUTPUT and ENTER.

A TRANSFER which uses a local I/O path name must terminate before a SUBEXIT, SUBEND, or RETURN (from a function) can return execution to the calling context. The system will detect that such a transfer is in progress and will make the SUBEXIT wait for the transfer to terminate. If this happens, the system will not process any ON EOT (or ON EOR) branch which had been defined for the transfer. To allow servicing of the branch, any statement which cannot execute in overlap with the TRANSFER can be inserted in the subprogram before the SUBEXIT. Two of the most sensible choices are WAIT FOR EOT @Non_buff or ASSIGN @Path to *.

A TRANSFER which uses only non-local I/O path names can execute in overlap with a SUBEXIT. One word of caution is necessary: if a local ON EOT (or ON EOR) statement is used in the subprogram, its branch will not be serviced if the SUBEXIT is encountered before termination of the TRANSFER. To ensure the possibility of servicing the branch, insert a statement that cannot execute in overlap with the TRANSFER. This is essentially the same technique discussed in the preceding paragraph.

More than one I/O path name can be assigned to a named buffer; however, each path name will maintain its own set of pointers. Using multiple path names on the same buffer could lead to corruption of the data in the buffer.

Special care should be taken when using REAL arrays as buffers since a device may send a bit pattern that is not a valid real number. Accessing the data as a REAL value may produce an error.

Statements Which Affect Concurrency

The following statements do **not** wait for the completion of a TRANSFER statement.

Buffer in use	Device in use
STATUS @Buf	STATUS @Dev
CONTROL @Buf	ON EOR @Dev
SCRATCH A	ON EOT @Dev
	OFF EOR @Dev
	OFF EOT @Dev

Statements which wait for completion of inbound transfers.

```
OUTPUT @Buf
TRANSFER @Dev TO @Buf
```

Statements which wait for completion of outbound transfers.

```
ENTER @Buf
TRANSFER @Buf TO @Dev
```

Statements which wait for completion of inbound and outbound transfers.

Buffer in use	Device in use
`ASSIGN @Buf TO *`	`ASSIGN @Dev TO *`
`ASSIGN @Buf TO BUFFER[bytes]`	
`ASSIGN @Buf TO BUFFER B$`	
`ASSIGN @Dev`	`ASSIGN @Dev`
`ASSIGN @Dev; (new attributes)`	`ASSIGN @Dev; (new attributes)`
	`WAIT FOR EOT @Dev`
	`OUTPUT @Dev`
	`ENTER @Dev`
	`TRANSFER @Buf TO @Dev`
	`TRANSFER @Dev TO @Buf`
`END`	`END`
`SUBEXIT`	`SUBEXIT`
`SUBEND`	`SUBEND`
`SCRATCH C`	`SCRATCH C`
`SCRATCH`	`SCRATCH`
`LOAD "PROG"`	`LOAD "PROG"`
`GET "PROG"`	`GET "PROG"`
`STOP`	`STOP`
	`CONTROL @Dev`

Error Reporting

If an error is encountered during an overlapped transfer, the error is logged in the non-buffer I/O path name and reported the next time the non-buffer I/O path name is referenced. Thus, the error line reported will be the most recently executed line containing the I/O path name and usually not the line containing the TRANSFER statement. For example:

```
10      !   This program shows delayed error reporting for TRANSFER
20      !
30      ON ERROR GOTO Ok
40      PURGE "bdat_file"                      ! Zap file if it already exists
50  Ok:OFF ERROR
60      !
70      CREATE BDAT "bdat_file",1              ! CREATE an empty file
80      ASSIGN @Non_buf TO "bdat_file"         ! ASSIGN I/O path name to the file
90      INTEGER B(100) BUFFER                  ! Declare a variable as a buffer
100     ASSIGN @Buf TO BUFFER B(*)             ! Assign I/O path name to buffer
110     PRINT
120     !
130     WAIT 2
140     LIST 150,150
150     TRANSFER @Non_buf TO @Buf;CONT          ! Error occurs in this line
160     !
170     WAIT 2
180     LIST 190,190
190     STATUS @Buf,10;Status_byte              ! Error not reported with @Buf
200     !
210     WAIT 2
220     LIST 230,230
230     STATUS @Non_buf;Status_byte            ! Error reported with @Non_buf
240     END
```

Since a continuous TRANSFER was specified, the error that occurs in line 150 is reported in line 230 when the non_buffer I/O path name is referenced. For continuous transfers, the error is always logged with the non-buffer I/O path name. Referencing the buffer's I/O path name (line 190) does not cause the error to be reported. After running the program, change the CONT parameter in line 150 to WAIT. The program will now report the error in line 150 since the WAIT parameter specified a serial TRANSFER.

At the time the error is reported, any ON END (for files), ON TIMEOUT (for devices), or ON ERROR statements will be triggered. However, ON END is not triggered when the END parameter is specified.

Suspended Transfers

When a TRANSFER statement is executed, that transfer is said to be "active". The transfer proceeds until either a termination condition is reached, or until there is nothing else the transfer can do for the time being. An example of the latter is a continuous TRANSFER, which does not terminate when the buffer is full and has not yet met any other termination conditions.

This TRANSFER will be "suspended" to give some other TRANSFER operation a chance to empty the buffer. It will not be reactivated until one of the following occurs:

1. The other TRANSFER operation reaches a record boundary, fills or empties the buffer, terminates, or is suspended.

2. An OUTPUT or ENTER operation active in the other direction fills or empties the buffer, or terminates.

3. A CONTROL statement is executed to change the fill or empty pointers, or buffer's byte count.

4. A CONTROL statement is executed to cancel continuous mode.

A TRANSFER cannot be suspended unless it has CONT as one of its transfer parameters.

Transfer Performance

For the best performance when transferring files, the buffer size should be a multiple of 256 bytes (the size of a disc sector). If the buffer is not a multiple of 256 bytes, the system must do sector buffering; this is handled automatically, but reduces the transfer rate.

While a TRANSFER can be assigned to the internal disc drives in the Model 226 and Model 236, no noticeable increase in speed (compared to OUTPUT or ENTER) will result. Transfers to and from external mass storage (except the 9885) will show an increase in speed especially if a DMA card is present.

Some of the discs are capable of overlapped operation. This means that other processing can occur while a non-continuous TRANSFER to or from the disc is taking place. In other words, the program can execute other statements before the transfer has completed. Overlapped discs include the CS80 discs, the HP9895, the HP9121, the HP9133, the HP9134, the HP9135, the HP82901, and the HP82902.

Discs which are not capable of overlapped operation are called serial discs. When executing a non-continuous TRANSFER to or from a serial disc, the program will not leave the TRANSFER statement until it completes. Serial discs include the internal discs and the HP9885.

The following example illustrates the difference between a serial disc and an overlapped disc.

```
10      OPTION BASE 1
20      INTEGER B(128,10)                 ! A 10-sector buffer
30      LINPUT "Enter msus:",Msus$
40      CREATE BDAT "bdat"&Msus$,10
50      ASSIGN @File TO "bdat"&Msus$
60      ASSIGN @Buffer TO BUFFER [2560];FORMAT OFF
70      OUTPUT @Buffer;B(*)               ! Fill @Buffer's buffer with 10 sectors
80      ON EOT @File GOTO Serial_eot      ! Branch taken if TRANSFER is serial
90      TRANSFER @Buffer TO @File
100     ON EOT @File GOTO Overlapped_eot  ! Branch taken if TRANSFER is overlapped
110     LOOP
120        I=I+1
130        PRINT I,"OVERLAPPED"
140     END LOOP
150 Serial_eot:   !
160     PRINT "SERIAL"
170 Overlapped_eot:  !
180     ASSIGN @File TO *
190     PURGE "bdat"&Msus$
200     END
```

If this program is used with a serial disc, the program stays in the TRANSFER statement until the transfer is complete. Upon completion of the transfer, the ON EOT branch to Serial_eot is taken.

If this program is used with an overlapped disc, the TRANSFER statement begins the transfer, but the program executes the next statement before the transfer completes. In this program, the next statement changes the ON EOT branch. During the transfer, a count and the word "OVERLAPPED" are printed. When the transfer is complete, the ON EOT branch to Overlapped_eot is taken.

If the CONT parameter is specified for a TRANSFER with a serial disc, the transfer appears overlapped because the program executes any statements which follow the TRANSFER statement before the transfer terminates. Here is what really happens in this case. The transfer proceeds until the buffer is full (for inbound transfers) or empty (for outbound transfers). The transfer is then suspended because CONT was specified. The TRANSFER statement is exited and the next statement is executed. The transfer will remain suspended until the continuous mode is terminated or until the buffer is filled (for inbound transfers) or until the buffer is emptied (for outbound transfers). If there is a second TRANSFER active for the buffer, an EOR or EOT condition for the second TRANSFER can also wake up the suspended TRANSFER.

In contrast to serial discs, overlapped discs would allow the statement following the TRANSFER to execute before the buffer was full or empty.

The following program illustrates a transfer to a serial device which appears overlapped.

```
10      OPTION BASE 1
20      INTEGER B(128,10)                    ! A 10-sector buffer
30      LINPUT "Enter Overlapped msus:",Overlapped$
40      CREATE BDAT "bdat"&Overlapped$,10
50      LINPUT "Enter Serial msus:",Serial$
60      CREATE BDAT "bdat"&Serial$,10
70      ASSIGN @Overlapped TO "bdat"&Overlapped$
80      OUTPUT @Overlapped;B(*)
90      RESET @Overlapped                    ! Position to beginning
100     ASSIGN @Buffer TO BUFFER [512];FORMAT OFF
110     ASSIGN @Serial TO "bdat"&Serial$
120     ON EOT @Overlapped GOTO Eof
130     TRANSFER @Overlapped TO @Buffer;END,CONT
140     TRANSFER @Buffer TO @Serial;CONT
150     LOOP
160        I=I+1
170        PRINT I,"OVERLAPPED"
180     END LOOP
190 Eof:   !
200     CONTROL @Buffer,9;0
210     ASSIGN @Overlapped TO *
220     PURGE "bdat"&Overlapped$
230     ASSIGN @Serial TO *
240     PURGE "bdat"&Serial$
250     END
```

In this example, an overlapped disc is used to fill the buffer while a serial disc empties the buffer. Any overlapped device could have been used. After both TRANSFER statements are executed, the program prints the count and the word "OVERLAPPED" while reading from one disc and writing to the other disc. The inbound transfer is terminated when it encounters the end of the file. The outbound transfer is terminated when the CONTROL statement cancels the CONT mode.

Transfer Method

All transfers use DMA mode whenever possible. However, any one of the following reasons will prevent a DMA transfer.

- The DMA card is not present
- Both DMA channels are busy
- The device involved is not HP-IB or GPIO
- The DELIM parameter is specified

If DMA cannot be used with the HP-IB or GPIO interfaces, the FHS mode will be used if the WAIT parameter was specified and INT mode will be used if the WAIT parameter was not specified.

The INT mode will always be used for the Serial and Datacomm interfaces.

If a very slow device is sending a few bytes at a time, the most efficient method of transfer would be to interrupt the processor whenever data is ready. Both DMA and INT modes operate in this way. The DMA hardware "steals" a single memory cycle from the processor to transfer each byte. The INT mode must completely interrupt the processor and therefore takes more time.

Either type of interrupt (DMA or INT) can occur at any time and will be handled immediately by the system. The interrupt doesn't have to wait for a statement to end before it is serviced. This is not the same as event-initiated branches which are serviced only at the end of a statement.

The INT transfers implemented on the HP-IB and GPIO interfaces use a specialized "burst interrupt" mode. When an interrupt occurs, the system's interrupt service routine will transfer the byte (or word) then wait approximately 20 μs for another byte. If the device is fast enough to accept or generate another byte each 20 μs, the net transfer rate will be much faster than if the system must exit the service routine and then re-enter the routine for the next byte.

Transfer Speeds for Devices

The following table shows the transfer speeds of various devices.

Device			Transfer method			
			Burst Interrupt	Fast Handshake	DMA	Burst DMA
HP-IB (98624)	inbound		55K	130K	350K	–
(bytes/second)	outbound		75K	120K	290K	–
GPIO (98622)	inbound		65K	115K	540K	930K
(transfers/second)	outbound		75K	115K	525K	1050K
Serial (98626)			19 200 Baud			
Datacomm (98628)			19 200 Baud			

Restrictions

All data must be buffered. This means every TRANSFER statement will have one I/O path assigned to a buffer and one I/O path assigned to a device (or file). Additionally, transfers are not permitted with the CRT, keyboard, BCD interface card, or to the tape backup on CS80 disc drives. For files, only BDAT type files can be used with TRANSFER.

A buffer can only have one inbound and one outbound I/O operation (using I/O path names) at any given time. The I/O operation can use TRANSFER, OUTPUT, or ENTER statements. A second I/O operation in the same direction must wait until the completion of the current operation. A second I/O operation in the opposite direction does not have to wait.

The HP-IB and GPIO interfaces support only one I/O operation at any given time. A second operation must wait until the completion of the first operation. The Serial and Datacomm interfaces allow concurrent inbound and outbound transfer operations if each TRANSFER has a unique I/O path name assigned to the device. An OUTPUT or ENTER must wait until completion of transfers in both directions. Thus, concurrent operation requires using TRANS-FER statements and not a mixture of TRANSFER, OUTPUT, and ENTER statements.

The I/O path name assigned to a device can be used in only one I/O operation at a time. However, the path name can be used with OUTPUT, ENTER, and TRANSFER interchangeably. An OUTPUT or ENTER to the I/O path name will be deferred until completion of any active TRANSFER for that path name. All file operations (including CAT, CREATE, OUTPUT, and ENTER) will be deferred until completion of any TRANSFER using the same interface select code.

Interactions

The TRANSFER statement restricts some of the interrupts on various devices. If an ON INTR statement and an ENABLE INTR statement have been executed for an interface, not all possible ON INTR conditions will be triggered during a transfer.

For the GPIO interface, the PFLG (data ready) interrupt is not triggered during a transfer that uses the interface. The EIR (External Interrupt Request) interrupt is triggered even if there is a transfer in progress.

For the Serial Serial interface, the Transmitter Holding Register Empty and Receiver Buffer Full interrupts are not triggered during a transfer that uses the interface. The Receiver Line Status and Modem Status Change interrupts are triggered even if there is a transfer in progress.

For the Datacomm interface, all interrupt conditions are triggered even if a transfer is in progress.

For the HP-IB interface, all interrupt conditions are triggered if they occur during a transfer. However, certain interrupt conditions may occur which will cause the transfer operation to be prematurely terminated.

With the exception of the Handshake Error, the majority of interrupt conditions only occur when the HP-IB interface is configured as a non-controller. If any of the following interrupt conditions are enabled and the given interrupt occurs during a transfer to or from the interface, the user interrupt will be logged and the TRANSFER will be prematurely terminated.

- Parallel Poll Configuration Change
- My Talk Address Received
- My Listen Address Received
- Talker/Listener Address Change
- Trigger Received
- Handshake Error
- Unrecognized Universal Command
- Secondary Command While Addressed
- Clear Received
- Unrecognized Address Command

If one of these interrupt conditions occurs and the given interrupt condition has not been enabled, the interrupt will be ignored and the TRANSFER will not be terminated.

Note

When an abortive interrupt condition is ignored, it is possible for data to be corrupted. It is recommended that abortive interrupt conditions be enabled during a transfer.

The Active Controller and IFC Received interrupt conditions will always prematurely terminate a TRANSFER, even if they have not been enabled.

When an overlapped TRANSFER is prematurely terminated because of an abortive interrupt condition, the following error is logged in the non-buffer I/O path name associated with the given TRANSFER. The error will then be reported the next time the I/O path name is referenced.

```
ERROR 167 I/O interface status error
```

Note that if an ON INTR condition is triggered during a transfer, the ON INTR service routine will be executed at the next end-of-line. However, if a TRANSFER is using the interface specified in an ENABLE INTR statement, the ENABLE INTR statement will wait for the transfer to complete. This means that only one interrupt condition can be triggered during a TRANS-FER since the interface's interrupts cannot be re-enabled until completion of the transfer.

Changing Buffer Attributes

You can change the I/O path name's attributes without changing the current buffer pointers. Just execute another ASSIGN statement with the new attributes. For example:

```
ASSIGN @Path;PARITY OFF
```

You will not be able to change all possible attributes in this manner. The BYTE and WORD attributes cannot be changed once assigned.

By specifying just the I/O path name, the default attributes (except BYTE) can be restored. For example:

```
ASSIGN @Path
```

See the ASSIGN statement in the *BASIC Language Reference* for a complete list of attributes.

Note

It is possible to assign more than one I/O path name to a single named buffer. Using two I/O path names on the same buffer could lead to the corruption of the data in the buffer. Although each path name maintains a separate set of buffer pointers, they are pointing to the same buffer.

Anatomy of a Buffer

Every buffer has two pointers associated with it; a fill pointer and an empty pointer. The fill pointer points to the next available location in the buffer. The empty pointer points to the next data item to be removed from the buffer. For example, the following diagram shows a buffer at some point in time. Two transfers are in use. An inbound transfer is filling the buffer and an outbound transfer is emptying the buffer.

(inbound)

```
TRANSFER @Device TO @Buffer
```
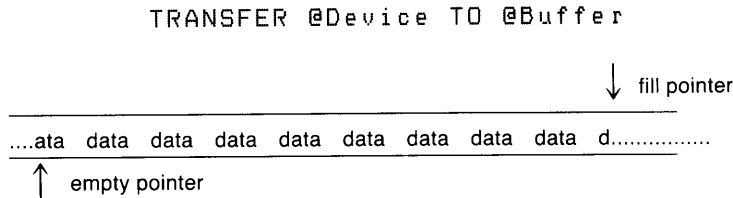

(outbound)

```
TRANSFER @Buffer to @File
```

The buffers used for transfers are circular buffers. In a circular buffer, when a pointer encounters the end of the buffer it "wraps around" to the beginning of the buffer. Normally, once a buffer is filled (the fill pointer catches up to the empty pointer) or emptied (the empty pointer catches up to the fill pointer), the associated TRANSFER is deactivated. Re-executing the TRANSFER statement will initiate another transfer. By using the CONT parameter, a transfer can be specified to remain active after a buffer is full or empty in which case the statement need not be re-executed.

When an I/O path name is assigned to a buffer, a control table is created. As data is transferred along the path, the control table is automatically updated.

When it is necessary to check the condition of the transfer, the STATUS statement can be used to examine the contents of the buffer's registers. The CONTROL statement can then be used to alter the buffer's pointers.

All I/O path names, including those assigned to buffers, use register 0 to indicate the I/O path type. See the table at the end of this section.

If you plan to transfer data through a buffer without using the I/O path name, it will be necessary to change the values of the pointers. Registers 3, 4, and 5 control the positioning of the pointers. If either the fill or empty pointer is changed the appropriate pointer is modified and no other action is taken. Assuming no active transfer, if the byte count is changed, the empty pointer is set to zero and the fill pointer is set to correspond to the length specified. If a transfer is active in both directions, you cannot change the byte count or either pointer. If an inbound transfer is active, the empty pointer will be adjusted to set the byte count as specified. Similarly, if an outbound transfer is active, the fill pointer will be adjusted to match the byte count specified.

When the byte count is set along with either the fill or empty pointer, the pointer is moved to the position specified and the remaining pointer is adjusted to correspond to the specified length.

If all three pointers are changed, they must be a consistent set to prevent the following error:

```
ERROR 19 Improper value or out of range.
```

If both fill and empty pointers are set to the same value, the length must be either zero (buffer empty) or the maximum buffer length (buffer full).

Attempting to change a pointer used by an active TRANSFER will result in the error:

```
ERROR 612 Buffer pointer(s) in use
```

The fill pointer can be changed during an outbound transfer, but not during an inbound transfer. Similarly, the empty pointer can be changed during an inbound transfer, but not during an outbound transfer.

Note

When string variables are used as buffers, the length of the string should not be changed. Although this does not affect the operation of the buffer, it can prevent access to the contents of the buffer by the variable name.

Buffer Status and Control Registers

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

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

Status Register 1   — Buffer type (1 = named, 2 = unnamed)

Status Register 2   — Buffer size in bytes

Status Register 3   — Current fill pointer
Control Register 3   — Set fill pointer

Status Register 4   — Current number of bytes in buffer
Control Register 4   — Set number of bytes

Status Register 5   — Current empty pointer
Control Register 5   — Set empty pointer

Status Register 6   — Interface select code of inbound TRANSFER

Status Register 7   — Interface select code of outbound TRANSFER

Status Register 8   — If non-zero, inbound TRANSFER is continuous
Control Register 8   — Cancel continuous mode inbound TRANSFER if zero

Status Register 9   — If non-zero, outbound TRANSFER is continuous
Control Register 9   — Cancel continuous mode outbound TRANSFER if zero

Status Register 10   — Termination status for inbound TRANSFER

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	TRANSFER Active	TRANSFER Aborted	TRANSFER Error	Device Termination	Byte Count	Record Count	Match Character
Value = 0	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Status Register 11   — Termination status for outbound TRANSFER

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	TRANSFER Active	TRANSFER Aborted	TRANSFER Error	Device Termination	Byte Count	Record Count	0
Value = 0	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 0

Status Register 12   — Total number of bytes transferred by last inbound TRANSFER

Status Register 13   — Total number of bytes transferred by last outbound TRANSFER

The HP-IB Interface	Chapter
	12

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. Load the TRANS and 10 BIN files.

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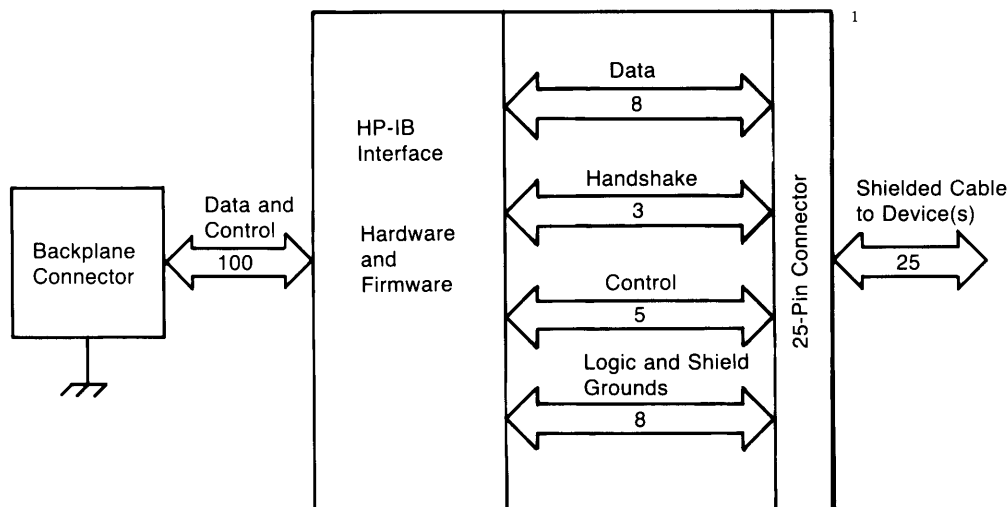
The HP-IB Interface is both easy to use and allows great flexibility in communicating data and control information between the computer and external devices. It is one of the easiest methods to connect more than one device to the same interface.

---

1 See HP-IB Standard for listing of all 24 lines (cable printouts). *HP-IB Installation and Service* manual.

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 **primary 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, respectively. 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's 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 over-seeing 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 assign-ments 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 (by the Active Controller) 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 talker's address is sent (here, the address of the computer; "My Talk Address"), which is also a command.
2. The unlisten command is sent.
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 talker's address (22) is sent, which is a command.
2. The unlisten command is sent.
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 the Active Controller can 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 Active Controller's talk address and then listen addresses 01, 02, and 03. Data is then sent by the controller and accepted by devices at addresses 1, 2, and 3.

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

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

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.

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" later in 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 computer 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 computer 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 computer 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 computer 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 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 computer can also respond to a trigger from another controller on the bus. See "Interrupts While Non-Active Controller" 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 computer 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
```

HP-IB Service Requests

Most HP-IB devices, such as voltmeters, frequency counters, and spectrum analyzers, are capable of generating a "service request" when they require the Active Controller to take action. Service requests are generally made after the device has completed a task (such as making a measurement) or when an error condition exists (such as a printer being out of paper). The operating and/or programming manuals for each device describe the device's capability to request service and conditions under which the device will request service.

To request service, the device sends a Service Request message (SRQ) to the Active Controller. The mechanism by which the Active Controller detects these requests is the SRQ interrupt. Interrupts allow an efficient use of system resources, because the system may be executing a program until interrupted by an event's occurrence. If enabled, the external event initiates a program branch to a routine which "services" the event (executes remedial action).

Chapter 7 described interrupt events in general. This chapter describes the two types of interrupts that can occur on an HP-IB Interface: SRQ interrupts from external devices (that can occur while the computer is an Active Controller), and interrupts that can occur while the computer is a non-Active Controller. The first type of interrupts are described in this section. The second type are described in the section called "The Computer as a Non-Active Controller."

Setting Up and Enabling SRQ Interrupts

In order for an HP-IB device to be able to initiate a service routine in the Active Controller, two prerequisites must be met: the SRQ interrupt event must have a service routine defined, and the SRQ interrupt must be enabled to initiate the branch to the service routine. The following program segment shows an example of setting up and enabling 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
```

Notice the difference in the use of the "@" character.

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 SRQ Interrupts
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 only one device is 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. The device then removes its request for service (by releasing SRQ).

Had the SRQ line not been released, 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.

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 ("I need service") 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 PPOLL 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 PPOLL CONFIGURE statement.

Example

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

The value of Configure_code (any numeric expression can be specified) is first rounded and then used to configure the device's Parallel Poll 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 logic sense of the Parallel Poll Response bit of the device. For instance, a value of 0 implies that the device's response is 0 when its Status Bit message is "I need service."

Example

The following statement configures device at address 01 on interface select code 7 to respond by placing a 0 on bit 4 (DIO5) when its Status Bit response is affirmative.

```
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 when the computer is the Active Controller.

Example

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

Disabling Parallel Poll Responses

The PPOLL UNCONFIGURE statement gives the computer (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 computer 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.

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, since it is 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. This section shows several uses of this statement.

Examples of Sending Commands

As a simple example, suppose the following statement is executed by the Active Controller 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; UNL MTA LISTEN 1
```

```
SEND 7; 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) can 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 Isc;TALK New_controller CMD 9 ! Pass Control
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 SEND statements.

```
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 SPD and Untalk.

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
LLO	Local Lockout	CMD 17
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)

The Computer As a Non-Active Controller

The section called "General Structure of the HP-IB" described how communications take place through HP-IB Interfaces. The functions of the System Controller and Active Controller were likened to a "committee chairman" and "acting chairman," respectively, and the functions of each were described. This section describes how the Active Controller may "pass control" to another controller and assume the role of a non-Active Controller. This action is analogous to designating another committee member to take the responsibility of acting chairman and then becoming a committee member who listens to the acting chairman and speaks when given the floor. The following topics will be discussed:

- Determining whether the computer is currently the Active Controller and/or System Controller
- Determining the computer's HP-IB primary address, and changing it, if necessary
- Passing control to another HP-IB controller
- Requesting service from the Active Controller
- Responsibilities of being a non-Active Controller
- Responding to interrupts that occur while non-Active Controller

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's 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).

Passing Control

The current Active Controller can pass this capability to another computer by sending the Take Control message (TCT). The Active Controller must first address the prospective new Active Controller to talk, after which the TCT message is sent. If the other controller accepts the message, it then assumes the role of Active Controller; this computer then assumes the role of a non-Active Controller.

Passing control can be accomplished in one of two ways: it can be handled by the system, or it can be handled by the program. The PASS CONTROL statement can be used. For example, the following statements first define the HP-IB Interface's select code and new Active Controller's primary address and then pass control to that controller.

```
100    Hp_ib=7
110    New_ac_addr=20
120    PASS CONTROL 100*Hp_ib+New_ac_addr
```

The following statements perform the same functions.

```
100    Hp_ib=7
110    New_ac_addr=20
120    SEND Hp_ib;TALK New_ac_addr CMD 9
```

Once the new Active Controller has accepted the TCT command, the controller passing control assumes the role of a non-Active Controller (or "HP-IB device") on the specified HP-IB Interface. The next section describes the responsibilities of the computer while it is a non-Active Controller.

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.

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.

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 has the ability to keep track of the occurrences of all of the preceding 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.

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 15 enables an interrupt upon becoming the Active Controller. The computer then has the ability to manage bus activities.

Bit 14[1] enables an interrupt upon detecting a change in Parallel Poll Configuration.

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

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

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 10 enables an interrupt when the Active Controller performs a Serial Poll on the computer (in response to its service request).

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.

---

[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 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 or Untalk command.

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 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 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 4[1] enables an interrupt upon receiving a Secondary Command (extended addressing) after the interface receives either its primary talk address or primary listen address. Again, this interrupt provides the computer with a way to detect and respond to special messages from another controller.

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 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 1 enables an interrupt upon detecting a Service Request.

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.

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.

---

[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.

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.

An oversimplified 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          RETURN! Ignore other interrupts,
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. See "Responding to Serial Polls" for further details.

Addressing a Non-Active Controller

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 computer 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 computer 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 computer 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) 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 computer'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".

Requesting Service

When the computer is a non-Active Controller, it has the capability of sending an SRQ to the current Active Controller. The following statement is an example of requesting service from the Active Controller of the HP-IB Interface on select code 7.

```
CONTROL 7,1;64
```

The REQUEST statement can be used to perform the same function.

```
REQUEST 7;64
```

Both of the preceding example place a logic True on the SRQ line. (Note that the line may already be set True by another device.) Other bits may be set in the Status Byte message, indicating that other device-dependent conditions exist.

The SRQ line is held True until the Active Controller executes a Serial Poll or this computer executes a REQUEST with bit 6 equal to 0. (Note also that the line may still be held True by another device.)

When the Active Controller detects an SRQ message, it usually polls device(s) on the bus to determine which need(s) service and what kind of service is needed. To determine *which* device(s) are requesting service, the Active Controller conducts a Parallel Poll. If there are not more than one device currently capable of requesting service, the Parallel Poll is not necessary.

The Parallel Poll is conducted by sending an Identify (ATN & EOI). This non-Active Controller's response to a Parallel Poll performed by the Active Controller depends on the current Parallel Poll Response set up for this controller. Setting up this controller's Parallel Poll Response is described in the next section.

If the Active Controller needs to determine what service action is required for a particular device, it performs a Serial Poll on the device(s) that responded to the Parallel Poll with an "I need service." As each device is Serially Polled, it responds by placing its Status Byte on the bus.

This non-Active Controller's response to a Serial Poll performed by the Active Controller is handled automatically by the system. The Status Byte is the byte sent to the Serial Poll Response Byte Register (with CONTROL or REQUEST, as shown above). A subsequent section further describes this non-Active Controller's responses to Serial Polls.

Responding to Parallel Polls

Before performing a Parallel Poll of bus devices, the Active Controller configures selected device(s) to respond on one of the eight data lines. Each device is directed to respond on a particular data line with a logic True or False; the logic sense of the response informs the Active Controller either "I do need service" or "I don't need service." The logic sense of the response is also specified by the Active Controller. This response to the Parallel Poll is known as the Status Bit message.

After the desired devices have been told how to respond, the Active Controller can send the Identify message and read the Status Bits placed on the data lines to determine which device(s) need service. Identify is sent by placing ATN and EOI in the logic True state. All devices which are currently configured for the poll respond as configured.

To configure its own Parallel Poll Response, the computer must receive a Parallel Poll Configure (PPC) command followed by a Parallel Poll Enable (PPE) command from the Active Controller. Receiving this "Parallel Poll Configuration Change" generates an interrupt (this type of interrupt is enabled by setting bit 14 of the Interrupt Enable Register). The service routine takes care of configuring this controller's response by first accepting the encoded "configure byte" (the PPE command from the Active Controller) and then setting up a corresponding response.

The desired Status Bit message can be configured and sent by one of two methods. The first, and simplest, method is to define an automatic response by using the PPOLL RESPONSE statement. With this method, the computer reads the configure byte from the data lines (HP-IB STATUS Register 7) and then writes the byte's numeric value into HP-IB CONTROL Register 5. The following statements show an example of configuring this controller's Parallel Poll Response.

```
100   STATUS 7,7;Configure_code
110   CONTROL 7,5;Configure_code
120   I_need_service=0
130   PPOLL RESPONSE 7;I_need_service
```

When the computer receives a subsequent Identify from the Active Controller, the specified response ("I do/don't need service") is automatically sent to the Active Controller. The computer will probably need to respond to a Serial Poll, which is described in the next section.

The second method requires that the service routine decode the configure byte and set up the corresponding response. The configure byte read from HP-IB STATUS Register 7 contains 5 bits of data encoded with the following information:

Control Register 5   **Parallel Poll Response 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		Uncon-figure	Logic Sense	Data Bit Used For Response		
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Bit 4 determines whether a response will or will not be configured. A 1 tells this controller *not* to configure a response, and a 0 tells the controller to configure a response.

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

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

The service routine calculates the desired response and places the appropriate bit pattern in HP-IB CONTROL Register 2. For instance, if the configure byte has a value of 12 (positive-true logic on DIO5 for "I need service"), the value sent to CONTROL Register 2 is 16 for "I need service." The appropriate statement might be:

```
CONTROL 7,2;16
```

When the Identify is received from the Active Controller, the specified response is made automatically.

As another example, suppose that the configure byte has a value of 7. The Status Bit to be written into DIO8 would be a 0 for "I need service." The corresponding statement might be:

```
CONTROL 7,2;0
```

The following general routine calculates the value to be sent to CONTROL Register 2:

```
790   STATUS 7,7;Config_code   ! Read data lines,
800   Config_code=Config_code MOD 256   ! Strip 8 MSBs,
810   Unconfig=BIT(Config_code,4)
820   Sense=BIT(Config_code,3)
830   IF Unconfig=1 OR Sense=0 THEN   ! Unconfigure,
840      Ppoll_response=0
850   ELSE ! Configure,
860      Status_bit=Config_code MOD 8 ! Get bits 2-0,
870      Ppoll_response=2^Status_bit   ! Set proper bit,
880   END IF
890   CONTROL 7,2;Ppoll_response
```

Responding to Serial Polls

As a non-Active Controller, the response to Serial Polls is automatically handled by the system. The desired Serial Poll Response Byte is sent to HP-IB CONTROL Register 1. If bit 6 is set (bit 6 has a value of 64), an SRQ is indicated from this controller. All other bits can be considered to be "device-dependent," and can thus be set according to the program's needs.

The following statement sets up a response with SRQ and bits 1 and 0 set to 1's.

```
CONTROL 7,1;64+2+1
```

When the Active Controller performs a Serial Poll on this non-Active Controller, the specified byte is automatically sent to the Active Controller by the system.

This non-Active Controller is automatically addressed to talk by the Active Controller during a Serial Poll. If interrupts are concurrently enabled for My Address Change and/or Talker Active interrupts, the ON INTR branch will be initiated due to the reception of this controller's talk address. However, since the Serial Poll Response is terminated with the Untalk command, this controller may no longer be addressed to talk when the service routine begins its execution. In such a case, the SPAS interrupt (if enabled) will also be indicated. If desired, the interrupt may be ignored.

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).

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

Bit 10 set indicates that the interface is in the Listener Addressed State and is currently an active listener. If Bit 4 of the Interrupt Enable register is set (Secondary Command While Addressed), two additional conditions are required to enter this state: the interface must have first received its own primary address followed by a secondary command, and it must have **accepted** the secondary command (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 and is currently an active talker. This state is entered in a manner analogous to the Listener Addressed State (see Bit 10 above).

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.

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 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 Active Controller. The response itself is set up by writing to CONTROL register 2 of the HP-IB interface.

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 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 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).

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 or Listener. 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.

HP-IB Control Lines
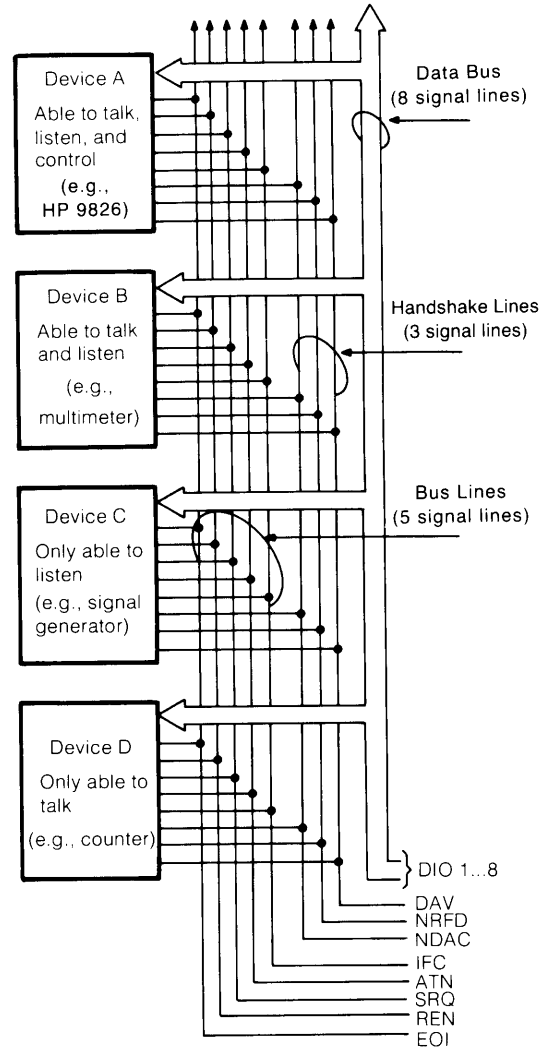
Handshake Lines

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". Only the Active Controller can set ATN true.

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 as an END indication (ATN false) during ENTER statements and as the Identify message (ATN true) 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 "%" 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 terminator or statement terminator	Item terminator or statement terminator	Immediate statement terminator
Statement terminator required?	Yes	Yes	No	No
Early termination allowed?	No	No	No	Yes

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 Active 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 Active 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.

HP-IB Status and Control Registers

Status Register 0   Card identification = 1

Control Register 0   Reset interface if non-zero

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	Hardware Interrupt Level Switches		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
0	0	0	0	Reserved For Future Use	Handshake In Progress	Interrupts Enabled	TRANSFER In Progress
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

HP-IB Status and Control Registers (cont.)

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   Writing anything to this register releases NDAC holdoff. If non-zero, accept last secondary address as valid. If zero, don't accept last secondary address (stay in LPAS or TPAS state).

HP-IB Status and Control Registers (cont.)

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

Control Register 5

Parallel Poll Response 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		Uncon- figure	Logic Sense	Data Bit Used For Response		
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

HP-IB Status and Control Registers (cont.)

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.

HP-IB Status and Control Registers (cont.)

Interrupt Enable Register (ENABLE INTR)

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

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	(MSB) ◄──── HP-IB Primary Address of Interface ────► (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 0
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 Command Group	Secondary Command While Addressed	Clear Received	My Address Received (MLA or MTA)	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 and DMA 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 ATN	Error	ATN MTA UNL ATN	Error
Not Active Controller	IFC (duration ≥100 μsec)* REN 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	$\overline{\text{REN}}$	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			

PASS CONTROL

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active Controller	Error	ATN TAD TCT ATN	Error	ATN TAD TCT ATN
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≥25μs) Read byte $\overline{EOI}$ Restore ATN to previous state	Error	ATN & EOI (duration≥25μs) Read byte $\overline{EOI}$ 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 $\overline{ATN}$	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 $\overline{SPE}$ $\overline{ATN}$ Read data ATN SPD UNT	Error	ATN UNL MLA TAD $\overline{SPE}$ $\overline{ATN}$ 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 UNL LAG GET	ATN GET	ATN MTA UNL LAG GET
Not Active Controller	Error			

The Datacomm Interface	Chapter
	13

Introduction

The HP 98628 Data Communications Interface enables your Series 200 computer to communicate with any device that is compatible with standard asynchronous or HP Data Link data communication protocols. Devices can include various modems or link adapters, as well as equipment with standard RS-232C or current loop links.

This chapter discusses both asynchronous and Data Link protocols, and provides useful programming techniques so you can quickly create working programs. Subject areas that are similar for both protocols are combined, while information that is unique to one protocol or the other is separated according to application.

Prerequisites

It is assumed that you are familiar with the information presented in Data Communication Basics, and that you understand data communication hardware well enough to determine your needs when configuring the datacomm link. Configuration parameters include such items as half/full duplex, handshake, and timeout requirements. If you have any questions concerning equipment installation or interconnection, consult the appropriate interface or adapter installation manuals.

The datacomm interface supports several cable and adapter options. They include:

- RS-232C Interface cable and connector wired for operation with data communication equipment (male cable connector) or with data terminal equipment (female cable connector).
- HP 13264A Data Link Adapter for use in HP 1000- or HP 3000-based Data Link network applications
- HP 13265A Modem for asynchronous connections up to 300 baud, including built-in autodial capability[1].
- HP 13266A Current Loop Adapter for use with current loop links or devices.

Illustrator: Diagram of interface and options here

Some of the information contained in this chapter pertains directly to certain of these devices in specific applications.

---

[1] The HP 13265A modem is compatible with Bell 103 and Bell 113 Modems, and is approved for use in the USA and Canada. Most other countries do not allow use of user-owned modems. Contact your local HP Sales and Service office for information about local regulations.

Before you begin datacomm operation, be sure all interfaces, cables, connectors, and equipment have been properly plugged in. Power must be on for all devices that are to be used. Consult applicable installation manuals if necessary.

Protocol

Two protocols are switch selectable on the datacomm interface. They are also software selectable during normal program operation. The switch setting on the interface determines the default protocol when the computer is first powered up. Protocol is changed between Async and Data Link during program operation by selecting the new protocol, waiting for the message to reach the card, then resetting the card. The exact procedure is explained in "Protocol Selection".

Asynchronous Communication Protocol

Asynchronous data communication is the most widely used protocol, especially in applications where high data integrity is not mandatory. Data is transmitted, one character at a time, with each character being treated as an individual message. Start and stop bits are used to maintain timing coordination between the receiver and transmitter. A parity bit is sometimes included to detect character transmission errors. Asynchronous character format is as follows: Each character consists of a start bit, 5 to 8 data bits, an optional parity bit, and 1, 1.5, or 2 stop bits, with an optional time gap before the beginning of the next character. The total time from the beginning of one start bit to the beginning of the next is called a character frame.

Parity options include:

- **NONE**   No parity bit is included.
- **ODD**   Parity set if EVEN number of "1"s in character bits.
- **EVEN**   Parity set if ODD number of "1"s in character bits.
- **ONE**   Parity bit is set for all characters.
- **ZERO**   Parity bit is zero for all characters.

Here is a simple diagram showing the structure of an asynchronous character and its relationship to previous and succeeding characters:

Preceding Character | Line in Idle State (Mark) | Start Bit | 1 | 0 | 1 | 0 | 0 | 0 | 1 | Parity Bit | Stop Bit(s) | Start Bit for Next Character

Beginning of Character ◄──────── Single Character Frame ────────► End of Character
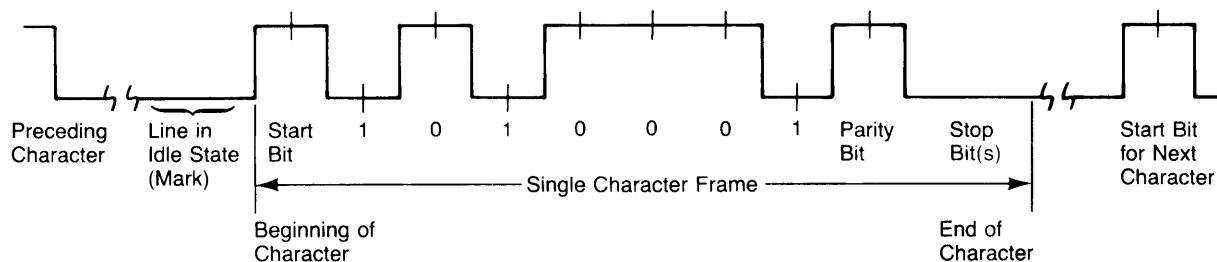
Data Link Communication Protocol

Data Link protocol overcomes the data integrity limitations of Async by handling data in blocks. Each block is transmitted as a stream of individual asynchronous characters, but protocol control characters and block check characters are also transmitted with the data. The receiver uses the protocol control characters to determine block boundaries and data format. Block check characters are used to detect transmission errors. If an error occurs, the block is usually retransmitted until it is successfully received. Block protocol and format is similar to Binary Synchronous Communication (BSC or Bisync, for short).

Data Link protocol provides for two transmission modes: transparent, and normal. In transparent mode, any data format can be transferred because datacomm control characters are preceded by a DLE character. If a control character is sent without an accompanying DLE, it is treated as data. When normal mode is used, only ASCII data can be sent, and datacomm control characters are not allowed in the data stream.

The HP 1000 and HP 3000 computers usually transmit in transparent mode. All transmissions from your desktop computer are sent as transparent data. If your application involves non-ASCII data transfers (discussed later in this chapter), be sure the HP 1000 or HP 3000 network host is using transparent mode for all transmissions to your computer.

Each data block sent to the network host by the datacomm interface is structured as follows:

←—Start of Block					End of Block—→
D_{LE}	S_{TX}	G_{ID}	D_{ID}	text (data)	D_{LE} E_{TX} B_{CC} B_{CC}
1	2			3	4 5
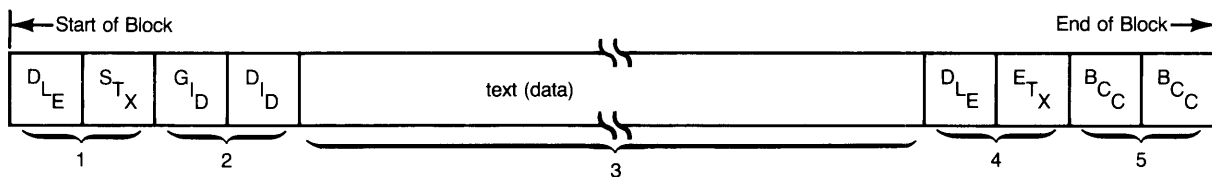
1. The "start transmission" control characters identify the beginning of valid data. If a DLE is present, the data is transparent; If absent, data is normal. All data from your desktop computer is transparent.

2. The terminal identification characters are included in blocks sent to the network host. Blocks received from the network host do not contain these two characters.

3. Data characters are transmited in succession with no time lapse between characters.

4. The "end transmission" control characters identify the end of data. DLE ETX or DLE ETB indicate transparent data. ETX or ETB indicates normal data.

5. Block check characters (usually two characters) are used to verify data integrity. If the value received does not match the value calculated by the receiver, the entire block is rejected by the receiver. Block check includes Group Identifier (GID) and Device Identifier (DID) characters in transmissions to the network host.

Protocol control characters are stripped from the data transfer, and are not passed from the interface to the computer. For information about network polling, terminal selection and other Data Link operations, consult the Data Link network manuals supplied with the HP 1000 or HP 3000 network host computer.

Data Transfers Between Computer and Interface

Data transfers between your desktop computer and its datacomm interface involve two message types: control blocks and data. Control blocks contain information sent to and received from the interface regarding its operation. Data is sent to and received from a remote device through the interface. Control blocks are not sent to or received from remote devices. Both types are encountered in both output and input operations as follows:

- Outbound control blocks are created by CONTROL statements.
- Outbound data messages are created by OUTPUT statements.
- Inbound control blocks are created by certain protocol operations such as Data Link block boundaries, or Async prompt, end-of-line, parity/framing error, or break detection.
- Inbound data messages are created by the interface as messages are received from the remote. They are transferred to BASIC by ENTER statements.

Outbound Control Blocks

Outbound control blocks are messages from your computer to the datacomm interface that contain interface control information. They are usually generated by CONTROL statements, although OUTPUT...END creates a control block that terminates a given Async transmission or forces a block to be sent on the Data Link. Outbound control blocks are serially queued with data, and executed by the interface in the same order as created by BASIC. The single exception to the queued control block rule is when a non-zero value is output to Control Register 0 (Interface Reset) which is executed immediately.

Note

When an interface card reset is executed by use of a CONTROL statement, the control block that results is transmitted directly to the interface. It is not queued up, so any previously queued data and control blocks are destroyed. To prevent loss of data, be sure that all queued messages have been sent before resetting the datacomm interface. Status Register 38 returns a value of 1 when the outbound queue is empty. Otherwise, its value is 0. To prevent loss of inbound data, Status Register 5 must return a value of zero prior to reset.

Inbound Control Blocks

Inbound control blocks are messages from the interface to the computer that identify protocol control information. Which item(s) are allowed to create a control block is determined by the contents of Control Register 14. Status Registers 9 and 10 identify the contents of the block, and Control Register 24 defines what protocol characters are also included with inbound Async data messages. Refer to the BASIC Language Reference Control and Status Register section for details about register contents for various control block types.

Two types of information are contained in each control block: type and mode. The type is contained in STATUS register 9; the mode in STATUS register 10. Type and Mode values can be used to interpret datacomm operation as follows:

Async Protocol Control Blocks

Type	Mode	Interpretation
250	1	Break received (channel A).
251	1[1]	Framing error in the following character.
251	2[1]	Parity error in the following character.
251	3[1]	Both Framing and Parity error in the following character.
252	1	End-of-line terminator detected.
253	1	Prompt received from remote.

Data Link Protocol Control Blocks

Type	Mode	Interpretation
254	1	Preceding block terminated by ETB character.
254	2	Preceding block terminated by ETX character.
253[2]		(See following table for Mode interpretation.)

Mode Bit(s)	Interpretation
0	1 = Transparent data in following block.
	0 = Normal data in following block.
2,1	00 = Device Select (most common).
	01 = Group Select
	10 = Line Select
3	1 = Command Channel
	0 = Data Channel

For Data Link applications, control blocks are normally set up for end-of-block (ETB or ETX). Control blocks are then used to terminate ENTER operations. Control block contents are not important for most applications unless you are doing sophisticated protocol-control programming.

For Async applications, terminal emulator programs usually use prompt and end-of-line control blocks. Use of other functions such as break or error detection depend on the requirements of the individual application.

---

[1] Parity/framing error control blocks are not generated when characters with parity and/or framing errors are replaced by an underscore (_) character.

[2] This type is used mainly in specialized applications. In most cases, you can expect a Mode value of zero or one for Type 253 Data Link control blocks. For most Data Link applications, control blocks are not used by programmers.

Outbound Data Messages

Outbound data messages are created when an OUTPUT statement is executed. Here is a short summary of how OUTPUT parameters can affect datacomm operation.

- Async protocol: Data is transmitted directly from the outbound queue. When operating in half-duplex, OUTPUT...END causes the interface to turn the line around and allow the remote device to send information back (line turn-around is initiated when the interface sets the Request-to-send line low). OUTPUT...END has no effect when operating in full duplex.

- Data Link protocol: Data messages are concatenated until at least 512 characters are available, then a block of 512 characters is sent. Block boundaries may or may not coincide with the end of a given OUTPUT message.
  You can force transmission of shorter blocks by using the OUTPUT...END statement. The interface then transmits the last pending block regardless of its length. This technique is useful for ensuring that block boundaries coincide with message boundaries, or for sending one message string per block when you are transmitting short records.

- Unless a semicolon or END appears at the end of a free-field OUTPUT statement, an EOL sequence is automatically sent at the end of the data. The EOL sequence is also suppressed by using the appropriate IMAGE specifier in an OUTPUT statement. See Chapter 4 for further information.

Inbound Data Messages

Inbound data messages are created by the datacomm interface as information is received from the remote. ENTER statements are terminated when a control block is encountered or the input variable is filled. Whether control characters are included in the data stream depends on the configuration of Control Register 24 (Async operation only). Control information is never included in inbound data messages when using Data Link protocol.

With this brief introduction to the data communications capabilities of the HP 98628 Datacomm Interface, you are ready to begin programming your desktop computer for datacomm operation. The next section of this chapter introduces BASIC datacomm programming techniques using simple terminal emulator examples that can be readily expanded into much more sophisticated datacomm programs.

Overview of Datacomm Programming

Your desktop computer uses four BASIC statements for data communication with remote computers, terminals, and other peripheral devices. Datacomm programs include part or all of the following elements:

- CONTROL statements to configure the datacomm link and establish the connection.
- OUTPUT and ENTER statements to transfer information.
- STATUS statements to monitor operation.
- CONTROL statements to alter link parameters during the session, if needed for unusual applications.
- OUTPUT and ENTER statements to transfer additional information.
- A CONTROL statement to disconnect at the end of the session.

Here is a simple example of an Async terminal emulator that uses default parameters. The user must disconnect at the end of a session by executing the command CONTROL Sc,12;0 from the keyboard.

```
1000        Sc=27                          ! Datacomm on Select Code 27.
1010        CONTROL Sc,14;6                ! Set Control Block Mask.
1020        OUTPUT Sc;CHR$(13);            ! Datacomm interface uses defaults
1025                                       ! and automatically connects to line.
1030   Check_reader:DIM A$[700]            ! Up to 700 characters per line.
1040        STATUS Sc,5;Rx_avail_bits      ! Get Rx queue status.
1050        IF Rx_avail_bits>1 THEN
1060            ENTER Sc USING "#,K";A$     ! Get data from queue.
1070            PRINT USING "#,K";A$        ! Print data.
1080            STATUS Sc,9;R               ! Get Control Block TYPE field.
1090            IF R=253 THEN
1100                LINPUT "Enter line to send to remote.";A$
1110                OUTPUT Sc;A$;CHR$(13);
1120            END IF
1130        END IF
1140        GOTO Check_reader
1150        END
```

While this program shows the relative simplicity of using your computer for data communication, most applications require more sophisticated techniques. The following pages show more elaborate structures to illustrate some of the concepts used in creating programs for datacomm applications.

Two sample terminal emulator programs, one for Async and one for Data Link, are used in this chapter to show you how to write datacomm programs with a minimum of difficulty and complexity. Both versions are very similar; differences are explained fully. The emulators are explained in logical sequence, with complete program listings included at the end. The examples can be used as written, or expanded to include other features. They are designed to demonstrate program structures and programming techniques that are used in many data communication applications.

Establishing the Connection

Determining Protocol and Link Operating Parameters

Before information can be successfully transferred between two devices, a communication link must be established. You must include the necessary protocol parameters to ensure compatibility between the communicating machines. To determine the proper parameters for your application, select Async or Data Link protocol, then answer the following questions:

For BOTH Async and Data Link Operation:
- Is a modem connection being used? What handshake provisions are required? (Data Link does not use modems, but multi-point Async modem connections use a protocol compatible with Data Link.)
- Is half-duplex or full-duplex line protocol being used?

For Async Operation ONLY:
- What line speed (baud rate) is being used for transmitting?
- What line speed is being used for receiving?
- How many bits (excluding start, stop, and parity bits) are included in each character?
- What parity is being used: none, odd, even, always zero, or always one?
- How many stop bits are required on each character you transmit?
- What line terminator should you use on each outgoing line?
- How much time gap is required between characters (usually 0)?
- What prompt, if any, is received from the remote device when it is ready for more data?
- What line terminator, if any, is sent at the end of each incoming line?

For Data Link Operation ONLY:
- What line speed (baud rate) is being used? (Data Link uses the same speed in both directions.)
- What parity is being used: none (HP 1000 network host), or odd (HP 3000 network host)?
- What is the device Group IDentifier (GID) and Device IDentifier (DID) for your terminal?
- What is the maximum block length (in bytes) the network host can accept from your terminal?

All these parameters are configured under program control by use of CONTROL statements. Alternately, default values for line speed, modem handshake, parity, and Async or Data Link protocol selection can be set using the datacomm interface configuration switches. Other default parameters are preset by the datacomm interface to accommodate common configurations. You can use the defaults, or you can override them with CONTROL statements for program clarity and immunity to card settings. Default Control Register values are shown in the "Interface Register" section in the back of the *BASIC Language Reference* manual. The *HP 98628 Datacomm Interface Installation* manual explains how to set the default switches on the interface.

The next section of this chapter shows a summary of the available default options and switch settings for both Async and Data Link.

Using Defaults to Simplify Programming

The datacomm interface includes two switch clusters. One cluster is used to program the select code and interrupt level (hardware priority). The other cluster sets defaults for protocol, line speed (baud rate), modem handshake, and parity. Setting the defaults on the card eliminates the need to program the corresponding interface CONTROL registers. These defaults are useful in applications where the configuration of the link is rarely altered, and the program is not used on other machines with dissimilar configurations. They also enable a beginning programmer to use OUTPUT and ENTER statements to perform simple datacomm operations without using CONTROL or STATUS statements. On the other hand, where link configuratiion may vary, or where programs are used on several different machines with dissimilar configurations, it is usually worthwhile to override the defaults with CONTROL statements as described in the programming examples. This assures known datacomm behavior, independent of interface defaults.

Here, for your convenience is a brief summary of the default switch options:
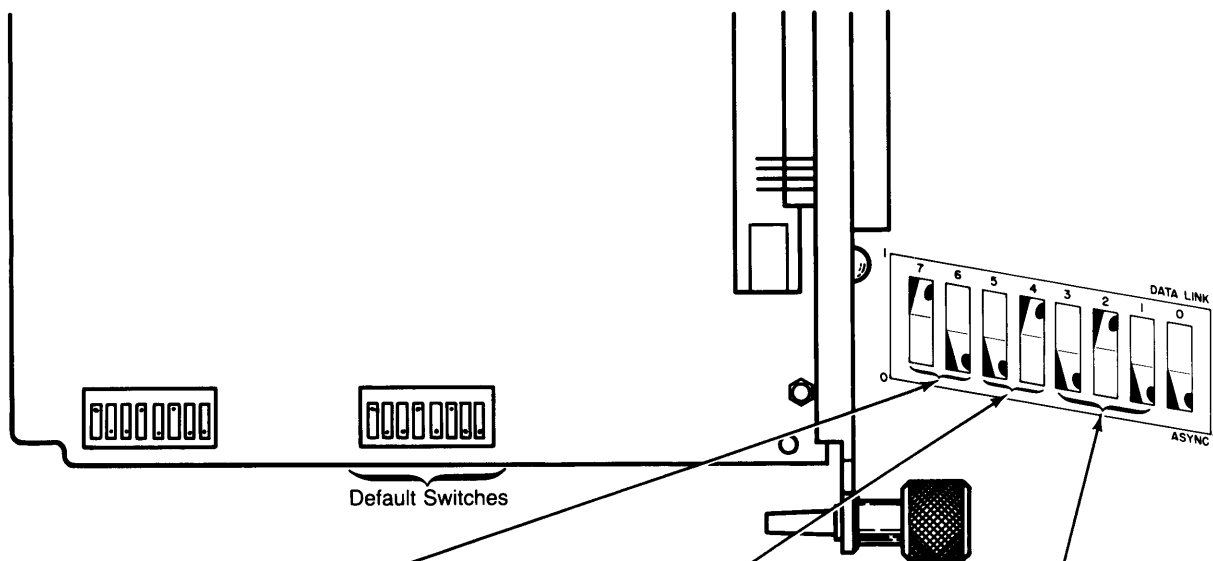
Parity	Bits/Char
00 = None	8
01 = None	7
10 = Odd	7
11 = Even	7

Hardware Handshake
00 = Handshake OFF, non-modem connection[1]
01 = FULL Duplex modem connection[2]
10 = HALF Duplex modem connection[2]
11 = Handshake ON, non-modem connection[1]

Baud Rate	Stop Bits
000 = 110	2
001 = 150	2
010 = 300	1
011 = 600	1
100 = 1200	1
101 = 2400	1
110 = 4800	1
111 = 9600	1

Async Default Configuration Switches

[1] Default No Activity timeout: Disabled
[2] Default No Activity timeout: 10 minutes

DID: ("@"..."G")		Baud Rate	Hardware Handshake
000 = @	100 = D	00 = 300	**00 = Handshake OFF, non-modem connection**
001 = A	101 = E	01 = 1200	01 = FULL Duplex modem connection
010 = B	110 = F	**10 = 9600**	10 = HALF Duplex modem connection
011 = C	111 = G	11 = 19200	11 = Handshake ON, non-modem connection

Default GID = "A"   Default No Activity timeout: 10 minutes
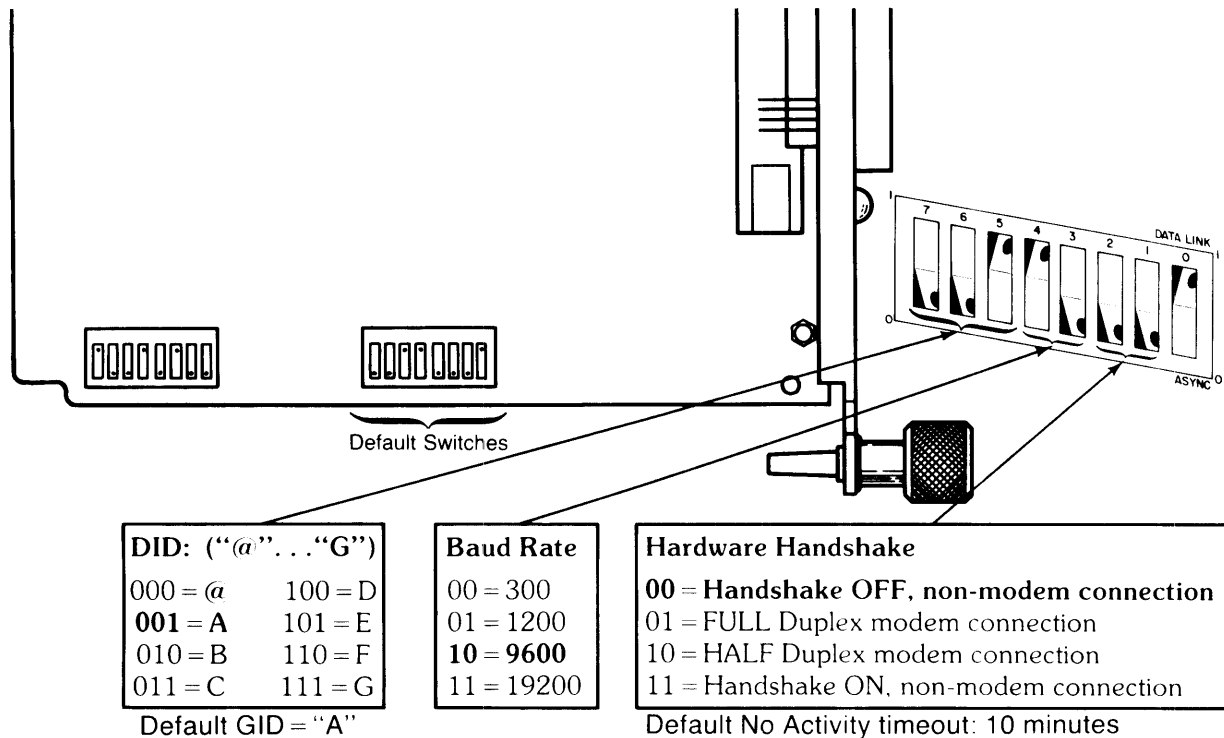
Data Link Default Configuration Switches

Resetting the Datacomm Interface

Before you establish a connection, the datacomm interface must be in a known state. **The datacomm interface does not automatically disconnect from the datacomm link when the computer reaches the end of a program.** To prevent potential problems caused by unknown link conditions left over from a previous session, it is a good practice to reset the interface card at the beginning of your program before you start configuring the datacomm connection. Resetting the card causes it to disconnect from the line and return to a known set of initial conditions.

In the following example, a numeric variable is used to define the select code. The second statement resets the card after the select code has been defined.

```
1110      Sc=20          ! Set select code to 20,
1160      CONTROL Sc,0;1  ! Reset the card to disconnect from line,
```

Protocol Selection

During power-up and reset, the card uses the default switches to preset the card to a known state. The protocol select switch defines which protocol the card uses at power-up only. If the default protocol is the same as you are using, you can skip the protocol selection statements. However, if the switch might be set to the wrong protocol, or if you want to change protocol in the middle of a program, you can use a CONTROL statement to select the protocol. After the protocol is selected, reset the card again to make the change. Here is how to do it:

Select the protocol to be used:

```
1170    CONTROL Sc,3;1      ! Select Async Protocol
or
1170    CONTROL Sc,3;2      ! Select Data Link Protocol
```

Wait until the protocol select message has been sent to the card, (lines 1180-1200) then reset the card. The Reset command restarts the interface microcomputer using the selected protocol.

```
1180    Wait:STATUS Sc,38;All_sent   ! Get transmit queue status,
1190         IF NOT All_sent THEN Wait  ! If not done, wait,
1200         CONTROL Sc,0;1             ! Reset interface card,
```

Note

Be careful when resetting the interface card during normal program operation. Data and Control information are sent to the card in the same sequence as the statements originating the information are executed. When a card reset is initiated by a CONTROL statement, the reset is not placed in the queue with outbound data, but is executed immediately. Therefore, if there is other information in the output queue waiting to be sent, a reset can cause the data to be lost. To prevent loss of data, use STATUS statements (register 38) to verify that all data transfers have run to completion before you reset the interface.

You are now ready to program datacomm options that are related to the selected protocol. In applications where defaults are used, the options are very simple. The following pair of examples shows how to set up datacomm options for each protocol.

Datacomm Options for Async Communication

This section explains how to configure the datacomm interface for asynchronous data communication. The example used shows how to set up all configurable options without considering default values. Some statements in the example are redundant because they override interface defaults having the same value. Others may or may not be redundant because they override configuration switch options. The remaining statements are necessary because they override the default values, replacing them with non-default values required for proper operation of the example program. If you are not familiar with Asynchronous protocol, consult the section on protocol for the needed background information.

The following program lines set up all the CONTROL register options (a 300-baud connection to an HP 1000 is assumed):

```
    1250   CONTROL  Sc,14;3        ! Set control block mask for EOL & Prompt.
*   1260   CONTROL  Sc,15;0        ! No modem line-change notification.
    1270   CONTROL  Sc,16;0        ! Infinite connection timeout.
→   1280   CONTROL  Sc,17;0        ! Disable No Activity timeout.
*   1290   CONTROL  Sc,18;40       ! Lost Carrier 400 ms. *
*   1300   CONTROL  Sc,19;10       ! Transmit timeout 10 s.
→   1310   CONTROL  Sc,20;7        ! Transmit speed = 300 baud.
→   1320   CONTROL  Sc,21;7        ! Receive speed = 300 baud.
    1330   CONTROL  Sc,22;2        ! EQ/AK (as terminal) handshake.
→   1340   CONTROL  Sc,23;1        ! Full Duplex connection.
    1350   CONTROL  Sc,24;66       ! Remove protocol characters except
    1360                           !   EOL.  Change errors to Underscore.
    1370   CONTROL  Sc,26;6        ! Assign AK character for EQ/AK.
    1380   CONTROL  Sc,27;5        ! Assign EQ character for EQ/AK.
*   1390   CONTROL  Sc,28;2,13,10  ! Set EOL sequence to be CR-LF.
*   1400   CONTROL  Sc,31;1,17     ! Set Prompt to be DC1. (33 not used).
→   1410   CONTROL  Sc,34;2        ! Seven bits per character.
→   1420   CONTROL  Sc,35;0        ! One stop bit.
→   1430   CONTROL  Sc,36;1        ! Odd parity.
*   1440   CONTROL  Sc,37;0        ! No inter-character time gap.
*   1450   CONTROL  Sc,39;4        ! Set BREAK to four character times.
```

*: Redundant statement. Same as interface default.

→: May be redundant. Overrides configuration switch option.

Refer to the Control Register tables in the back of the *BASIC Language Reference* manual as you examine the CONTROL statements. The paragraphs which follow explain register functions and how to configure them.

Control Block Contents

Configuration of the link begins with register 14 which determines what information is placed in the control blocks that appear in the input (receive) queue. In this example, only the end-of-line position and prompts are identified. Parity or framing errors in received data, and received breaks are not identified in the queue. This register interacts with Control registers 28 thru 33.

Modem-initiated ON INTR Branching Conditions

Register 15 is rarely used in most applications because the interface usually manages all interaction with the modem. Modem interrupts are helpful when you are simulating your own line protocol. This register determines what changes in one or more modem lines can cause a program branch to occur when an ON INTR statement is active for that select code. Values from 0 thru 31 can be used, where a "1" in a bit position enables branching whenever the corresponding signal line changes state. Lines correspond to bits 0 thru 4 of STATUS register 7. In this example, modem functions are handled by the interface; no interaction with BASIC is necessary. If this register is given a non-zero value, bit 3 of the ENABLE INTR mask should be set. (ENABLE INTR statement is line 1820 of the example terminal emulator program.)

Datacomm Line Timeouts

Registers 16-19 set timeout values to force an automatic disconnect from the datacomm link when certain time limits are exceeded. For most applications, the default values are adequate. A value of zero disables the timeout for any register where it is used. Each register accepts values of 0 thru 255; units vary with the register function.

- Register 16 (Connection timeout) sets the time limit (in seconds) allowed for connecting to the remote device. It is useful for aborting unsuccessful attempts to dial up a remote computer using public telephone networks.

- Register 17 (No Activity timeout) sets an automatic disconnect caused by no datacomm activity for the specified number of minutes. Default value is determined by default hand-shake switch setting. Default is not affected by CONTROL statements to Control Register 23 (hardware handshake).

- Register 18 (Lost Carrier timeout) disconnects when:

    Full Duplex: Data Set Ready (Data Mode) or Data Carrier Detect go false, or
    Half Duplex: Data Set Ready goes false,

    indicating that the carrier from the remote modem has disappeared from the line. Value is in multiples of 10 milliseconds.

- Register 19 (Transmit timeout) disconnects when a loss-of-clock occurs or a clear-to-send (CTS) is not returned by the modem within the specified number of seconds.

Line Speed (Baud Rate)

The transmit and receive line speed(s) are set by Control Registers 20 and 21, respectively. Each is independent of the other, and they are not required to have identical values. The following baud rates are available for Async communication:

Register Value	Baud Rate	Register Value	Baud Rate	Register Value	Baud Rate	Register Value	Baud Rate
0	0^1	4	134.5	8	600^2	12	3600
1	50	5	150^2	9	1200^2	13	4800^2
2	75	6	200	10	1800	14	9600^2
3	110^2	7	300^2	11	2400^2	15	19 200

[1] An external clock must be provided for this option.

[2] These speeds can be programmed using the default switches on the interface card. Other speeds are accessed by CONTROL statements. (The HP 13265A Modem can be operated up to 300 baud.)

All configurable line speeds are available to CONTROL Registers 20 and 21. Only the eight speeds indicated can be selected using the default switches (see the switch configuration diagram earlier in this chapter). When the configuration switch defaults are used, transmit and receive speeds are identical. The selected line speed must not exceed the capabilities of the modem or link.

Handshake

Registers 22 and 23 configure handshake parameters. There are two types of handshake:

- **Software or protocol handshake** specifies which of the participants is allowed to transmit while the other agrees to receive until the exchange is reversed. **Options** include:

    1. **No handshake**, commonly used with connections to non-interactive devices such as printers.

    2. **Enq/Ack (EQ/AK)** or DC1/DC3 handshake, with the desktop computer configured either as a host or a terminal. Handshake characters are defined by registers 26 and 27.

    3. **DC1/DC3 handshake** with the desktop computer as both a host AND a terminal. Handshake characters are defined by registers 26 and 27. This option simplifies communication between two desktop computers.

- **Hardware or modem handshake** that establishes the communicating relationship between the interface and the associated datacomm hardware such as a modem or other link device. The four available **options** are:

    1. **Handshake Off, non-modem** connection – most commonly used for 3-wire direct connections to a remote device.

    2. **Full Duplex modem** connection – used with full-duplex modems or equivalent connections.

    3. **Half Duplex modem** connection – used with half-duplex modems or equivalent connections.

    4. **Handshake On, non-modem** connection – used with printers and other similar devices that use the Data Carrier Detect (DCD) and Clear-to-send (CTS) lines to signal the interface card. When DCD is held down by the peripheral, the interface ignores incoming data. When CTS is held down, the interface does not transmit data to the device until CTS is raised.

Options 2 and 3 are usually associated with modems or similar devices, but may be used occasionally with direct connections when the remote device provides the proper signals. Refer to the table at the end of this chapter for a list of handshake signals and how they are handled for each cable or adapter option.

Handling of Non-data Characters

Register 24 specifies what non-data characters are to be included in the input queue. For each bit that is set, the corresponding information is passed along with the incoming data. If the bit is not set, the information is discarded, and is not included in the inbound data stream that is passed to the desktop computer by the interface.

Bit 0: Include handshake characters in data stream. They are defined by Control Registers 26 and 27.

Bit 1: Include incoming end-of-line character(s). EOL characters are defined by Control Registers 28-30.

Bit 2: Include incoming prompt character(s). Prompt is defined by Control Registers 31-33.

Bit 3: Include any null characters encountered.

Bit 4: Include any DEL (rubout) characters in data.

Bit 5: Include any CHR$(255) encountered. This character is encountered ONLY when 8-bit characters are received.

Bit 6: Change any characters received with parity or framing errors to an underscore. If this bit is not set, all inbound characters are transferred exactly as received, with or without errors.

Register 25 is not used.

Protocol Handshake Character Assignment

Registers 26 and 27 establish what characters are to be used for handshaking between communicating machines. You can select the values of 6 (AK) or 17 (DC1) for register 26, and 5 (EQ) or 19 (DC3) for register 27. Any ASCII value from 0 thru 255 can be used, but non-standard values should be reserved for exceptional situations.

End-of-line Recognition

Registers 28, 29, and 30 operate in conjunction with registers 14 (control block mask) and 24 (non-data character stripping) and defines the end-of-line sequence used to identify boundaries between incoming records. Register 28 (value of 0, 1 or 2) defines the number of characters in the sequence, while registers 29 and 30 contain the decimal equivalent of the ASCII characters. If register 28 is set for one character, register 30 is not used. Register 29 contains the first EOL character, and register 30, if used, contains the second. If register 28 is zero, registers 29 and 30 are ignored and the interface cannot recognize line separators.

Prompt Recognition

Registers 31, 32, and 33 operate in conjunction with registers 14 and 24 and define the prompt sequence that identifies a request for data by the remote device. As with end-of-line recognition, the first register defines the number of characters (0, 1, or 2), while the second and third registers contain the decimal equivalents of the prompt character(s). Register 33 is not used with single-character prompts. If register 31 is zero, registers 32 and 33 are ignored and the interface is unable to recognize any incoming prompts.

Character Format Definition

Registers 34 through 37 are used to define the character format for transmitted and incoming data.

- Register 34 sets the character length to 5, 6, 7, or 8 bits. The value used is the number of bits per character minus five (0 = 5 bits, 3 = 8 bits). When 8-bit format is specified, parity must be Odd, Even, or None (parity "1" or "0" cannot be used).

- Register 35 specifies the number of stop bits sent with each character. Values of 0, 1, or 2 are used to select 1, 1.5, or 2 stop bits, respectively.

- Register 36 specifies the parity to be used. Options include:

Register Value	Parity	Result
0	None	Characters are sent with no parity bit. No parity checks are made on incoming data.
1	Odd[1]	Parity bit is set if there is an EVEN number of ones in the character code. Incoming characters are also checked for odd parity.
2	Even[1]	Parity bit is set if there is an ODD number of ones in the character code.
3	0	Parity bit is present, but always zero. No parity checks are made on incoming data.
4	1	Parity bit is present, but always one. No parity checks are made on incoming data.

Parity must be odd, even, or none when 8-bit characters are being transferred.

- Register 37 sets the time gap (in character times, including start, stop, and parity bits) between one character and the next in a transmission. It is usually included to allow a peripheral, such as a teleprinter, to recover at the end of each character and get ready for the next one. A value of zero causes the start bit of a new character to immediately follow the last stop bit of the preceding character.

Control Register 38 is not used.

Break Timing

Register 39 sets the break time (2-255 character times). A Break is a time gap sent to the remote device to signify a change in operating conditions. It is commonly used for various interrupt functions. The interface does not accept values less than 2. Register 6 is used to transmit a break to the remote computer or device.

---

[1] Parity sense is based on the number of ones in the character including the parity bit. An EVEN number of ones in the character, plus the parity bit set produces an ODD parity. An ODD number of ones in the character plus the parity bit set produces an EVEN parity.

Datacomm Options for Data Link Communication

This section explains how to configure the datacomm interface for Data Link operation. The example used shows how to set up configuration options without considering default values. Some statements in the example are redundant because they override interface defaults having the same value. Others may or may not be redundant because they override configuration switch options. The remaining statements are necessary because they override the default values, replacing them with non-default values required for proper operation of the example program. If you are not familiar with Data Link protocol and terminology, consult the section called "Protocol."

The following program lines set up all the CONTROL register options (a 9600-baud connection to an HP 1000 network host is assumed):

```
* 1250   CONTROL Sc,14;6    ! Set Control Block Mask for ETB/ETX,
* 1260   CONTROL Sc,15;0    ! No modem line-change notification,
  1270   CONTROL Sc,16;0    ! Disable Connection timeout,
→ 1280   CONTROL Sc,17;0    ! Disable No Activity timeout,
* 1290   CONTROL Sc,18;40   ! Set Lost Carrier to 400 ms,
* 1300   CONTROL Sc,19;10   ! Set Transmit Timeout=10 s,
→ 1310   CONTROL Sc,20;14   ! Set Line Speed to 9600 baud,
* 1320   CONTROL Sc,21;1    ! Set GID character to "A",
→ 1330   CONTROL Sc,22;1    ! Set DID character to "A",
→ 1340   CONTROL Sc,23;0    ! Hardware Handshake Off for HP 13264A,
* 1350   CONTROL Sc,24;0    ! Set transmit block size to 512,
* 1360   CONTROL Sc,36;0    ! Parity not used with HP 1000,
```

*: Redundant statement. Same as interface default.

→: May be redundant. Overrides configuration switch option.

If your application requires a different GID/DID pair, you can use either of the following two techniques (assume: GID = "C" and DID = "@"):

```
  1320   CONTROL Sc,21;3    ! Set GID character to "C",
  1330   CONTROL Sc,22;0    ! Set DID character to "@",
or
  1320   CONTROL Sc,21;3,0  ! Set GID/DID to "C@",
```

    (Line 1330 is not needed in this case.)

Here is an alternative method using string operations:

```
  1320   CONTROL Sc,21;NUM("C")-64
  1330   CONTROL Sc,22;NUM("@")-64
or
  1320   CONTROL Sc,21;NUM("C")-64,NUM("@")-64
```

Refer to the Control Register tables in the back of the *BASIC Language Reference* manual as you examine the CONTROL statements. The paragraphs which follow explain register functions and how to configure them. When the register function is identical for both Async and Data Link, you are referred to the previous explanation in the Async section.

Control Block Contents

Data Link configuration begins with Control Register 14. This register determines what information is to be placed in control blocks and included with inbound data transferred from the interface to the desktop computer.

- ETX (Bit 1) identifies the end of a transmission block that contains one or more complete records.

- ETB (Bit 2) identifies the end of a transmission block where the last record is continued in the next block of data.

- Bit 0 causes a control block to be inserted that identifies the beginning of a new block of data.

ON INTR Branching Conditions,
Datacomm Line Timeouts, and Line Speed

Registers 15 through 19 are functionally identical for both Async and Data Link. Refer to the preceding Async section for more information. Register 20 sets the line speed for both transmitting and receiving (Data Link does not accommodate split-speed operation). The following line speed options are available:

Register Value	Baud Rate	Register Value	Baud Rate	Register Value	Baud Rate	Register Value	Baud Rate
0	External Clock[1]	9	1200[2]	12	3600	15	19 200[2]
7	300[2]	10	1800	13	4800		
8	600	11	2400	14	9600[2]		

Terminal Identification

Registers 21 and 22 specify the terminal identifier characters for the datacomm interface. Register 21 contains the GID (Group IDentifier), and register 22 contains the DID (Device IDentifier. Values of 0-26 correspond to the characters @, A, B, . . ., Z. These registers must be configured to match the terminal identification pair assigned to your device by the Data Link Network Manager. In the example, Line 1320 is redundant because it duplicates the default GID value. Line 1330 overrides the DID default switch on the interface card, and may or may not be necessary. Alternate methods for assigning different GID/DIDs are shown following the group of configuration CONTROL statements.

Handshake

Register 23 establishes the hardware handshake type. There is no formal software handshake with Data Link because the network host controls all data transfers. Hardware or modem handshake options are identical to Asynchronous operation. Handshake should be OFF (register set to 0) when using the HP 13264A Data Link Adapter. When you are using non-standard interconnections such as direct or modem links to the network host, select the handshake option that fits your application. Refer to the table at the end of this chapter for a list of handshake signals and how they are handled for each cable or adapter option.

---

1 An external clock must be provided for this option.

2 These speeds can be programmed using the default switches on the interface card. Other speeds are accessed by CONTROL statements.

Transmitted Block Size

Register 24 defines the maximum transmitted block length. When transmitting blocks of data to the network host, the block length must not exceed the available buffer space on the receiving device. Block size can be specified for increments of two from 2 to 512 characters per block. A value of zero forces the block length to a maximum of 512 bytes. For other values, the block length limit is twice the value sent to the register. For example, a register value of 130 produces a transmitted block length not exceeding 260 characters (bytes).

Parity

Register 36 defines the parity to be used. Unlike Async, Data Link has only two parity options: None, or Odd. Odd parity is:

Register Value	Parity	Application
0	NONE	Required for operation with HP 1000 network host
1	ODD	Required for operation with HP 3000 network host

Registers 25 through 35, and 37 and above are not used.

Connecting to the Line

Interface configuration is now complete. You are ready to begin connecting to the datacomm line. The exact procedure used to connect to the line varies slightly, depending on the type of link being used. Before you connect, you must know what the link requirements are, including dialing procedures, if any.

Switched (Public) Telephone Links

When you are using a public or switched telecommunications link, the modem connection between computers must be established. The HP 13265A Modem can be used in any Async application that requires a Bell 103- or Bell 113-compatible modem operating at up to 300 baud line speed. However, the HP 13265A Modem is not suitable for data rates exceeding 300 baud. For higher baud rates, use a modem that is compatible with the one at the remote computer site. Modems cannot be used for remote connections from a terminal to the data link.

Private Telecommunications Links

Private (leased) links require modems unless the link is short enough for direct connection (up to 50 feet, depending on line speed). The HP 13265A Modem can be used at data rates up to 300 baud. For higher speeds, a different modem must be used.

Direct Connection Links

For short distances, a direct connection may be used without modems or adapters, provided both machines use compatible interfaces. Async connections normally use RS-232C interfaces. You can also operate as a Data Link terminal directly connected to an HP 1000 or HP 3000 host computer through a dedicated Multipoint Async interface on the network host, although such connections are unusual.

Data Link Connections
Most Data Link connections use an HP 13264A Data Link Adapter to connect directly to the
Data Link. In special situations, a modem may be used to communicate with a Multipoint Async
interface on the HP 1000 or HP 3000 network host. When the Data Link Adapter is used, no
special procedures are required. If you are using a leased or switched telecommunications link,
the procedures are the same as when using point-to-point Async with modems.

Connection Procedure

This section describes procedures for modem connections using telephone telecommunications
circuits. If you are NOT using a switched, modem link, skip to the next section: Initiating the
Connection.

Dialing Procedure for Switched (Public) Modem Links
Except for dialing, connection procedures do not usually vary between switched and dedicated
links. Dialing procedures depend on whether the modem is designed for manual or automatic
dialing. Automatic dialing can be used with the HP 13265A Modem, but other modems must be
operated with manual dialing unless you design your own interface to an Automatic Calling
Unit. For manual dialing procedures, consult the operating manual for the modem you are
using.

Automatic Dialing with the HP 13265A Modem:
The automatic dialer in the HP 13265A Modem is accessed by Control Register 12. The
CONTROL statement is followed by an OUTPUT statement that contains the telephone num-
ber string, including dial rate and timing characters. The two statements set up the automatic
dialer, but dialing is not started until a "start connection" command is sent to Control Register
12. Here is an example sequence:

```
1500   CONTROL Sc,12;2      ! Enable the Automatic Dialer,
1510   OUTPUT Sc;"> 9 @@@ (303)-555-1234";
```

Unrecognized characters are ignored.

3-second wait for secondary dial tone.

Select FAST dial rate.

The OUTPUT statement contains several essential elements.

- The first character (">"), if included, specifies a fast dialing rate. If it is omitted, the default
  slow dialing rate is used.

- A time delay character "@" may be inserted anywhere in the string. A one-second time
  delay is executed in the dialing sequence each time a delay character is encountered.

- Numeric character sequences define the telephone number. Multiple dial-tone sequences,
  such as when calling out from a PBX (Private Branch Exchange), can be used by inserting
  a suitable delay to wait for the next dial tone.

- Unrecognized characters such as parentheses, hyphens, and spaces can be included for
  clarity. They are ignored by the automatic dialer.

- Up to 500 characters can be included in the telephone number string.

Here is how an autodial connection is executed:

- The CONTROL Sc,12;2 statement places a "start dialing" control block in the outbound queue to the interface. The OUTPUT statement places the telephone number string (including spaces and other characters) in the queue after the control block. When the interface encounters the control block, it transfers the string to the HP 13265A Modem's autodial circuit. No other action is taken at this time.

- When a CONTROL Sc,12;1 statement (line 1600 in the example) is executed, another control block is queued up. When the interface encounters the block, it sends a "start connection" command to the modem. The modem then disconnects from the line, waits two seconds, then reconnects. The autodialer waits 500 milliseconds, then starts executing the telephone number string. The string is executed character-by-character in the same sequence as sent by the OUTPUT statement.

- If your application requires more than 500 milliseconds to guarantee a dial tone is present, you can increase the delay by adding delay characters ("@") where needed, one second per character. Be sure to provide adequate delays in multiple dial tone sequences, such as when calling through a private branch exchange (PBX) to a public telephone network.

- When dialing is complete, the modem is connected to the line, and you are ready to start communication. The next section explains how to determine when connection is complete.

Two dialing rates are available: slow (default) and fast. To select the fast rate, you must include the fast rate character (">") as the FIRST character in the telephone number string. Here is a summary of differences between the two options:

Parameter	Slow Dialing	Fast Dialing
Click Length	60 milliseconds	32.5 milliseconds
Click Gap	40 milliseconds	17.5 milliseconds
Number Gap	700 milliseconds	300 milliseconds

One to ten dial pulses (clicks) are sent for each digit 1 through 0, respectively. The number gap is the time lag between the end of the last click of one number and the beginning of the first click of the next number.

Most Bell System facilities can handle both fast and slow dialing rates, but private or independent telephone systems or companies may require slow dialing.

Initiating the Connection

After you have executed the necessary dialing procedures, if any, you are ready to initiate the connection. The following statement is used to start the connection:

```
1600   CONTROL Sc,12;1        ! Start Connection,
```

This statement sends a control block to the interface telling it to connect to the datacomm line. If the HP 13265A Modem is being used, and the autodialer is enabled, it starts dialing the number. Otherwise, the interface executes a direct connection to the line, or tells the modem or data link adapter to connect.

The status of the connection process can be monitored by using the STATUS statement. The following lines hold the computer in a continuous loop until the connection is complete:

```
1650   Conn:STATUS Sc,12;Line_state    ! Get datacomm line status,
1660        IF Line_state=2 THEN DISP "Dialing"
1670        IF Line_state=1 THEN DISP "Trying to Connect"
1680        IF Line_state<>3 THEN Conn
1690        DISP "Connected"
```

Refer to the "Interface Registers" section of the *BASIC Language Reference* for interpretation of the values in Status Register 12. Only values of 1, 2, or 3 are usually encountered at this stage of the program.

As soon as Status Register 12 indicates that connection is complete, you are ready to continue into the main body of the terminal emulator or other program you are writing. This completes the datacomm initialization and connection phase of the program.

Setting up the Interrupt System

Most datacomm programs, especially complex ones, use interrupt branching extensively to maintain efficient, orderly program operation. Branching is usually set up for:

- I/O interrupts from peripheral devices by use of ON INTR and ENABLE INTR statements.
- Datacomm interrupts from the datacomm interface. Statements used are the same as for other I/O interrupts.
- Operator interrupts using softkeys for program control. A separate ON KEY statement is used to set up the branch for each key used.
- Operator interrupts using ASCII keys for program input. The ON KBD statement is used to set up the branch, and KBD$ is the keyboard-entry string holding the data.

Each interrupt branch must be provided with a corresponding interrupt service routine, with priority levels assigned when appropriate. General I/O interrupt techniques are explained in Chapter 7. This section explains the interrupt structures commonly encountered in datacomm applications.

Setting up Softkey Interrupts

Softkeys are usually set up for repetitively executed functions to improve operator convenience and efficiency. Labels can have up to eight or 14 characters for each key, depending on CRT screen width. The following statements add a disconnect and break capability to the emulator example we are using:

```
1750   ON KEY 0 LABEL " Disconn" GOTO Disconnect
1760   ON KEY 1 LABEL "  Break" GOSUB Break
```

Other keys can be set up and labelled as needed, but remember a service routine is required for each label specified by a GOTO, GOSUB, CALL, or RECOVER.

Setting Up Program Operator Inputs

Two methods are commonly used to input information from the operator through the computer keyboard. The first method uses the LINPUT (or INPUT) statement for data entry. An example program using the LINPUT statement is shown in the overview of datacomm programming earlier in this chapter. When the LINPUT statement requests a data entry, type the information, use the keyboard editor to make any necessary corrections, then press CONTINUE to transfer the information to the running program. This is the simplest method for programming keyboard entry. The second method is used in our ongoing example. It uses the ON KBD statement in conjunction with an interrupt service routine that is responsible for all data manipulation, including display, editing, and transfer to the program. The following statement sets up the keyboard interrupt. The interrupt service routine is discussed later.

```
1770  ON KBD GOSUB Keyboard
```

Setting Up Datacomm Interrupts

The ON INTR and ENABLE INTR statements are used to set up program branching for the datacomm interface. STATUS Register 4 contains information that shows the cause(s) of the most recent interrupt. The interrupt mask specified in the ENABLE INTR statement determines the events that are allowed to cause an interrupt branch. Bits 0 thru 5 of the interrupt mask and STATUS register are identical for both Async and Data Link protocols. Bits 6 and 7 are used for Async only.

The following statements set up the interrupt structure for datacomm:

```
1810  ON INTR Sc GOSUB Datacomm
1820  ENABLE INTR Sc;1           ! Interrupt when data received.
```

In more elaborate applications, you may want to enable additional interrupt causes by changing the interrupt mask. Here are the available interrupt bits and their functions:

Interrupt Mask Bits for Async Operation

Bit	Value	Function	Bit	Value	Function
0	1	Data in Receive Queue	4	16	No Activity Timeout
1	2	Prompt Received	5	32	Lost Carrier Timeout
2	4	Framing/Parity Error	6	64	End-of-line Received
3	8	Modem Line Change	7	128	Break Received

Interrupt Mask Bits for Data Link Operation

Bit	Value	Function	Bit	Value	Function
0	1	Data in Receive Queue	3	8	Modem Line Change
1	2	Block Successfully Sent	4	16	No Activity Timeout
2	4	Transmit or Receive Error	5	32	Lost Carrier Timeout

Interrupt mask bits 6 and 7 are not used with Data Link protocol.

To construct the interrupt mask value, add the bit values for each function that is to cause an interrupt. For example, to interrupt when there is data in the receive queue (bit value = 1), or a modem line change (bit value = 8) or a Lost Carrier timeout (bit value = 32), the interrupt mask becomes: 1 + 8 + 32 = 41. The ENABLE INTR statement becomes:

```
1820   ENABLE INTR Sc;41
```

Background Program Routines

After the interrupt structures have been established by the running program, the program begins executing a "background" routine while it waits for interrupts. Background routines vary according to application, and can consist of anything from a simple idle loop to a very complex program. They are called background programs or background routines because their execution is generally suspended whenever interrupts from previously defined sources are received. See Chapter 7 for more discussion of interrupt and software priority.

Background program operations can affect interrupt handling under certain conditions. For example, if the background program contains a subprogram call, the interrupt service routines are temporarily suspended until subprogram execution is complete if the ON INTR statements use GOSUB, or GOTO. Incoming data is held in the receive queue during subprogram execution, and the remote is held off by the interface when the queue is full, if handshaking between devices is active. If handshaking is not being used in Async operation, buffer overflow can occur. When handshake is being used, be sure that the remote computer does not disable the link when extended hold-offs occur.

When interrupt service routines are subprograms accessed by an ON INTR...CALL statement, background subprograms may be temporarily suspended to allow interrupt processing. Be careful when using subprograms to be sure that variables are properly used for orderly flow of information between contexts.

Most BASIC programmers, to maintain clarity in program flow, place interrupt service routines after the background routines. This technique simplifies documentation and makes it easier for others to understand program operation. The location of subroutines or program labels in BASIC programs does not affect efficiency or speed of execution by the desktop computer.

A detailed discussion of background programs is beyond the scope of this chapter because they are dependent upon the individual application. In the example shown in this chapter, a simple idle loop is sufficient. A typical idle loop resembles the following statement:

```
1880   Background: GOTO Background   ! Background program idle loop.
```

The next topics addressed are interrupt service routines for datacomm and keyboard operations.

Interrupt Service Routines

Interrupt service routines are required to service any peripheral device or interface that uses interrupt to access the computer. In the example we are using, interrupt service routines are required for the datacomm interface, computer keyboard, and softkeys. Each routine is treated separately in this section.

Servicing Datacomm Interrupts

Whenever the datacomm interface interrupts a running BASIC program, the interrupt request is first logged and then DISABLE INTR is automatically executed by the system. The cause of interrupt is then placed in STATUS Register 4. The interrupt service routine must do several things to guarantee that: (1) the interrupt is properly handled, (2) the interrupt structure is restored after the current interrupt is acknowledged, and (3) no data is left in the receive queue after the last interrupt request is processed. The following items outline the basic elements of the datacomm interrupt service routine (similar techniques are used for other interfaces).

- Read STATUS Register 4 to clear the interrupt request and determine the cause of the interrupt. If you do not clear the interrupt request, it remains active and a new interrupt is generated as soon as you exit the service routine, whether or not there is any information to process.
- Use ENABLE INTR (usually without specifying a new interrupt mask) to reactivate the datacomm interrupt system. It is usually unnecessary to redefine the interrupt mask when this is done.
- Take appropriate action based on what caused the interrupt.
- Exit the interrupt service routine with a RETURN (or equivalent statement as appropriate) taking care to maintain proper program structure.

In most applications, interrupts are generated when data is available for transfer between the interface and your desktop computer. The interrupt service routine then processes the transfer using the ENTER statement. Here is an example of a typical datacomm interrupt service routine where A$ is dimensioned to a length of one character (DIM A$[1]). The calling sequence might be:

```
ON INTR Sc GOSUB Datacomm
ENABLE INTR Sc;Mask

2090   Datacomm:STATUS Sc,4;Interrupt_cause
2100          ENABLE INTR Sc
2110   Dc:    STATUS Sc,5;Rx_queue_status
2120          IF Rx_queue_status=0 THEN RETURN
2130          ENTER Sc USING "#,-K";A$
2140          PRINT USING "#,K";A$
2150          GOTO Dc
```

While this **interrupt service routine** (ISR) looks deceptively simple, its structure performs several important functions:

- Line 2090 acknowledges the interrupt and places the cause-of-interrupt information in Interrupt_cause.
- Line 2100 reenables the interrupt without changing the mask.

- Line 2110 gets the receive queue status. Four values are possible:

Rx_queue_status = 0: Receive queue is empty.

Rx_queue_status = 1: Receive queue contains data.

Rx_queue_status = 2: Receive queue contains at least one control block.

Rx_queue_status = 3: Receive queue contains both data and at least one control block.

- Line 2120 checks to make sure there is data or control information available before continuing. This prevents attempts to enter data that does not exist. The placement of this statement is explained under Exit Conditions.

- Line 2130 enters the data. The format used guarantees that no data is lost during searches for end-of-line delimiters. The "#" IMAGE specifier prevents search for end-of-line (EOL) delimiters. Use of " − K" places CR, LF, and CR-LF end-of-line delimiters in the string variable when they are encountered. BASIC can then locate the delimiters by using separate operations.

- Line 2140 prints the data on the current PRINTER IS device. The "#" specifier suppresses the EOL sequence because terminators are already contained in the string variable.

- Line 2150 goes back to check for more data before exiting. This guarantees that no data is missed in the event that additional data arrives during interrupt service. Otherwise, some interrupt requests may be missed.

To understand why the interrupt is handled as shown, consider the following sequence of events:

Interface places data in queue and requests interrupt.

Interface receives more data and requests a second interrupt.

Interface requests a third interrupt.

t0   t1   t2   t3   t4   t5

ISR begins processing second interrupt.

ISR finishes first interrupt.

ISR acknowledges then reenables interrupt.
It then begins entering the data.
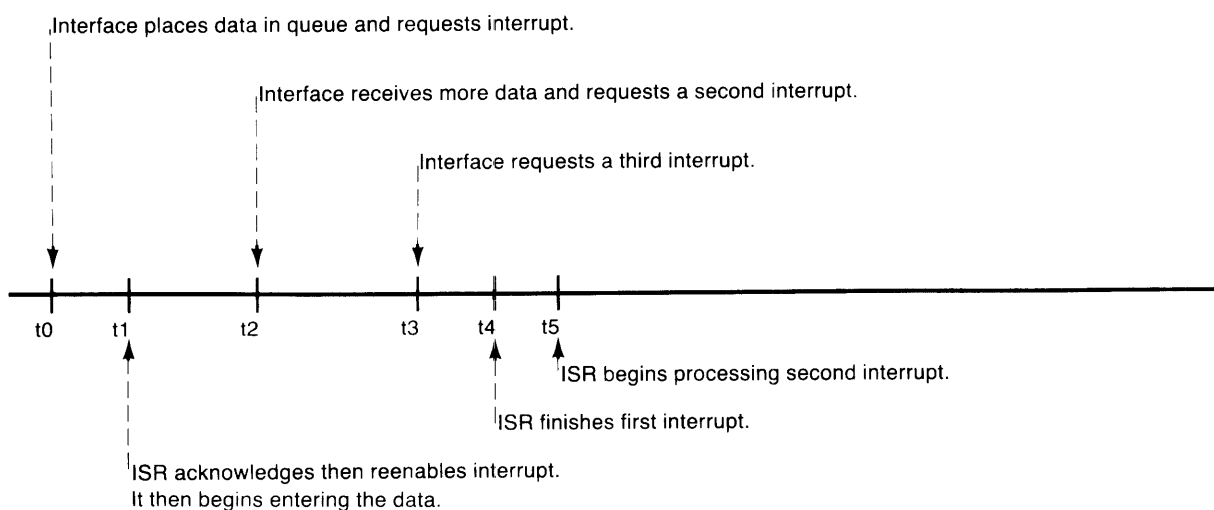
At time t0, the interface places data in the receive queue and requests interrupt service. At t1, the ISR responds and acknowledges the interrupt. The interrupt is reenabled, but subsequent interrupt service requests are logged but not serviced until the routine is finished. While the ISR is processing the first interrupt request, a second and third request are made at t2 and t3. (The already active interrupt request line is reactivated by the third request. From the computer's point of view, nothing happened because the second interrupt request was already active). When the ISR completes the first interrupt process (t4), it exits, then acknowledges, the second interrupt (t5).

Here is what really happens when the example routine is executed: Since the routine checks for no more data in the queue before it processes the interrupt, and remains in the ENTER/PRINT loop until the queue is empty, all available information is processed before exit occurs. Therefore, data placed in the queue at the time of the second and third interrupt requests is processed before the exit at t4, guaranteeing that nothing is left. When the second entry is made to the routine (t5) in response to the second interrupt request, no data is in the queue unless it was placed there between exit and reentry. In this case, the queue is empty, so exit is immediate. The third interrupt request cannot be recognized, because the second was still pending when it occurred.

If the routine were written differently, and only one ENTER statement was executed for each interrupt request, the example sequence would result in only two interrupts being acknowledged. The third interrupt request and its corresponding data would not be processed until a fourth request caused the third data entry to be executed. Such a structure presents a risk of data loss.

Exit Conditions
In the preceding example, line 2120 exits or continues the interrupt service routine, depending on the status of the receive queue. The example shown assumes that A$ can hold only a single ASCII character or data byte. The ENTER statement is terminated as soon as A$ is filled, so data transfer is one byte at a time. By checking for Status Register 5 = 0, you are guaranteed that no data messages remain in the receive queue. Control blocks are immaterial in this case.

When using Data Link protocol, most programmers specify data transfer formats of one record per block. This eliminates the need to search data for delimiters[1]. Since the datacomm interface can receive Data Link transmission blocks up to 1000 characters, it is wise to dimension A$ to a length exceeding the maximum expected block length; for example, D I M A $ [1 0 5 0]. In such cases, it is necessary to modify line 2120 to provide exit if a full block is not available for A$. Instead of examining for the presence of data, a test is made to look for a control block in the queue, indicating the presence of a full block of data. (Control Register 14 must be set so that only ETB/ETX terminators are allowed to create a control block.) If a control block is present, a full block of data is also available. When the ENTER statement is executed, the input operation terminates when the control block is encountered, and the resulting length of A$ matches the received block length. To operate in "block mode" instead of "character mode" as earlier, change line 2120 to:

```
2120        IF Rx_avail_bits<2 THEN RETURN
```

Only the dimension of A$ is affected by this change. Other interrupt service routine statements remain unchanged.

---

1 The HP 3000 packs multiple records per block when transferring ASCII text files, so you must decode delimiters to find record boundaries. Consult the appropriate HP 3000 Data Link manuals for more information.

Note

It is good programming practice to be sure the receive queue or input buffer is completely empty before exiting an interrupt service routine, and make sure there is data present before trying to process it.

This example datacomm interrupt service routine is adequate for most applications where data is not sent with a known, fixed format, and where prevention of data loss is important. In other situations, where loss of data between the end of the input variables list and the delimiter in incoming data is unimportant, or a fixed format is used, other formats can be specified. It is usually wise to avoid using multiple variables with the ENTER statement when using the formats shown in this example. Here's why:

A control block indicates End-of-data, not End-of-information. Consequently, an ENTER statement is terminated whenever a control block is encountered (variables are terminated by EOI, not EOD). If more than one variable is included in the statement, and EOD (control block) occurs before the list is filled, the unfilled variables retain their previous values which can lead to improper results.

Data Formats for Datacomm Transfers

All datacomm data transfers use the OUTPUT and ENTER statements. Consequently, any formatting techniques that are compatible with these statements can also be used. However, since most computers send and expect to receive a limited variety of data formats, most data transfers use a limited assortment of formats.

ASCII Data Transfers – In asynchronous data communications applications, information is usually transferred as lines of ASCII text. In most cases, lines are terminated by a carriage-return followed by a line-feed (CR-LF), or by a carriage-return only. Other methods may be used occasionally to recognize record boundaries in special applications.

Most Data Link applications consist of ASCII text records transferred between the network host computer and other terminals and/or computers in the network. Records are transmitted in blocks, one or more records per block. When multiple-record blocks are transferred, delimiters between records are included as part of the text, and individual records must be unpacked by the receiver.

Non-ASCII Data Transfers – Non-ASCII data includes non-text or non-ASCII text data that must be transmitted over the datacomm link, but may contain characters that could be interpreted as datacomm control characters. Examples of non-ASCII data includes encoded data files, non-text program files, or specially formatted data. To provide a means of transferring non-ASCII data formats requires non-standard techniques in Async, and transparent transmission when using Data Link.

To transfer non-ASCII data using asynchronous protocol, use an eight-bit character format with or without parity as dictated by your application. End-of-line and prompt recognition, and any character stripping functions must be disabled to allow passage of arbitrary character patterns. Use of Async for such applications is uncommon, primarily because of the limited reliability of parity checks as a means for error detection.

Transfer of non-ASCII data using Data Link protocol is much easier because all data transmitted by the desktop computer through the datacomm interface is sent as transparent data; i. e., data that could be mistaken for control characters is transferred intact. Data Link transfers from the network host are also sent as transparent data. In order to transfer non-ASCII data from the network host, a cooperating program on the host must originate the data, and suppress end-of-line and other unwanted character sequences.

Servicing Keyboard Interrupts

The keyboard interrupt service routine has several functions. In the case of a terminal emulator or similar application, it inputs keystrokes, interprets them, then transmits the results to the datacomm interface. In addition, it may be required to display the keystroke(s) or perform backspace and editing operations (such as in line-mode terminal emulators). Certain keys may also be reserved to perform program command functions while others are used to transmit information to the host.

Here is a simple example of a keyboard interrupt service routine that sends ASCII keystrokes to the datacomm interface as each key is pressed, then sends an end-of-line (CR) if Async, or end-of-block if Data Link. The example shown is for Async protocol; Line 2410 is changed for Data Link. The calling sequence might be ON KBD GOSUB Keyboard. An explanation follows the example.

```
2290    Keyboard:K$=KBD$
2300    K:    IF NOT LEN(K$) THEN RETURN
2310          Key=NUM(K$)
2320          K$=K$[2]
2330          IF Key=255 THEN
2340              Key=NUM(K$)
2350              K$=K[2]
2360              IF Key=255 THEN
2370                  Key=NUM(K$)
2380                  K$=K$[2]
2390              END IF
2400              IF Key=NUM("E") THEN
2410                  OUTPUT Sc;CHR$(13);END
2420              ELSE
2430                  BEEP
2440              END IF
2450          ELSE
2460              OUTPUT Sc;CHR$(Key);
2470          END IF
2480          GOTO K
```

To change the example for Data Link, eliminate the carriage return in line 2410 as follows:

```
2410                    OUTPUT Sc;END
```

This Async example assumes that the host echoes any data sent to it; that is, when a character is sent to the host, the host sends the same character back to the terminal where it is displayed. Consequently, keystrokes are displayed AFTER they are returned by the host. Data Link protocol does not provide this feature (called echo-plex). To print each keystroke on the CRT as it is keyed in, add the following line to the Data Link example:

```
2465                    PRINT CHR$(Key);
```

This keyboard routine is a good illustration of how to use an IF...THEN...ELSE structure to decode a keystroke, and decide whether it is ASCII, end-of-line, or an unrecognized character. If ASCII, it is transmitted. If the ENTER key is pressed, it sends an EOL. Any other key is ignored, but the computer beeps to acknowledge the keystroke.

To understand the routine, you must be aware that several data formats are found in KBD$. ASCII keystrokes are stored, one byte per stroke, as key codes equivalent in value to the NUM value of the corresponding ASCII character code. Non-ASCII keys are stored as two bytes; the first byte is CHR$(255), the second byte is the keycode. If the CONTROL key is pressed simultaneously with a non-ASCII key, a three-byte entry is made in KBD$. The first is CHR$(255) representing a non-ASCII key, the second is also CHR$(255) representing the CONTROL key, and the third byte is the keystroke. Keycode values for non-ASCII keys are listed in the Keyboard Output Codes table in the back of the BASIC Language Reference for your computer. The following table shows the KBD$ data format for each keystroke:

Keystroke(s)	First Byte	Second Byte	Third Byte
ASCII or CONTROL-ASCII	ASCII keycode	None	None
Non-ASCII Key	CHR$(255)	Non-ASCII keycode	None
CONTROL-Non-ASCII Key	CHR$(255)	CHR$(255)	Non-ASCII keycode

The contents of KBD$ is destroyed when you transfer it to another string or perform any other operation on KBD$. Since only one read from KBD$ is possible, K$ is used as a temporary storage and work area for the contents of KBD$, permitting additional string operations.

The first IF...THEN...ELSE looks for a CHR$(255) indicating a non-ASCII key. If none is found, the ASCII key is sent to the datacomm interface. The second IF...THEN...ELSE is entered ONLY if the first character indicates a non-ASCII key. It looks for a second CHR$(255), which is discarded, if found. (Both ENTER and CTRL-ENTER are accepted as end-of-line.) The keystroke data byte is then checked to see if it is the ENTER key. If the value is not equivalent to NUM("E"), the key is rejected. Otherwise, and end-of-line/end-of-block is sent to the datacomm interface.

In more elaborate applications, other keys such as backspace or other cursor control characters could be interpreted, and the CRT display and other program parameters varied accordingly.

Note that the interrupt service routine remains active until the entire contents of KBD$ as it existed at time of interrupt is processed. If, in the meantime, more keystrokes are placed in KBD$, a new interrupt occurs as soon as the service routine is finished.

Service Routines for ON KEY Interrupts

ON KEY interrupt service routines are usually simpler than ON KBD service routines. The tasks are usually well-defined and relatively simple. In this example, KEY 0 disconnects the datacomm line, and KEY 1 sends a BREAK. The routines are implemented as follows:

To send a BREAK on either Async or Data Link, set bit zero of Control Register 6. Here is how:

```
2520    Break:CONTROL  Sc,6;1
2530         RETURN
```

To disconnect from the datacomm line, clear Control Register 12 as follows:

```
2570    Disco:CONTROL Sc,12;0
2580          DISP "Disconnected"
2590          END
```

You now have a working terminal emulator.

Cooperating Programs

Some applications, while similar in some respects to terminal emulators, require unattended operation of the desktop computer and network host. In such cases, cooperating programs on the host and terminal computer are used. Applications can include such things as the desktop computer controlling a local data gathering system, making preliminary calculations, and sending the results to the network host. Since data integrity is important in such cases, Data Link is frequently used because of its ability to detect transmission errors.

Here is an example of cooperating programs you can run on your desktop computer and an HP 1000 Data Link network host computer. The FORTRAN program COOP runs on the HP 1000, and is responsible for opening and transferring the specified file(s) from the HP 1000 to the Data Link. A cooperating BASIC program on the desktop computer acts as an interface between the operator and the HP 1000. The specified file is transferred from the Data Link to local mass storage as it is received from the HP 1000. Assuming the file is an ASCII program file containing valid BASIC statements, it can then be attached to the cooperating program and run. Note that variables used by both the original BASIC program and the downloaded program must be specified as COM variables to prevent destroying their values during pre-RUN initialization of the downloaded program. The program listings are as follows:

FORTRAN Program COOP for the HP 1000:

```
FTN4,L
        PROGRAM COOP

C   This is a FORTRAN program that runs on the HP 1000 and cooperates
C   with a compatible program running simultaneously on a desktop
C   computer.
C
C   This program waits in I/O suspend until the desktop computer returns
C   a file name.  When the name is received, it is parsed, and the
C   success status of the parse is sent to the desktop computer.  If the
C   file name parses successfully, this program tries to open the file.
C   The status of the OPEN is also sent to the desktop computer.
C
        INTEGER DCB(144),IDBUF(10),IBUF(80)
        INTEGER NAME(3),SCODE,CRN
        INTEGER DTC,ERROR,OK
        EQUIVALENCE (NAME,IDBUF),(SCODE,IDBUF(5)),(CRN,IDBUF(6))

C   ***INITIALIZE DTC TO BE THE LU# OF THE DESKTOP COMPUTER***

        DTC=21

C   ***Send the ASCII string "SYNCHRONIZE" to the desktop computer***
C       This signals the desktop computer to begin executing the sister
C       program to this one.
```

```
      CALL EXEC(2,DTC,11HSYNCHRONIZE,-11)

C  ***Now wait in I/O suspend until the desktop computer sends the***
C     name of the program file that is to be downloaded to the
C     desktop computer.

      CALL EXEC(1,DTC,IBUF,-40)
      CALL ABREG(IA,LEN)
      IP=1
      IF ( NAMR (IDBUF,IBUF,LEN,IP) ) 9200,100
100   CALL EXEC (2,DTC,2HOK,-2)

C  ***OPEN THE FILE AND SEND THE CONTENTS TO THE DESKTOP COMPUTER***

      IF ( OPEN (DCB,ERROR,NAME,0,SCODE,CRN)    ) 9100,200
200   CALL EXEC (2,DTC,2HOK,-2)

250   CALL READF(DCB,ERROR,IBUF,80,LENGTH)
      IF (LENGTH,EQ,-1) GOTO 300
      CALL EXEC (2,DTC,IBUF,LENGTH)
      GOTO 250

C  ***TELL THE DESKTOP COMPUTER THAT THE END OF FILE HAS BEEN***
C     REACHED, THEN STOP.

300   CALL EXEC(2,DTC,11H*ENDOFFILE*,-11

      STOP

C  ***********************************************************************
C                    ERROR HANDLING ROUTINES
C  ***********************************************************************

C  ************THIS ROUTINE HANDLES DISC ERRORS*********************
C        BY SENDING THE FMP ERROR AND CLOSING THE FILE.

9100  WRITE(DTC,9101)ERROR
9101   FORMAT ("THE OPEN FMP ERROR CODE WAS "I6)
      CALL CLOSE(DCB)
      STOP

C  ************THIS ROUTINE HANDLES PARSING ERRORS******************

9200  WRITE(DTC,9201)
9201   FORMAT ("THE FILE NAME RECEIVED DID NOT PARSE CORRECTLY")
      STOP
      END
```

Cooperating BASIC Program for the Desktop Computer:

```
1000   ! ************************************************************************
1010   ! This BASIC program cooperates with the FORTRAN program "COOP" and
1020   ! downloads a BASIC program file from the HP 1000 for execution on
1030   ! the desktop computer.  While the program is not elegant, it
1040   ! illustrates the basic concepts involved in downloading files to
1050   ! local mass storage, then loading them into memory for execution.
1060   ! The same technique is useful for transferring data files.
1070   !
1080   ! ************************************************************************
1090   !
1100      COM Sc,Insep$[4],Prompt$[2]   ! The values of these variables must be
1110                                     ! preserved between programs.
1120      Sc=20                         ! Set select code.
1130      DIM Rx$[1050],Tx$[1050]       ! Set up data transfer strings.
1140      Insep$=CHR$(13)&CHR$(10)&CHR$(27)&"_" ! HP 1000 EOL string.
1150      Esc_u_score$=CHR$(27)&"_"              ! Escape-Underscore.
1160      INTEGER A
1170   !
1180   ! ************************************************************************
1190   ! Set up DATA LINK protocol
1200   !
1210            CONTROL Sc,0;1         ! Reset the interface.
1220            CONTROL Sc,3;2         ! Set Data Link protocol.
1230 Wait:      STATUS Sc,38;All_sent
1240            IF NOT All_sent THEN Wait    ! Wait for control block sent.
1250            CONTROL Sc,0;1         ! Reset interface to start new protocol.
1260   !
1270   ! ************************************************************************
1280   ! Set up the datacomm configuration.
1290   !
1300            CONTROL Sc,16;0        ! Disable Connect timeout.
1310            CONTROL Sc,17;0        ! Disable No Activity timeout.
1320            CONTROL Sc,20;14       ! Set baud rate to 9600.
1330            CONTROL Sc,21;1        ! GID="A".
1340            CONTROL Sc,22;1        ! DID="A".
1350            CONTROL Sc,23;0        ! Override default switches and set
1360                             ! Hardware Handshake OFF, non-modem connection.
1370            CONTROL Sc,24;0        ! Transmit block length maximum: 512 bytes.
1380            CONTROL Sc,36;0        ! Set parity: NONE (HP 1000 connection).
1390   !
1400   ! ************************************************************************
1410   ! Connect to the Data Link.
1420   !
1430            CONTROL Sc,12;1        ! Send connection command to the interface.
1440            DISP "Trying to connect"
1450 Conn:      STATUS Sc,12;Line_state
1460              IF Line_state<>3 THEN Conn ! Wait for connection complete.
1470              DISP "Connected"
1480   !
1490   ! ************************************************************************
1500   ! This is a MINIMAL Terminal Emulator.
1510   !
1520 Prompt:    LINPUT Tx$                  ! Get line to send to network host.
1530            PRINT USING "#,K";Tx$       ! Print line on CRT.
1540            OUTPUT Sc;Tx$;END           ! Send line to host.
1550   !
1560 Idle:      STATUS Sc,5;Receive         ! Look for reply from host.
1570            IF NOT Receive THEN Idle    ! If nothing, try again.
1580   !
1590            ENTER Sc USING "#,-K";Rx$   ! Get reply message.
1600            PRINT USING "#,K";Rx$[1,POS(Rx$,Esc_u_score$)-1]   ! Print reply.
1610   !
1620   ! Trap messages from HP-1000:
1630   !
1640            IF POS(Rx$,"UNABLE TO COMPLETE LOG-ON") THEN Prompt  ! If error,
1650            IF POS(Rx$,"END OF SESSION") THEN Prompt        ! try again.
1660            IF POS(Rx$,"SYNCHRONIZE") THEN Coop    ! When synchronized, start.
```

```
1670  !
1680            STATUS Sc,5;Receive   ! Look for line with EOL characters missing.
1690                                  ! If not CrLfEsc_, it is a system or sub-
1700                                  ! system prompt from the HP 1000. Otherwise,
1710                                  ! go to idle loop.
1720            IF NOT Receive AND (POS(Rx$,Insep$)=0) THEN Prompt   ! Prompt?
1730            GOTO Idle                                            ! No.
1740  !
1750  ! ************************************************************************
1760  ! This section starts the cooperating program.
1770  !
1780 Coop:      LINPUT "TYPE IN A FILE NAME",Tx$   ! Get file name for transfer.
1790 T1:        STATUS Sc,4;Transmit              ! Get transmit queue status.
1800            IF NOT BIT(1,Transmit) THEN T1    ! If not empty, wait.
1810               OUTPUT Sc;Tx$;END              ! Send file name.
1820  !
1830 R1:        STATUS Sc,5;Receive              ! Get receive queue status.
1840            IF NOT Receive THEN R1            ! If empty, wait for data.
1850               ENTER Sc USING "#,-K";Rx$      ! Get data. Keep CR-LF.
1860               IF POS(Rx$,"OK") THEN R2       ! If OK, continue.
1870                  PRINT Rx$                   ! Not OK. Print error message.
1880                  STOP                        ! Error. STOP.
1890  !
1900 R2:        STATUS Sc,5;Receive              ! Look for another OK from
1910            IF NOT Receive THEN R2            ! the HP 1000.
1920               ENTER Sc USING "#,-K";Rx$
1930               IF POS(Rx$,"OK") THEN Rd_prog  ! If OK, start download.
1940                  PRINT Rx$                   ! Not OK. Print error message.
1950                  STOP                        ! Error. STOP.
1960  !
1970  ! ************************************************************************
1980  ! For this section to work, the HP 1000 must send the 4-character
1990  ! end-of-line sequence: CR-LF followed by escape-code, underscore.
2000  ! Auto-answer must be disabled, and the data being sent from the
2010  ! HP 1000 MUST consist of valid BASIC program lines, each including a
2020  ! valid line number.
2030  !
2040 Rd_prog:   ! ASSIGN @File TO "DOWNLOAD"      ! Assign destination file for
2050  !                                            file transfer.
2060 R3:        STATUS Sc,5;Receive              ! Look for data record.
2070            IF NOT Receive THEN R3            ! If nothing, wait for record.
2080               ENTER Sc USING "#,-K";Rx$      ! Get record. Keep CR-LF.
2090               PRINT Rx$                      ! Print record on printer.
2100               IF POS(Rx$,"*ENDOFFILE*") THEN Get_prog !Check for end-of-file.
2110               OUTPUT @File;Rx$,[1,POS(Rx$,Esc_u_score)-1] ! Store record on
2120               GOTO R3          ! Mass Storage file and repeat for next record.
2130  !
2140 Get_prog:  !  File has been downloaded to local mass storage.
2150            ASSIGN @File TO *           ! Close the file.
2160            GET "DOWNLOAD",2200,2200    ! Get the downloaded program.
2170  !
2200            END  ! This statement is destroyed by GET.
```

Program File to be Downloaded from the HP 1000:

```
1000  !        This program is downloaded to the desktop computer for execution.
1010  !
1020           DIM A$[20]
1030           INPUT "HI. I'm the downloaded program.  What is your name?",A$
1040           PRINT "Now I'll count to 10."
1050           FOR I=1 TO 10
1060              PRINT "                     :";I
1070           NEXT I
1080           PRINT "That's the end of the demo!!"
1090           PRINT "Nice to meet you, ";A$
1100           GOTO Idle
1110           END
```

Modified Cooperating BASIC Program After Loading:

```
2080                ENTER Sc USING "#,-K";Rx$        ! Get record.  Keep CR-LF.
2090                PRINT Rx$                        ! Print record on printer.
2100                IF POS(Rx$,"*ENDOFFILE*") THEN Get_prog !Check for end-of-file.
2110                OUTPUT @File;Rx$,[1,POS(Rx$,Esc_u_score)-1] ! Store record on
2120                GOTO R3           ! Mass Storage file and repeat for next record.
2130    !
2140 Get_prog:      !   File has been downloaded to local mass storage.  GET it.
2150                ASSIGN @File TO *               ! Close the downloaded file first.
2160                GET "DOWNLOAD",2200,2200     ! Get the downloaded program.
2170    !
2200    !           This program is downloaded to the desktop computer for execution.
2210    !
2220                DIM A$[20]
2230                INPUT "HI.  I'm the downloaded program.  What is your name?",A$
2240                PRINT "Now I'll count to 10."
2250                FOR I=1 TO 10
2260                    PRINT "                            :";I
2270                NEXT I
2280                PRINT "That's the end of the demo!!"
2290                PRINT "Nice to meet you, ";A$
2300                GOTO Idle
2310                END
```

Results:

Assuming you have logged onto the HP 1000, the printed output that is displayed on the CRT screen or current PRINTER IS device should look something like this:

```
:
RU,COOP
SYNCHRONIZE
TYPE IN A FILE NAME
FAB2::10
HI, I'm the downloaded program.  What is your name?
SUE
Now I'll count to 10
                :1
                :2
                :3
                :4
                :5
                :6
                :7
                :8
                :9
                :10
That's the end of the demo!!
Nice to meet you SUE
   COOP  : STOP  0000
:
EX
$END FMGR
FMG21 REMOVED

SESSION   21  OFF  1:26 PM  FRI.,  11  SEP.,  1981
CONNECT TIME:            00 HRS.,  08 MIN.,  28 SEC.
CPU USAGE               00 HRS.,  00 MIN.,  00 SEC.,    470 MS.
CUMULATIVE CONNECT TIME  01 HRS.,  09 MIN.,  02 SEC.
END OF SESSION
```

Datacomm Errors and Recovery Procedures

Several errors can be encountered during datacomm operation. They are listed here with probable causes and suggested corrective action.

Error	Description and Probable Cause
306	Interface card failure. This error occurs during interface self-test, and indicates an inteface card hardware malfunction. You can repeat the power-up self-test by pressing (SHIFT) (PAUSE) (or (RESET)). If the error persists, replace the defective card. Using a defective card may result in improper datacomm operation, and should be considered only as a last resort.
313	USART receive buffer overflow. The SIO buffer is not being cleared fast enough to keep up with incoming data. This error is uncommon, and is usually caused by excessive processing demands on the interface microprocessor. To correct the problem, examine BASIC program flow to reduce interference with normal interface operation. This error causes the interface to disconnect from the datacomm line and go into a SUSPENDED state. Clear or reset the interface card to recover.
314	Receive Buffer overflow. Data is not being consumed fast enough by the desktop computer. Consequently, the buffer has filled up causing data loss. This is usually caused by excessive program demands on the desktop computer CPU, or by poor program structure that does not allow the desktop computer to properly service incoming data when it arrives. Modify the BASIC program(s) to allow more frequent interrupt processing by the desktop computer, or change to a lower baud rate and/or use protocol handshaking to hold off incoming data until you are ready to receive it. This error causes the interface to disconnect from the datacomm line and go into a SUSPENDED state. Clear or reset the interface to recover.
315	Missing Clock. A transmit timeout has occurred because the transmit clock has not allowed the card to transmit for a specified time limit (Control Register 19). This error can occur when the transmit speed is 0 (external clock), and no external clock is provided, or be caused by a malfunction. The interface is disconnected from the datacomm line and is SUSPENDED. To recover, correct the cause, then reset the card.
316	CTS false too long. Due to clear-to-send being false on a half-duplex line, the interface card was unable to transmit for a specified time limit (Control Register 19). The card has disconnected from the datacomm line, and is in a SUSPENDED state. To recover, determine what has caused the problem, correct it, then reset or clear the interface card.
317	Lost Carrier disconnect. Data Set Ready (DSR) (and/or Data Carrier Detect, if full-duplex) went inactive for the specified time limit (Control Register 18). This condition is usually caused by the telecommunications link or associated equipment. The card has disconnected from the datacomm line and is in a SUSPENDED state. To recover, clear or reset the interface card.
318	No Activity Disconnect. The interface card disconnected from the datacomm line automatically because no information was transmitted or received within the time limit specified by Control Register 17. The card is in a SUSPENDED state. Clear or reset the interface to recover.
319	Connection not established. The card attempted to establish connection, but Data Set Ready (DSR) (and Data Carrier Detect, if full duplex) was not active within the time limit specified by Control Register 16. The card has disconnected from the datacomm line and is in a SUSPENDED state. Clear or reset the interface to recover.

Error	Description and Probable Cause
325	Illegal DATABITS/PARITY combination. CONTROL statements have attempted to program 8 bits per character and parity "1" or "0". The CONTROL statement causing the error is ignored, and the previous setting remains unchanged. To correct the problem, change the CONTROL statement(s) and/or interface default switch settings.
326	Register address out of range. A CONTROL or STATUS statement has attempted to address a non-existing register. The command is ignored, and the interface card state remains unchanged. This error can also occur when illegal HP-IB statements are used with this interface.
327	Register value out of range. A CONTROL command attempted to place an illegal value in a defined register. The command is ignored, and the interface card state remains unchanged.

Error Recovery

When any error from Error 313 through Error 319 occurs, it forces the interface card to disconnect from the datacomm line. When a forced disconnect terminates the connection, the interface is placed in a SUSPENDED state, indicated by Status Register 12 returning a value of 4. The interface cannot be reconnected to the datacomm line when it is SUSPENDED. CLEAR, ABORT, and RESET are used to recover from the suspended state and resume normal card operation. Executing OUTPUT and CONTROL statements while the card is suspended places corresponding data and control block(s) in the transmit (outbound) queue and can continue to do so until the queue is filled, at which time the desktop computer operating system hangs. ENTER statements can be executed to retrieve data that was there prior to SUSPEND until the receive (inbound) queue is empty. Subsequent ENTER statements, if executed while the card is suspended, hang the computer.

To recover from a SUSPENDED interface, three programmable options are available, all of which destroy any existing data in the transmit and receive queues. They are:

- The CLEAR statement clears the receive and transmit queues. In addition, if the interface card is suspended, it disconnects the card from the datacomm line. If the card is not suspended, its connection state is not changed, but the queues are cleared.
- The ABORT statement is identical to the CLEAR statement, except that the interface card is unconditionally disconnected from the datacomm line.
- RESET interface (Control Register 0) clears all buffers and queues, and resets all CONTROL options to their power-up state EXCEPT the protocol which is determined by the most recent CONTROL statement (if any) addressed to register 3 since power-up.

A fourth (keyboard only) option is available. (SHIFT) (PAUSE) (or (RESET)) causes a hardware reset to be sent to ALL peripherals. This completely resets the datacomm interface to its power-up state with protocol and other options determined by the default switch settings.

Error Detection and Program Recovery

When a timeout or datacomm error occurs, an interrupt is generated by the interface card to BASIC. If an ON ERROR is active for that select code, the error is trapped and handled by the error routine specified by the ON ERROR statement. If no ON ERROR is active for that select code, the program is stopped at the end of the current line by the BASIC operating system, and an error message is sent to the PRINTER IS device.

When a datacomm error is trapped by an error routine, the routine must decide what to do about the problem. Options include the suggested recovery techniques discussed previously with the error messages, or orderly program termination. The options you select are determined by your specific application. Since datacomm interface errors are not related to a specific program line, the ERRL function is always false, and ERRN returns the error number generated by the interface card. ERRL and ERRN are discussed in greater detail in the BASIC Programming Techinques manual for your desktop computer.

Terminal Emulator Example Programs

The following pages contain complete listings of two terminal emulator programs based on the preceding discussion. The first program is for asynchronous data communication with an HP 1000. It can be easily adapted for other remote computers and different operating parameters. The second program uses Data Link to communicate with an HP 1000 network host. It can be used with the HP 3000, but the parity specifier must be changed, and other changes made as appropriate.

Both programs can be enhanced and expanded to include many additional features. The examples shown illustrate the general structure of terminal emulator programs, and are recommended as a basis for developing your own.

Other example programs are also included for your convenience and to further illustrate some of the concepts discussed in this chapter. If you have an HP 46020 keyboard, you need to adjust the ON KEY 0 LABEL statement in line 1750 (and any other affected lines).

```
1000  !     **********************************************************************
1010  !     *                                                                    *
1020  !     *               *****Example Async Terminal Emulator*****             *
1030  !     *                                                                    *
1040  !     **********************************************************************
1050  !     * This sample terminal emulator program is a simple example of the   *
1060  !     * program structure of general-purpose emulators.  It is not elegant,*
1070  !     * but contains the essential elements and illustrates commonly used  *
1080  !     * programming techniques.                                            *
1090  !     **********************************************************************
1100  !
1110           Sc=20                         ! Select code of datacomm interface.
1120           DIM A$[1],K$[100]             ! Set up string variables.
1130  !
1140  !     Reset datacomm interface and enable Async protocol.
1150  !
1160           CONTROL Sc,0;1                ! Reset card to disconnect from line.
1170           CONTROL Sc,3;1                ! Select Async protocol.
1180  Wait:STATUS Sc,38;All_sent            ! Wait until Control Block is sent to
1190           IF NOT All_sent THEN Wait     !   interface before resetting again.
1200           CONTROL Sc,0;1                ! Reset card to start new protocol.
1210  !
1220  !     Set up datacomm options.  Normally just a few are included in the
1230  !     program.  This group overrides ALL defaults including switches.
1240  !
1250           CONTROL Sc,14;3               ! Set Control Block mask for EOL and Prompt.
1260           CONTROL Sc,15;0               ! No modem line-change notification.
1270           CONTROL Sc,16;0               ! Disable connection timeout.
1280           CONTROL Sc,17;0               ! Disable No Activity timeout.
1290           CONTROL Sc,18;40              ! Lost Carrier 400 ms (default).
1300           CONTROL Sc,19;10              ! Transmit timeout 10 s (default).
1310           CONTROL Sc,20;7               ! Transmit Speed: 300 baud.
1320           CONTROL Sc,21;7               ! Receive Speed: 300 baud.
```

```
1330          CONTROL Sc,22;2      ! EQ/AK (as terminal) handshake.
1340          CONTROL Sc,23;1      ! Full Duplex Modem connection.
1350          CONTROL Sc,24;66     ! Remove protocol characters except
1360   !                               EOL.  Change errors to underscores.
1370          CONTROL Sc,26;6      ! Assign AK character for EQ/AK.
1380          CONTROL Sc,27;5      ! Assign EQ character for EQ/AK.
1390          CONTROL Sc,28;2,13,10  ! Set EOL sequence to CR/LF (default).
1400          CONTROL Sc,31;1,17   ! Set prompt to be DC1 (default).
1405   !                               Register 33 is not used.
1410          CONTROL Sc,34;2      ! Seven bits per character.
1420          CONTROL Sc,35;0      ! One stop bit per character.
1430          CONTROL Sc,36;1      ! Odd parity.
1440          CONTROL Sc,37;0      ! No inter-character time gap (default).
1450          CONTROL Sc,39;4      ! Set BREAK to four character times (default).
1460   !
1470   !  You are now ready to connect to the remote computer.  Optionally, this
1480   !  may include autodialing with the HP 13265A Modem.
1490   !
1500          CONTROL Sc,12;2                  ! Start Autodial.
1510          OUTPUT Sc;"> 9 @ (303) 555-1234"    ! Send telephone number string.
1520   !                    ^    ^        ^ ^
1530   !                    !    !      Unrecognized characters are ignored.
1540   !                    !    Insert 1-second pause (used with PBX to wait for
1550   !                 Select FAST dialing rate.                    dial tone).
1560   !
1570   !  Autodialing is not started until Start Connection is initiated by the
1580   !  following CONTROL statement:
1590   !
1600          CONTROL Sc,12;1                  ! Start the connection.
1610   !
1620   !  If desired, this is the proper place to monitor STATUS Register 12 to
1630   !  see if the connection is actually made.
1640   !
1650   Conn:STATUS Sc,12;Line_state      ! Get Line State from STATUS Register.
1660          IF Line_state=2 THEN DISP "Dialing"               ! State=2.
1670          IF Line_state=1 THEN DISP "Waiting to Connect"    ! State=1.
1680          IF Line_state<>3 THEN Conn                 ! Wait for connection.
1690          DISP "Connected"                   ! Connection is now complete.
1700   !
1710   !  Softkey 0 is set up so you can disconnect easily.
1720   !  Softkey 1 sends a break to the remote computer.
1730   !  Most other keys are trapped by the ON KBD interrupt service routine.
1740   !
1750          ON KEY 0 LABEL " Disconn" GOTO Disconnect      ! Set up Softkey 0.
1760          ON KEY 1 LABEL "  Break" GOSUB Break           ! Set up Softkey 1.
1770          ON KBD GOSUB Keyboard                      ! Set up keyboard interrupt.
1780   !
1790   !  Now set up the datacomm ON INTR service routine then enable interrupts
1800   !  for any data and/or Control Blocks (see STATUS Register 4 definition).
1810          ON INTR Sc GOSUB Datacomm
1820          ENABLE INTR Sc;1
1830   !
1840   !  Everything is handled under interrupt.  The background routine can be
1850   !  an idle loop doing nothing or a program that runs when interrupts are
1860   !  not being processed.
1870   !
1880   Background:GOTO Background
1890   !
1900   ! -->>>>>>>>>>>>>>> Datacomm Interrupt Service Routine <<<<<<<<<<<<<<<--
1910   !  This emulator operates in character mode, handling only one character
1920   !  at a time.  It is set up for no control blocks in the receive queue,
1930   !  and the dimension of A$ limits inputs from datacomm to one character.
1940   !
1950   !  The STATUS...4 acknowledges the interrupt from the card.  Since only
1960   !  one interrupt condition is enabled, there is no reason to check the
1970   !  value of STATUS Register 4.
1980   !
1990   !  The ENABLE INTR allows the card to generate another interrupt when it
2000   !  is ready.  BASIC does not branch to the service routine until after
```

```
2010  !    the RETURN exit is completed,
2020  !
2030  !    Since the datacomm interface can interrupt much faster than BASIC can
2040  !    service, exit from the routine occurs ONLY after ALL data has been
2050  !    removed from the receive queue,  Since an interrupt can be generated
2060  !    even though the data has already been ENTERed, we must check STATUS
2070  !    Register 5 FIRST to see if any data is available,
2080  !
2090       Datacomm:STATUS Sc,4;Interrupt_bits     ! Acknowledge interrupt by card,
2100            ENABLE INTR Sc                      ! Reenable interrupt,
2110       Dc:  STATUS Sc,5;Rx_avail_bits           ! Get data available status bits,
2120            IF Rx_avail_bits=0 THEN RETURN      ! If empty, exit service routine,
2130            ENTER Sc USING "#,-K";A$            ! Get next data byte,
2140            PRINT USING "#,K";A$                ! Print the character,
2150            GOTO Dc                             ! Check for more data available,
2160  !
2170  !    This keyboard routine is not very exotic, but it CAN handle a fast
2180  !    typist,  Some of the nested IF,,,THENs are used to decode the 255-
2190  !    and 255-255 notations for special and CONTROL-special keys,  The only
2200  !    special key allowed by this routine is ENTER (code is NUM("E")),  It
2210  !    is converted to a carriage-return followed by a line turn-around
2220  !    (;END) indication,  All ASCII keys are transmitted to the card without
2230  !    alteration,
2240  !
2250  !    The keyboard routine loops until the keyboard string has been
2260  !    completely serviced,  Notice the similarities between the keyboard and
2270  !    datacomm interrupt service routines,
2280  !
2290       Keyboard:K$=KBD$
2300       K:   IF NOT LEN(K$) THEN RETURN          ! Stay in routine until K$ is empty,
2310            Key=NUM(K$)                          ! Get key or prefix (255=non-ASCII),
2320            K$=K$[2]                             ! Strip first character from string,
2330            IF Key=255 THEN                      ! If not 255, transmit character,
2340               Key=NUM(K$)                       ! 255, Get value of next character,
2350               K$=K$[2]                          ! Strip second character,
2360               IF Key=255 THEN                   ! If 255 (CONTROL),
2370                  Key=NUM(K$)                    !    get third character value,
2380                  K$=K$[2]                       ! Strip third character and check
2390               END IF                            !    for ENTER,
2400               IF Key=NUM("E") THEN              ! Check non-ASCII to see if ENTER,
2410                  OUTPUT Sc;CHR$(13);END         ! Send CR then turn line around,
2420               ELSE                              ! Illegal character, Beep and return
2430                  BEEP                           !    for next character(s),
2440               END IF
2450            ELSE                                 ! ASCII key,  Send it to the remote
2460               OUTPUT Sc;CHR$(Key);             !    computer,
2470            END IF                               ! End of character check routine,
2480            GOTO K                               ! Go get next keystroke, if any,
2490  !
2500  !    Key 1 sends a BREAK indication to the datacomm interface card,
2510  !
2520       Break:CONTROL Sc,6;1                     ! Tell card to send a BREAK,
2530            RETURN                               ! End of routine,
2540  !
2550  !    Key 0 disconnects the card and stops the program,
2560  !
2570       Disco:CONTROL Sc,12;0                    ! Disconnect gracefully,
2580            DISP "Disconnected"
2590            END
```

If you have an HP 46020 keyboard, adjust the ON KEY statements to reflect available keys.

```
1000  !     ********************************************************************
1010  !     *                                                                  *
1020  !     *           *****Example Data Link Terminal Emulator*****           *
1030  !     *                                                                  *
1040  !     ********************************************************************
1050  !     * This sample terminal emulator program is a simple example of the *
1060  !     * program structure of general-purpose emulators.  It is not elegant,*
1070  !     * but contains the essential elements and illustrates commonly used *
1080  !     * programming techniques.  Line numbers are matched to the Async    *
1081  !     * example for your convenience in comparing the two versions.       *
1090  !     ********************************************************************
1100  !
1110         Sc=20                            ! Select code of datacomm interface.
1120         DIM A$[1050],K$[100]  ! ******->-> A$ now handles 1000 characters.
1130  !
1140  !   Reset datacomm interface and enable Async protocol.
1150  !
1160         CONTROL Sc,0;1                   ! Reset card to disconnect from line.
1170         CONTROL Sc,3;2                   ! Select Data Link protocol.
1180  Wait:STATUS Sc,38;All_sent             ! Wait until Control Block is sent to
1190       IF NOT All_sent THEN Wait         !  interface before resetting again.
1200       CONTROL Sc,0;1                     ! Reset card to start new protocol.
1210  !
1220  !   Set up datacomm options.  Normally just a few are included in the
1230  !   program.  This group overrides ALL defaults including switches.
1240  !
1250         CONTROL Sc,14;6      ! Set Control Block Mask for ETB/ETX.
1260         CONTROL Sc,15;0      ! Set ON INTR mask for data in receive queue.
1270         CONTROL Sc,16;0      ! Disable Connection timeout.
1280         CONTROL Sc,17;0      ! Disable Lost Carrier timeout.
1290         CONTROL Sc,18;40     ! Set Lost Carrier to 400 ms (default).
1300         CONTROL Sc,19;10     ! Set Transmit Timeout=10 s (default).
1310         CONTROL Sc,20;14     ! Set Line Speed to 9600 baud.
1320         CONTROL Sc,21;1      ! Set GID character to "A" (default).
1330         CONTROL Sc,22;1      ! Set DID character to "A".
1340         CONTROL Sc,23;0      ! Hardware Handshake OFF for HP 13264A.
1350         CONTROL Sc,24;0      ! Set transmit block size to 512 (default).
1360         CONTROL Sc,36;0      ! Parity not used with HP 1000 (default).
1460  !
1570  !   Now we can initiate Start Connection.
1590  !
1600         CONTROL Sc,12;1                     ! Start the connection.
1610  !
1620  !   If desired, this is the proper place to monitor STATUS Register 12 to
1630  !   see if the connection is actually made.
1640  !
1645         DISP "Trying to connect"
1650  Conn:STATUS Sc,12;Line_state      ! Get Line State from STATUS Register.
1680       IF Line_state<>3 THEN Conn                ! Wait for connection.
1690       DISP "Connected"                  ! Connection is now complete.
1700  !
1710  !   Softkey is set up so you can disconnect easily.
1720  !   Softkey sends a break to the remote computer.
1730  !   Most other keys are trapped by the ON KBD interrupt service routine.
1740  !
1750         ON KEY LABEL " Disconn" GOTO Disconnect      ! Set up Softkey.
1760         ON KEY LABEL "  Break" GOSUB Break           ! Set up Softkey.
1770         ON KBD GOSUB Keyboard                     ! Set up Keyboard interrupt.
1780  !
1790  !   Now set up the datacomm ON INTR service routine then enable interrupts
1800  !   for anything received (see STATUS Register 4 definition).  ***********
1810         ON INTR Sc GOSUB Datacomm
1820         ENABLE INTR Sc;1
1830  !
1840  !   Everything is handled under interrupt.  The background routine can be
1850  !   an idle loop doing nothing or a program that runs when interrupts are
1860  !   not being processed.
1870  !
1880     Background:GOTO Background
```

```
1890  !
1900  !  -->>>>>>>>>>>>>> Datacomm Interrupt Service Routine <<<<<<<<<<<<<<--
1910  !     This emulator operates in block mode, handling incoming data one block
1920  !     at a time.  Entire data blocks are read from the receive queue, but
1930  !     they MUST be properly terminated by a Control Block.
1940  !
1950  !     The STATUS...4 acknowledges the interrupt from the card.  Since only
1960  !     one interrupt condition is enabled, there is no reason to check the
1970  !     value of STATUS Register 4.
1980  !
1990  !     The ENABLE INTR allows the card to generate another interrupt when it
2000  !     is ready.  BASIC does not branch to the service routine until after
2010  !     the RETURN exit is completed (i.e., the routine does not call itself).
2020  !
2030  !     Since the datacomm interface can interrupt much faster than BASIC can
2040  !     service, exit from the routine occurs ONLY after ALL data has been
2050  !     removed from the receive queue.  Since an interrupt can be generated
2060  !     even though the data has already been ENTERed, we must check STATUS
2070  !     Register 5 FIRST to see if any data is available.
2080  !
2090    Datacomm:STATUS Sc,4;Interrupt_bits      ! Acknowledge interrupt by card.
2100         ENABLE INTR Sc                      ! Reenable interrupt.
2110    Dc:  STATUS Sc,5;Rx_avail_bits           ! Get data available status bits.
2120         IF Rx_avail_bits<2 THEN RETURN !****If no control block, exit.
2130         ENTER Sc USING "#,-K";A$            ! Get next data byte.
2140         PRINT USING "#,K";A$                ! Print the incoming block.
2150         GOTO Dc                             ! Check for more data available.
2160  !
2170  !     This keyboard routine is not very exotic, but it CAN handle a fast
2180  !     typist.  Some of the nested IF...THENs are used to decode the 255-
2190  !     and 255-255 notations for special and CONTROL-special keys.  The only
2200  !     special key allowed by this routine is ENTER (code is NUM("E").  It
2210  !     is converted to an end-of-block (;END) indication.  All ASCII keys are
2220  !     are transmitted to the card without alteration.
2240  !
2250  !     The keyboard routine loops until the keyboard string has been
2260  !     completely serviced.  Notice the similarities between the keyboard and
2270  !     datacomm interrupt service routines.
2280  !
2290    Keyboard:K$=KBD$
2300    K:   IF NOT LEN(K$) THEN RETURN         ! Stay in routine until K$ is empty.
2310         Key=NUM(K$)                        ! Get key or prefix (255=non-ASCII).
2320         K$=K$[2]                           ! Strip first character from string.
2330         IF Key=255 THEN                    ! If not 255, transmit character.
2340              Key=NUM(K$)                   ! 255. Get value of next character.
2350              K$=K$[2]                      ! Strip second character.
2360              IF Key=255 THEN               ! If 255 (CONTROL),
2370                   Key=NUM(K$)              !      get third character value.
2380                   K$=K$[2]                 ! Strip third character and check
2390              END IF                        !      for ENTER.
2400              IF Key=NUM("E") THEN          ! Check non-ASCII to see if ENTER.
2410                   OUTPUT Sc;END            ! Send end-of-block.
2420              ELSE                          ! Illegal character. Beep and return
2430                   BEEP                     !      for next character(s).
2440              END IF
2450         ELSE                               ! ASCII key.  Send it to the remote
2460              OUTPUT Sc;CHR$(Key);          !      computer.
2465              PRINT USING "#,A";CHR$(Key)   ! Print character not echoed by DL.
2470         END IF                             ! End of character check routine.
2480         GOTO K                             ! Go get next keystroke, if any.
2490  !
2500  !     Key 1 sends a BREAK indication to the datacomm interface card.
2510  !
2520    Break:CONTROL Sc,6;1                    ! Tell card to send a BREAK.
2530         RETURN                             ! End of routine.
2540  !
2550  !     Key 0 disconnects the card and stops the program.
2560  !
2570    Disco:CONTROL Sc,12;0                   ! Disconnect gracefully.
2580         DISP "Disconnected"
2590         END
```

Datacomm Programming Helps

This section is designed to assist you in writing datacomm programs for special applications by discussing selected techniques and characteristics that can present obstacles to the beginning programmer.

Terminal Prompt Messages

Care must be exercised to ensure that messages are never transmitted to the network host if the host is not prepared to properly handle the message. Receipt of a poll from the host does not necessarily mean that the host can handle the message properly when it is received. Therefore, prompts or interpretation of messages from the host are used to determine the status of the host operating system.

Prompts are message strings sent to the terminal by a cooperating program. They are well-defined and predictable, and are usually tailored to specific applications. When the terminal interacts directly with RTE or one or more subsystems, the process becomes less straightforward. Each subsystem usually has its own prompt which is not identical to other subsystem prompts. To maintain orderly communication with subsystems, you must interpret each message string from the host to determine whether it is to be treated as a prompt.

Prevention of Data Loss on the HP 1000

On the HP 1000, the RTE Operating System manages information transfer between programs or subsystems and system I/O devices, including DSN/DL. Terminals are continually polled by the host's data link interface (unless auto-poll has been disabled by use of an HP 1000 File Manager CN command). Since there is no relationship between automatic polling and HP 1000 program and subsystems execution, it is possible to poll a terminal when there is no need for information from that terminal. If the terminal sends a message in response to a poll when no data is being requested, the HP 1000 discards the message, causing the data to be lost, and treats it as an asynchronous interrupt. A break-mode prompt is then sent to the terminal by the host.

The terminal must determine that the host is ready to receive a message in order to ensure that messages are properly handled by the host. This is done by checking all messages from the host (ENTER until queue is empty) and not transmitting (OUTPUT) until a prompt message or its equivalent has been received (unless you want to enter break-mode operation). Since the HP 1000 does not generate a consistent prompt message for all programs and subsystems, it is easiest to use cooperating programs to generate a predictable prompt. If your application requires interaction with other subsystems, prompts can usually be most easily identified by the ABSENCE of the sequence: $C_R L_F E_C$_ at the end of a message. When a proper sequence has been identified, you are reasonably certain that the host is ready for your next message block.

Here is an example of host messages where a prompt is sent by the File Manager (FMGR) and answered by a RUN,EDITR command. Note that the prompt from the interactive editor fits the description of a prompt because a line-feed is not included after the carriage-return in the sequence.

:^{E}c_	Prompt is sent by FMGR to terminal.
RU ,EDITR	EDITR Run command is sent to host.
SOURCE FILE NAME?^{C}R^{L}F^{E}c_	File name message is sent by the host, followed by a
^{C}R/^{B}L^{E}c_	prompt sequence which has no line-feed. Sequence is different from FMGR prompt.

Whenever an unexpected message from a terminal is received by RTE, it is treated as an asynchronous interrupt which terminates normal communication with that terminal. A break-mode prompt is sent to the terminal by RTE, and the next message is expected to be a valid break-mode command. If the the message is not a valid command (such as data in a file being transferred), the data is discarded, and an error message is sent to the terminal. If, in the meantime, the cooperating program or subsystem generates an input request, the next data block is sent to the proper destination, but is out of sequence because at least one block has been lost. You can prevent such data losses and the mass confusion that usually ensues (especially during high-speed file transfers to the host), by disabling auto-poll on the HP 1000 data link interface. With auto-poll OFF, no polls are sent to your terminal unless the host is prepared to receive data.

Disabling Auto-poll on the HP 1000

To operate with auto-poll OFF, log on to the network host, disable auto-poll, perform all datacomm activities and file transfers, enable auto-poll, then log off. **If you don't enable auto-poll at the end of a session, polling is suspended to your terminal after log-off, and you cannot reestablish communication with the host unless polling is restored from another terminal or the network host System Console.**

The auto-poll ON/OFF commands are:

       CN ,LU# ,23B ,101401B   Auto-poll OFF[1]
       CN ,LU# ,23B ,001401B   Auto-poll ON[1]

where LU# is the logical unit number assigned to your terminal.

When auto-poll is disabled, no polls are sent to your terminal unless an input request is initiated by the cooperating program or subsystem on the network host. When the request is made, a poll is scheduled, and polling continues until a reply is received from the terminal. When the reply is received, and acknowledged, polling is suspended until the next input is scheduled. Operating with auto-poll OFF is especially useful when transferring files TO the HP 1000. Otherwise, in most applications, it is practical to leave auto-poll ON.

---

[1] The File Manager CN (Control) command parameters for the multipoint interface are described in more detail in the 91730A Multipoint Terminal Interface Subsystem User's Guide

Prevention of Data Loss on the HP 3000

Neither the HP 1000 nor the HP 3000 provide a DC1 poll character when they are ready for data inputs from DSN/DL. The HP 3000, like the HP 1000, also discards data if it has not requested the transfer. Since the HP 3000 does not provide an auto-poll disable command, you must interpret messages from the HP 3000 to determine that it is ready for the next data block before you transmit the block.

Secondary Channel, Half-duplex Communication

Half-duplex telecommunications links frequently use secondary channel communication to control data transmission and provide for proper line turn-around. This is done by using Secondary Request-to-send (SRTS) and Secondary Data Carrier Detect (SDCD) modem signals.

Consider two devices communicating with each other: Each connects to the datacomm link, then waits for SDCD to become active (true). As each device connects to the line, Secondary Request-to-send is enabled, causing each modem to activate its secondary carrier output. The Secondary Data Carrier Detect is, in turn, activated by each modem as it receives the secondary data carrier from the other end.

When communication begins, the first device to transmit (assumed to be your computer, in this case) clears its Secondary Request-to-send modem line. This removes the secondary data carrier from the line, causing the other modem to clear SDCD to its terminal or computer, telling it that you have the line. (The modems also maintain proper line switching and prevent timing conflicts so both ends don't try to get the line simultaneously.) The other device receives data, and must not attempt to transmit until you relinquish control of the line as indicated by SDCD true. After you finish transmitting, you must again activate SRTS so that SDCD can be activated to the other device, allowing it to use the line if it has a message.

The following example is a simple terminal emulator that uses secondary channel communication to control data flow on a half-duplex link:

```
1000 ! ***************************************************************************
1010 ! *                                                                         *
1020 ! *     HALF-DUPLEX TERMINAL EMULATOR FOR SECONDARY CHANNEL OPERATION       *
1030 ! *                                                                         *
1040 ! *  This program uses secondary channel modem lines to indicate which      *
1050 ! *  end is in control of the line.  BASIC is used to assemble data         *
1060 ! *  for transmission to the other end.  This example is compatible         *
1070 ! *  with the Option 001 (male) cable only.                                 *
1080 ! *                                                                         *
1090 ! ***************************************************************************
1100 !
1110          Sc=20                 ! Select code of HP 98628 datacomm interface.
1120          DIM A$[1],K$[100]     ! Size of datacomm and keyboard strings.
1130 !
1140 ! Reset the card to disconnect, then select Async protocol.
1150 !
1160          CONTROL Sc,0;1
1170          CONTROL Sc,3;1
1180 Wait:    STATUS Sc,38;All_sent
1190          IF NOT All_sent THEN Wait
1200          CONTROL Sc,0;1
1210 !
1220 ! Set up all the interface configuration options for Async protocol.
```

```
1230 !
1240          CONTROL Sc,14;0      ! Set Control Block mask off.
1250          CONTROL Sc,15;16     ! Interrupt when Secondary Carrier Detect
1255                               !   modem line changes state.
1260          CONTROL Sc,16;0      ! Disable connection timeout.
1270          CONTROL Sc,17;0      ! Disable No Activity timeout.
1280          CONTROL Sc,18;40     ! Lost Carrier 400 ms (default).
1290          CONTROL Sc,19;10     ! Transmit timeout 10 s (default).
1300          CONTROL Sc,20;7,7    ! Line speed: 300 baud in both directions.
1310          CONTROL Sc,22;0      ! Disable protocol handshake.
1320          CONTROL Sc,23;2      ! Half duplex modem connection.
1330          CONTROL Sc,24;255    ! Do not remove protocol characters.
1340          CONTROL Sc,28;2,13,10! EOL sequence CR/LF (default).
1350          CONTROL Sc,31;1,17   ! Prompt DC1 (default).
1360          CONTROL Sc,34;2      ! 7 bits per character.
1370          CONTROL Sc,35;0      ! 1 stop bit.
1380          CONTROL Sc,36;1      ! odd parity.
1390          CONTROL Sc,37;0      ! No inter-character gap (default).
1400          CONTROL Sc,39;4      ! Set Break to 4 character times (default).
1410 !
1420 ! Initiate connection to the telecommunications line.
1430 !
1440          CONTROL Sc,12;1
1450 !
1460 ! Tell the operator what is happening, then wait for connection to finish.
1470 !
1480          DISP "Waiting to connect"
1490 Conn:    STATUS Sc,12;Line_state
1500          IF Line_state=1 THEN Conn
1510          DISP "Waiting for SDCD to become active"
1520 !
1530 ! Get the SDCD handshake started properly by waiting for the other end to
1540 ! relinquish control of the line by activating SDCD.
1550 !
1560 Statck:STATUS Sc,7;Modem_lines
1570          IF NOT BINAND(Modem_lines,16) THEN Statck
1580          DISP "Connected"
1590 !
1600 ! Set up a key to gracefully disconnect the datacomm connection.
1610 !
1620          ON KEY 0 LABEL " Disconn" GOTO Disconnect
1630 !
1640 ! Interrupt on data received or modem line change (change in SDCD).
1650 !
1660          ON INTR Sc GOSUB Datacomm
1670          ENABLE INTR Sc;1+8
1680 !
1690 ! Send a "READY" message to the remote to get things started.  This is
1700 ! optional.
1710 !
1720          CONTROL Sc,8;7    ! Put down SRTS
1730          OUTPUT Sc;"READY";CHR$(13);END
1740          CONTROL Sc,8;15   ! Put up SRTS
1750 !
1760 ! The background idle loop simply waits for interrupts to happen.
1770 !
1780 Background: GOTO Background
1790 !
1800 ! *****************************************************************
1810 !                  DATACOMM INTERRUPT SERVICE ROUTINE
1820 !
1830 ! First, acknowledge interrupt by reading STATUS register 4.
1840 !
1850 ! Read all existing data in the buffer.
1860 !
1870 ! When SDCD becomes true, it indicates that the remote is through
1880 ! transmitting.  A LINPUT statement is provided to let the user enter a
1890 ! line of data.  The line is then sent to both the screen and the
1900 ! datacomm card.  To maintain control of the line, we disable SRTS (Control
```

```
1910 ! Register 8), then reactivate it when we are through sending,
1920 !
1930 ! Finally, re-enable interrupts and exit the interrupt routine,
1940 !
1950 Datacomm:STATUS Sc,4;Interrupt_bits
1960 Read:   STATUS Sc,5;Rx_avail_bits
1970         IF Rx_avail_bits=0 THEN Chkmdm
1980             ENTER Sc USING "#,-K";A$
1990             PRINT USING "#,K";A$
2000         GOTO Read
2010 Chkmdm:STATUS Sc,7;Modem_lines
2020         IF BINAND(Modem_lines,16) THEN
2030             CONTROL Sc,8;7   ! Put down SRTS
2040             LINPUT "Line to send...?",K$
2050             PRINT K$
2060             OUTPUT Sc;K$;CHR$(13);END
2070             CONTROL Sc,8;15  ! Put up SRTS
2080         END IF
2090         ENABLE INTR Sc
2100         RETURN
2110 ! ********************************************************************
2120 ! Key 0 was set up to disconnect from the datacomm line,
2130 !
2140 Disconnect:CONTROL Sc,12;0
2150         DISP "Disconnected"
2160         END
```

Automatic Answering Applications

Desktop computers are sometimes used in applications where they may have to be able to automatically answer incoming calls from other computers by means of public (switched) telephone lines. For instance, a desktop computer may be located at an unattended remote site in a data gathering network where the network host computer periodically calls the remote site for data updates. In other situations, the desktop computer may be the host for several computers or terminals that originate the calls. Other applications may require that two (or more) desktop computers be able to call each other in either direction at will.

In automatic answering applications, the Ring Indicator (RI) modem line is used by the desktop computer to recognize incoming calls from the host. This enables the desktop computer to answer the call by connecting to the datacomm line. Usually, a continuously running program on the unattended computer contains an ON INTR statement set up to monitor the RI modem signal. When RI is activated by the incoming call, normal program flow is interrupted, and the connection is initiated. The desktop computer then sets up the necessary datacomm and other program interrupts, and proceeds to the program segment responsible for transferring data to the remote computer. The following example illustrates the general technique and how it fits into overall program structure:

```
1000 ! ********************************************************************
1010 ! *                                                                  *
1020 ! *      TERMINAL EMULATOR WITH AUTOMATIC ANSWERING CAPABILITY       *
1030 ! *                                                                  *
1040 ! *   This program waits for the ring-indicator modem line to change *
1050 ! *   (indicating an incoming datacomm call), then connects to the   *
1060 ! *   datacomm line,  Use with Option 001 (male) modem cable,        *
1070 ! *                                                                  *
1080 ! ********************************************************************
1090 !
1100         Sc=20            ! Select code of HP 98628 datacomm interface,
1110         DIM A$[1],K$[100]! Size of datacomm and keyboard strings,
1120 !
```

```
1130 ! Reset the card to disconnect, then select Async protocol.
1140 !
1150        CONTROL Sc,0;1
1160        CONTROL Sc,3;1
1170 Wait:  STATUS Sc,38;All_sent
1180        IF NOT All_sent THEN Wait
1190        CONTROL Sc,0;1
1200 !
1210 ! Set up all the interface configuration options for Async protocol.
1220 !
1230        CONTROL Sc,14;0        ! Set Control Block mask off.
1240        CONTROL Sc,15;8        ! Interrupt when Ring Indicator line changes.
1250        CONTROL Sc,16;0        ! Disable connection timeout.
1260        CONTROL Sc,17;0        ! Disable No Activity timeout.
1270        CONTROL Sc,18;40       ! Lost Carrier 400 ms (default).
1280        CONTROL Sc,19;10       ! Transmit timeout 10 s (default).
1290        CONTROL Sc,20;7,7      ! Line speed: 300 baud in both directions.
1300        CONTROL Sc,22;0        ! Disable protocol handshake.
1310        CONTROL Sc,23;1        ! Full duplex modem connection.
1320        CONTROL Sc,24;255      ! Remove no protocol characters.
1330        CONTROL Sc,28;2,13,10! EOL sequence CR/LF (default).
1340        CONTROL Sc,31;1,17     ! Prompt DC1 (default).
1350        CONTROL Sc,34;2        ! 7 bits per character.
1360        CONTROL Sc,35;0        ! 1 stop bit.
1370        CONTROL Sc,36;1        ! Odd parity.
1380        CONTROL Sc,37;0        ! No inter-character gap (default).
1390        CONTROL Sc,39;4        ! Set Break to 4 character times (default).
1400 !
1410 ! Wait for Ring Indicator modem line to change.
1420 !
1430        ON INTR Sc GOTO Ri_int
1440        ENABLE INTR Sc;8
1450        DISP "Waiting for ring to come in"
1460 Waitri:GOTO Waitri
1470 !
1480 ! When interrupt occurs, initiate connection to the datacomm line.
1490 !
1500 Ri_int:CONTROL Sc,12;1
1510 !
1520 ! Tell the operator what is happening, then wait for connection to finish.
1530 !
1540        DISP "Waiting to connect"
1550 Conn:  STATUS Sc,12;Line_state
1560        IF Line_state=1 THEN Conn
1570        DISP "Connected"
1580 !
1590 ! Set up key 0 to gracefully disconnect from the datacomm line, then
1600 !   set up key 1 to send a break.
1610 !
1620        ON KEY 0 LABEL " Disconn" GOTO Disconnect
1630        ON KEY 1 LABEL "  Break" GOSUB Break
1640 !
1650 ! Interrupt on data received.  Also set up keyboard interrupts.
1660 !
1670        ON INTR Sc GOSUB Datacomm
1680        ENABLE INTR Sc;1
1690        ON KBD GOSUB Keyboard
1700 !
1710 ! The background idle loop simply waits for interrupts to happen.
1720 !
1730 Background: GOTO Background
1740 ! ***************************************************************************
1750 !                      DATACOMM INTERRUPT SERVICE ROUTINE
1760 !
1770 ! First, acknowledge interrupt by reading STATUS register 4.
1780 !
1790 ! Re-enable interrupts, then read all existing data in the buffer.
1800 !
1810 ! When the buffer is empty, exit the service routine.
```

```
1820 !
1830 Datacomm:STATUS Sc,4;Interrupt_bits
1840         ENABLE INTR Sc
1850 Read:   STATUS Sc,5;Rx_avail_bits
1860         IF Rx_avail_bits=0 THEN RETURN
1870             ENTER Sc USING "#,-K";A$
1880             PRINT USING "#,K";A$
1890         GOTO Read
1900 ! ***********************************************************************
1910 ! This keyboard interrupt service routine is similar to the other
1920 ! examples in this chapter.  It sends ASCII keys to the remote, and
1930 ! accepts ENTER as a Carriage-Return.  Other keys cause a BEEP.
1940 !
1950 Keyboard:K$=KBD$
1960 K:      IF NOT LEN(K$) THEN RETURN       ! Repeat until K$ is empty:
1970         Key=NUM(K$)                      ! Get key or prefix
1980         K$=K$[2]
1990         IF Key=255 THEN
2000             Key=NUM(K$)                  ! If prefix, get next character
2010             K$=K$[2]
2020             IF Key=255 THEN              ! If control-key prefix, get
2030                 Key=NUM(K$)              ! the third character
2040                 K$=K$[2]
2050             END IF
2060             IF Key=NUM("E") THEN         ! Check for ENTER key
2070                 OUTPUT Sc;CHR$(13);END   ! If so, send carriage return
2080             ELSE
2090                 BEEP
2100             END IF
2110         ELSE
2120             OUTPUT Sc;CHR$(Key);         ! ASCII key: just send it
2130         END IF
2140         GOTO K                           ! Repeat until K$ is empty
2150 ! ***********************************************************************
2160 ! Key 1 was set up to send a break.
2170 !
2180 Break:  CONTROL Sc,6;1
2190         RETURN
2200 !
2210 ! Key 0 was set up to disconnect the interface from the datacomm line.
2220 !
2230 Disconnect:CONTROL Sc,12;0
2240         DISP "Disconnected"
2250         END
```

Communication Between Desktop Computers

Two desktop computers can be connected, directly, or by use of modems. DC1/DC3 handshake protocol can be used conveniently to enable each computer to transmit at will without risk of buffer or queue overruns. To ensure proper operation, the following guidelines apply:

- Set up Control Register 22 with a value of 5. This allows both computers to act either as host or terminal in any given situation, depending on which one initiates the action.
- Set up Control Registers 26 and 27 for DC1 and DC3 respectively, or use two other characters if necessary.
- Data to be transmitted must NOT contain any characters matching the contents of Control Register 26 or 27. This prevents the receiving interface from confusing data with control characters.
- If both computers attempt to transmit large amounts of data at the same time, a lock-up condition may result where each side is waiting for the other to empty its buffers.

Cable and Adapter Options and Functions

The HP 98628A Datacomm Interface is available with RS-232C DTE and DCE cable configurations, or it can be connected to various modems or adapters for other applications.

DTE and DCE Cable Options

DTE and DCE cable options are designed to simplify connecting two desktop computers without the use of modems. The DTE cable (male RS-232 connector) is configured to make the datacomm interface look like standard data terminal equipment when it is connected to an RS-232C modem. The DCE cable (female RS-232 connector) is configured so that it eliminates the need for modems in a direct connection. When you connect two computers to each other in a direct non-modem connection, both datacomm interfaces are functionally identical. The DCE cable acts as an adapter so that both interfaces behave exactly as they would if they were connected to a pair of modems by means of DTE cables.

Several signal lines are rerouted in the DCE cable so that, in direct connections, outputs from one interface are connected to the corresponding inputs on the other interface. Certain outputs on each interface are also connected to inputs on the same card by "loop-back" connections in the DCE cable.

The schematic diagram in this section shows two datacomm interfaces directly connected through a DTE-DCE cable pair. Note that the DCE cable wiring complements the DTE cable so that output signals are properly routed to their respective destinations. Signal names at the RS-232C connector interface are the same as the signal names for the DTE interface. However, because the DCE cable adapts signal paths, the signal name at the RS-232C connector does not necessarily match the signal name at the DCE interface. Connector pin numbers are included in the diagram for your convenience.

RS-232C DTE (male) Cable Signal Identification Tables

Signal RS-232C	V.24	Interface Pin #	RS-232C Pin #	Mnemonic	I/O	Function
AA	101	24	1	–	–	Safety Ground
BA	103	12	2		Out	Transmitted Data
BB	104	42	3		In	Received Data
CA	105	13	4	RTS	Out	Request to Send
CB	108	44	5	CTS	In	Clear to Send
CC	107	45	6	DSR	In	Data Set Ready
AB	102	48	7	–	–	Signal Ground
CF	109	46	8	DCD	In	Data Carrier Detect
SCF (OCR2)	122	47	12	SDCD	In	Secondary DCD
DB	114	41	15		In	DCE Transmit Timing
DD	115	43	17		In	DCE Receive Timing
SCA (OCD2)	120	15	19	SRTS	Out	Secondary RTS
CD	108.1	14	20	DTR	Out	Data Terminal Ready
CE (OCR1)	125	9	22	RI	In	Ring Indicator
CH (OCD1)	111	40	23	DRS	Out	Data Rate Select
DA	113	7	24		Out	Terminal Transmit Timing

Optional Circuit Driver/Receiver Functions

Two optional drivers and receivers are used with the RS-232C cable options. Their functions are as follows:

	Drivers			**Receivers**	
Name	**Function**		**Name**	**Function**	
OCD1	Data Rate Select		OCR1	Ring Indicator	
OCD2	Secondary Request-to-send		OCR2	Secondary Data Carrier Detect	
OCD3	Not used				
OCD4	Not used				

OCD2 is used for autodial pulsing in the HP 13265A Modem. None of the optional drivers and receivers are used for Data Link and Current Loop Adapters.
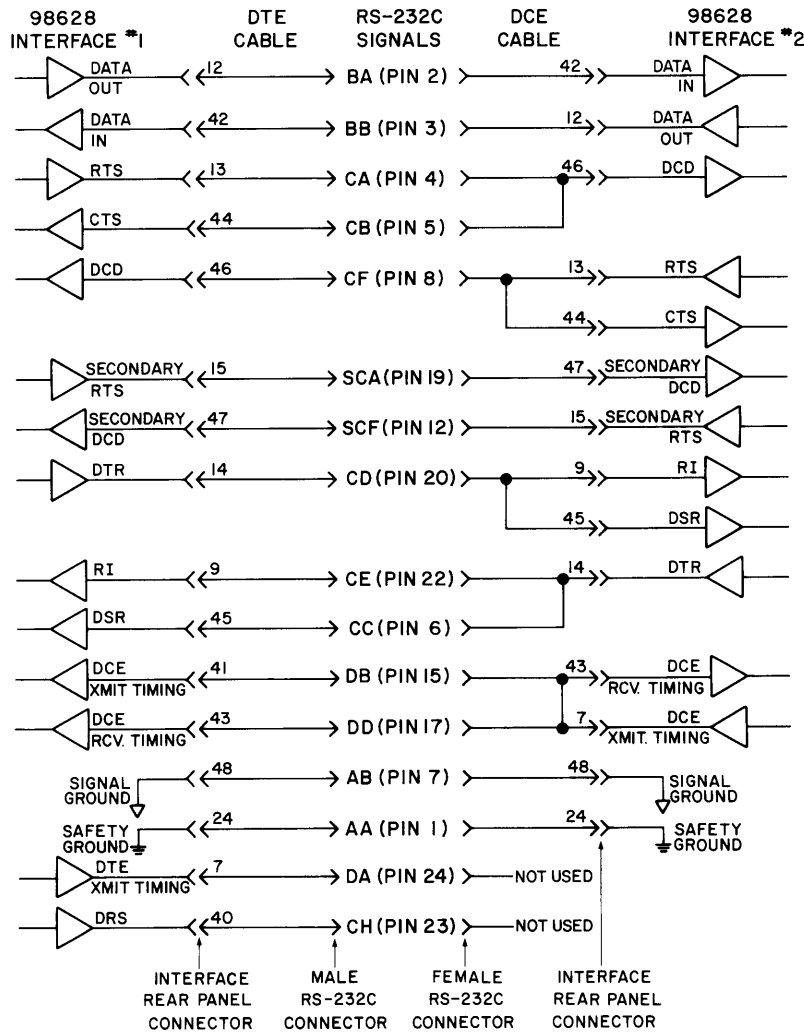
DTE/DCE Interface Cable Wiring

RS-232C/CCITT V24[1]

The following table provides information about each data communications interface function. The pin assignments are also shown. Not all of the functions provided by RS-232C are implemented. The functions denoted with an * **are implemented.**

RS-232C/CCITT V24[1]

RS-232C	CCITT V24	Signal Name
*Pin 1	101	PROTECTIVE GROUND. Electrical equipment frame and ac power ground.
*Pin 2	103	TRANSMITTED DATA. Data originated by the terminal to be transmitted via the sending modem.
*Pin 3	104	RECEIVED DATA. Data from the receiving modem in response to analog signals transmitted from the sending modem.
*Pin 4	105	REQUEST TO SEND. Indicates to the sending modem that the terminal is ready to transmit data.
*Pin 5	106	CLEAR TO SEND. Indicates to the terminal that its modem is ready to transmit data.
*Pin 6	107	DATA SET READY. Indicates to the terminal that its modem is not in a test mode and that modem power is ON.
*Pin 7	102	SIGNAL GROUND. Establishes common reference between the modem and the terminal.
*Pin 8	109	DATA CARRIER DETECT. Indicates to the terminal that its modem is receiving carrier signals from the sending modem.
Pin 9		Reserved for test.
Pin 10		Reserved for test.
Pin 11		Unassigned.
*Pin 12	122	SECONDARY DATA CARRIER DETECT. Indicates to the terminal that its modem is receiving secondary carrier signals from the sending modem.
Pin 13	121	SECONDARY CLEAR TO SEND. Indicates to the terminal that its modem is ready to transmit signals via the secondary channel.

These signals are commonly used for 3 wire (no modem) links.

[1] International Telephone and Telegraph Consultative Committee European standard.

RS-232C/CCITT V24 (Cont'd)

RS-232C	CCITT V24	Signal Name
Pin 14	118	SECONDARY TRANSMITTED DATA. Data from the terminal to be transmitted by the sending modem's channel.
*Pin 15	114	TRANSMITTER SIGNAL ELEMENT TIMING. Signal from the modem to the transmitting terminal to provide signal element timing information.
Pin 16	119	SECONDARY RECEIVED DATA. Data from the modem's secondary channel in response to analog signals transmitted from the sending modem.
*Pin 17	115	RECEIVER SIGNAL ELEMENT TIMING. Signal to the receiving terminal to provide signal element timing information.
Pin 18		Unassigned.
*Pin 19	120	SECONDARY REQUEST TO SEND. Indicates to the modem that the sending terminal is ready to transmit data via the secondary channel.
*Pin 20	108.2	DATA TERMINAL READY. Indicates to the modem that the associated terminal is ready to receive and transmit data.
Pin 21	110	SIGNAL QUALITY DETECTOR. Signal from the modem telling whether a defined error rate in the received data has been exceeded.
*Pin 22	125	RING INDICATOR. Signal from the modem indicating that a ringing signal is being received over the line.
*Pin 23	111	DATA SIGNAL RATE SELECTOR. Selects one of two signaling rates in modems having two rates.
*Pin 24	113	TRANSMIT SIGNAL ELEMENT TIMING. Transmit clock provided by the terminal.
Pin 25		Unassigned.

Summary of Datacomm
Status and Control Registers

Unless indicated otherwise, the Status Register returns the current value for a given parameter; the Control Register sets a new value.

Register	Function
0	Control: Interface Reset; Status: Interface Card ID
1 (Status only)	Hardware Interrupt Status: 1 = Enabled, 0 = Disabled
2 (Status only)	Datacomm activity: 0 = inactive, 1 = ENTER in process, 2 = OUTPUT in process
3	Select Protocol: 1 = Async, 2 = Data Link
4 (Status only)	Cause of ON INTR program branch
5	Control: Terminate transmission; Status: Inbound queue status
6	Control: Send BREAK to remote; Status: 1 = BREAK pending
7 (Status only)	Current modem receiver line states
8	Modem driver line states
9 (Status only)	Control block TYPE
10 (Status only)	Control block MODE
11 (Status only)	Available outbound queue space
12	Control: Connect/Disconnect line; Status: Line connection status
13	ON INTR mask
14	Control Block mask
15	Modem Line interrupt mask
16	Connection timeout limit
17	No Activity timeout limit
18	Lost Carrier timeout limit
19	Transmit timeout limit
20	Async: Transmit baud rate (line speed) Data Link: Set Transmit/Receive baud rate (line speed)
21	Async: Incoming (receiver) baud rate (line speed) Data Link: GID address (0 thru 26 corresponds to "@" thru "Z")
22	Async: Protocol handshake type Data Link: DID address (0 thru 26 corresponds to "@" thru "Z")
23	Hardware handshake type: ON/OFF, HALF/FULL duplex, Modem/Non-modem

Register	Function
24	Async: Control Character mask
	Data Link: Block Size limit
25 (Status only)	Number of received errors since last interface reset
26	Async: First protocol character (ACK/DC1)
	Data Link: NAKs received since last interface reset

Registers 27-35, 37, and 39 are used with Async protocol only. They are not accessible during Data Link operation.

27	Second protocol handshake character (ENQ/DC3)
28	Number of characters in End-of-line sequence
29	First character in EOL sequence
30	Second character in EOL sequence
31	Number of characters in PROMPT sequence
32	First character in PROMPT sequence
33	Second character in PROMPT sequence
34	Data bits per character excluding start, stop and parity
35	Stop bits per character (0 = 1, 1 = 1.5, and 2 = 2 stop bits)
36	Parity sense: 0 = NONE, 1 = ODD, 2 = EVEN, 3 = ZERO, 4 = ONE
	Data Link: 0 = NONE (HP 1000 host), 1 = ODD (HP 3000 host)
37	Inter-character time gap in character times (Async only)
38 (Status only)	Transmit queue status (1 = empty)
39	BREAK time in character times (Async only)

HP 98628 Data Communications Interface
Status and Control Registers

General Notes: Control registers accept values in the range of zero through 255. Some registers require specified values, as indicated. Illegal values or values less than zero or greater than 255, cause ERROR 327.

Reset value, shown for various Control Registers, is the default value used by the interface after a reset or power-up until the value is overridden by a CONTROL statement.

Status 0   Card Identification
Value returned: 52 (if 180 is returned, check select code switch cluster and make sure switch R is ON).

Control 0   Card Reset
Any value, 1 thru 255, resets the card. Immediate execution. Data in queues is destroyed.

Status 1   Hardware Interrupt Status (not used in most applications)
1 = Enabled   0 = Disabled

Status 2   Datacomm Activity
0 = No activity pending on this select code.
Bit 0 set: ENTER in process.
Bit 1 set: OUTPUT in process.
(Non-zero ONLY during multi-line function calls.)

Status 3   Current Protocol Identification:
1 = Async, 2 = Data Link Protocol

Control 3   Protocol to be used after next card reset (CONTROL Sc,0;1)
1 = Async Protocol   2 = Data Link Protocol
This register overrides default switch configuration.

Status 4   Cause of ON INTR program branch.

Bit	Function: Async Protocol	Function: Data Link Protocol
0	Data and/or Control Block available	Data Block Available
1	Prompt received	Space available for a new transmission block
2	Framing and or parity error	Receive or transmit error
3	Modem line change	Modem line change
4	No Activity timeout (forces a disconnect)	No Activity timeout (forces a disconnect)
5	Lost carrier or connection timeout (forces a disconnect)	Lost carrier or connection timeout (forces a disconnect)
6	End-of-line received	Not Used
7	Break received	Not Used

Contents of this register are cleared when a STATUS statement is executed to it.

Status 5   Inbound queue status

Value	Interpretation
0	Queue is empty
1	Queue contains data but no control blocks
2	Queue contains one or more control blocks but no data
3	Queue contains both data and one or more control blocks

Control 5   Terminate Transmission

OUTPUT S,5;0   is equivalent to OUTPUT S;END

Data Link: Sends previous data as a single block with an ETX terminator, then idles the line with an EOT.

Async: Tells card to turn half-duplex line around. Does nothing when line is full-duplex. The next data OUTPUT automatically regains control of the line by raising the RTS (request-to-send) modem line.

Status 6   Break status: 1 = BREAK transmission pending, 0 = no BREAK pending.

Control 6   Send Break; causes a Break to be sent as follows:

Data Link Protocol: Send Reverse Interrupt (RVI) reply to inbound block, or CN character instead of data in next outbound block.

Async Protocol: Transmit Break. Length is defined by Control Register 39.

Note that the value sent to the register is arbitrary.

Status 7   Modem receiver line states (values shown are for male cable connector option for connection to modems).

Bit 0: Data Mode (Data Set Ready) line
Bit 1: Receive ready (Data Carrier Detect line)
Bit 2: Clear-to-send (CTS) line
Bit 3: Incoming call (Ring Indicator line)
Bit 4: Depends on cable option or adapter used

Status 8   Returns modem driver line states.

Control 8   Sets modem driver line states (values shown are for male cable connector option for connection to modems).

Bit 0: Request-to-send (RS or RTS) line   1 = line set (active)
Bit 1: Data Terminal Ready (DTR) line   0 = line clear (inactive)
Bit 2: Driver 1: Data Rate Select
Bit 3: Driver 2: Depends on cable option or adapter used
Bit 4: Driver 3: Depends on cable option or adapter used
Bit 5: Driver 4: Depends on cable option or adapter used
Bits 6,7: Not used

Reset value = 0 prior to connect. Post-connect value is handshake dependent.

Note that RTS line cannot be altered (except by OUTPUT or OUTPUT...END) for half-duplex modem connections.

Status 9   Returns control block TYPE if last ENTER terminated on a control block. See Status Register 10 for values.

Status 10   Returns control block MODE if last ENTER terminated on a control block.

Async Protocol Control Blocks

Type	Mode	Interpretation
250	1	Break received (Channel A)
251	1^1	Framing error in the following character
251	2^1	Parity error in the following character
251	3^1	Parity and framing errors in the following character
252	1	End-of-line terminator detected
253	1	Prompt received from remote
0	0	No Control Block encountered

Data Link Protocol Control Blocks

Type	Mode	Interpretation
254	1	Preceding block terminated by ETB character
254	2	Preceding block terminated by ETX character
253²	—	(see following table for Mode interpretation)
0	0	No Control Block encountered.

Mode Bit(s)	Interpretation
0	1 = Transparent data in following block
	0 = Normal data in following block
2,1	00 = Device select
	01 = Group select
	10 = Line select
3	1 = Command channel
	2 = Data channel

Status 11   Returns available outbound queue space (in bytes), provided there is sufficient space for at least three control blocks. If not, value is zero.

---

[1] Parity framing error control blocks are not generated when characters with parity and/or framing errors are replaced by an underscore (_) character

[2] This type is used primarily in specialized applications.

Status 12 Datacomm Line connection status

Value	Interpretation
0	Disconnected
1	Attempting Connection
2	Dialing
3	Connected[1]
4	Suspended
5	Currently receiving data (Data Link only)
6	Currently transmitting data (Data Link only)

Note

When the datacomm line is suspended, CLEAR, ABORT, or RESET
must be executed before the line can be reconnected.

Reset value − 0 if R̄ on interface select code switch cluster is ON (1).

Control 12 Connects, initiates auto-dial sequence, and disconnects interface from datacomm line.

Value	Interpretation
0	Disconnect from datacomm line
1	Connect to datacomm line (set DTR & RTS)
2	Start auto dial. (Followed by OUTPUT of telephone numbers)

Status 13 Returns current ON INTR mask
Control 13 Sets ON INTR mask[2]

Data Link Protocol:

Bit	Value	Enables interrupt when:
0	1	A full block is available in receive queue
1	2	Transmit queue is empty
2	4	Receive or transmit error detected
3	8	A modem line changed
4	16[3]	No Activity timeout forced a disconnection
5	32[3]	Lost Carrier or Connection timeout caused a disconnection

Async Protocol:

Bit	Value	Enables interrupt when:
0	1	Data or control block available in receive queue
1	2	Prompt received from remote device
2	4	Framing or parity error detected in incoming data
3	8	A modem line changed
4	16[3]	No Activity timeout forced a disconnection
5	32[3]	Lost Carrier or Connection timeout caused a disconnection
6	64	End-of-line received
7	128	Break received

Reset value = 0

[1] When using Data Link: Connected - datacomm idle

[2] If a CONTROL statement is used to access this register, the control block is placed in the outbound queue. If the ENABLE INTR... statement is used with a mask, the mask value is placed directly in the control register, bypassing any queue delays.

[3] If bits 4 and 5 are not set, the corresponding errors can be trapped by using an ON ERROR statement.

Status 14 Returns current Control Block mask.

Control 14 Sets Control Block mask. Control block information is queued sequentially with incoming data as follows:

Bit	Value	Async Control Block Passed	Data Link Control Block Passed
0	1	Prompt position	Transparent/Normal Mode[1]
1	2	End-of-line position	ETX Block Terminator[2]
2	4	Framing and/or Parity error[3]	ETB Block Terminator[2]
3	8	Break received	
Reset Value:		**0** (Control Blocks disabled)	**6** (ETX/ETB Enabled)

Bits 4, 5, 6, and 7 are not used.

Status 15 Returns current modem line interrupt mask.

Control 15 Sets modem line interrupt mask. Enables an interrupt to ON INTR when Bit 3 of Control Register 13 is set as follows:

Bit	Value	Modem Line to Cause Interrupt
0	1	Data Mode (Data Set Ready)
1	2	Receive Ready (Data Carrier Detect)
2	4	Clear-to-send
3	8	OCR1, Incoming Call (Ring Indicator)
4	16	OCR2, Cable or adapter dependent

Reset Value = 0

Note that bit functions are the same as for STATUS register 7. Functions shown are for male connector cable option for modem connections.

Status 16 Returns current connection timeout limit.

Control 16 Sets Attempted Connection timeout limit.
Acceptable values: 1 thru 255 seconds.   0 = timeout disabled.
Reset Value = 25 seconds

Status 17 Returns current No Activity timeout limit.

Control 17 Sets No Activity timeout limit.
Acceptable values: 1 thru 255 minutes.   0 = timeout disabled.
Reset Value = 10 minutes (disabled if Async, non-modem handshake).

Status 18 Returns current Lost Carrier timeout limit.

Control 18 Sets Lost Carrier timeout limit in units of 10 ms.
Acceptable values: 1 thru 255.   0 = timeout disabled.
Reset Value = 40   (400 milliseconds)

---

[1] Transparent/Normal format identification control block occurs at the BEGINNING of a given block of data in the receive queue.

[2] ETX and ETB Block Termination identification control blocks occur at the END of a given block of data in the receive queue.

[3] This control block precedes each character containing a parity or framing error.

Status 19 Returns current Transmit timeout limit.

Control 19 Sets Transmit timeout limit (loss of clock or CTS not returned by modem when transmission is attempted).

Acceptable values: 1 thru 255. 0 = timeout disabled.

Reset Value = 10 seconds

Status 20 Returns current transmission speed (baud rate). *See* table for values.

Control 20 Sets transmission speed (baud rate) as follows:

Register Value	Baud Rate	Register Value	Baud Rate
0	External Clock	8	600
*1	50	9	1200
*2	75	10	1800
*3	110	11	2400
*4	134.5	12	3600
*5	150	13	4800
*6	200	14	9600
7	300	15	19200

* Async only. These values cannot be used with Data Link. These values set transmit speed ONLY for Async; transmit AND receive speed for Data Link. Default value is defined by the interface card configuration switches.

Status 21 Protocol dependent. Returns receive speed (Async) or GID address (Data Link) as specified by Control Register 21.

Control 21 Protocol dependent. Functions are as follows:

Data Link: Sets Group IDentifier (GID) for terminal. Values 0 thru 26 correspond to identifiers @, A, B,...Y, Z, respectively. Other values cause an error. Default value is 1 ("A").

Async: Sets datacomm receiver speed (baud rate). Values and defaults are the same as for Control Register 20.

Status 22 Protocol dependent. Returns DID (Data Link) or protocol handshake type (Async) as specified by Control Register 22.

Control 22 Protocol dependent. Functions are as follows:

Data Link: Sets Device IDentifier (DID) for terminal. Values are the same as for Control Register 21. Default is determined by interface card configuration switches.

Async: Defines protocol handshake type that is to be used.

Value	Handshake type
0	Protocol handshake disabled
1	ENQ/ACK with desktop computer as the host
2	ENQ/ACK, desktop computer as a terminal
3	DC1/DC3, desktop computer as host
4	DC1/DC3, desktop computer as a terminal
5	DC1/DC3, desktop computer as both host and terminal

Status 23 Returns current hardware handshake type.
Control 23 Sets hardware handshake type as follows:

0 = Handshake OFF, non-modem connection.
1 = FULL-DUPLEX modem connection.
2 = HALF-DUPLEX modem connection.
3 = Handshake ON, non-modem connection.
Reset Value is determined by interface configuration switches.

Status 24 Protocol dependent. Returns value set by preceding CONTROL statement to Control Register 24.
Control 24 Protocol dependent. Functions as follows:
Data Link protocol: Set outbound block size limit.

Value	Block size	Value	Block size
0	512 bytes	4	8 bytes
1	2 bytes	.	.
2	4 bytes	.	.
3	6 bytes	255	510 bytes

Reset outbound block size limit = 512 bytes

Async Protocol: Set mask for control characters included in receive data message queue.
Bit set: transfer character(s).
Bit cleared: delete character(s).

Bit set	Value	Character(s) passed to receive queue
0	1	Handshake characters (ENQ, ACK, DC1, DC3)
1	2	Inbound End-of-line character(s)
2	4	Inbound Prompt character(s)
3	8	NUL (CHR$(0))
4	16	DEL (CHR$(127))
5	32	CHR$(255)
6	64	Change parity/framing errors to underscores (_) if bit is set.
7	128	Not used

Reset value = 127 (bits 0 thru 6 set)

Status 25 Returns number of received errors since power up or reset.

Note

Control Registers 26 through 35, Status Registers 27 through 35, and Control and Status Registers 37 and 39 are used for ASYNC protocol ONLY. They are not available during Data Link operation.

Status 26 Protocol dependent
Data Link protocol: Returns number of transmit errors (NAKs received) since last interface reset.
Async protocol: Returns first protocol handshake character (ACK or DC1).
Control 26 Sets first protocol handshake character as follows:
(Async only) 6 = ACK, 17 = DC1. Other values used for special applications only. **Reset value = 17** (DC1). Use ACK when Control Register 22 is set to 1 or 2. Use DC1 when Control Register 22 is set to 3, 4, or 5.

Status 27 Returns second protocol handshake character.
(Async only)
Control 27 Sets second protocol handshake character as follows:
(Async only) 5 = ENQ, 19 = DC3. Other values used for special applications only. **Reset value = 19** (DC3). Use ENQ when Control Register 22 is set to 1 or 2. Use DC3 when Control Register 22 is set to 3, 4, or 5.

Status 28 Returns number of characters in inbound
(Async only) End-of-line delimiter sequence.
Control 28 Sets number of characters in End-of-line delimiter sequence
(Async only) Acceptable values are 0 (no EOL delimiter), 1, or 2. **Reset Value = 2**

Status 29 Returns first End-of-line character.
(Async only)
Control 29 Sets first End-of-line character. **Reset Value = 13** (carriage return)
(Async only)

Status 30 Returns second End-of-line character.
(Async only)
Control 30 Sets second End-of-line character. **Reset Value = 10** (line feed)
(Async only)

Status 31 Returns number of characters in Prompt sequence.
(Async only)
Control 31 Sets number of characters in Prompt sequence.
(Async only) Acceptable values are 0 (Prompt disabled), 1 or 2.
Reset Value = 1

Status 32 Returns first character in Prompt sequence.
(Async only)
Control 32 Sets first character in Prompt sequence.
(Async only) **Reset Value = 17** (DC1)

Status 33 Returns second character in Prompt sequence.
(Async only)
Control 33 Sets second character in Prompt sequence.
(Async only) **Reset Value = 0** (null)

Status 34 Returns the number of bits per character.
(Async only)
Control 34 Sets the number of bits per character as follows:
(Async only) 0 = 5 bits/character   2 = 7 bits/character
             1 = 6 bits/character   3 = 8 bits/character)
             When 8 bits/char, parity must be NONE, ODD, or EVEN.
             Reset Value is determined by interface card default switches.

Status 35   Returns the number of stop bits per character.
(Async only)
Control 35   Sets the number of stop bits per character as follows:
(Async only)   0 = 1 stop bit   1 = 1.5 stop bits   2 = 2 stop bits
   Reset Value: 2 stop bits if 150 baud or less, otherwise 1 stop bit.
   Reset Value is determined by interface configuration switch settings.

Status 36   Returns current Parity setting.
Control 36   Sets Parity for transmitting and receiving as follows:

   Data Link Protocol: 0 = NO Parity; Network host is HP 1000 Computer.
                                1 = ODD Parity; Network host is HP 3000 Computer.
                                Reset Value = 0
   Async Protocol   :   0 = NONE; no parity bit is included with any characters.
                                1 = ODD; Parity bit SET if there is an EVEN number of
                                     "1"s in the character body.
                                2 = EVEN; Parity bit OFF if there is an ODD number of
                                     "1"s in the character body.
                                3 = "0"; Parity bit is always ZERO, but parity is not checked.
                                4 = "1"; Parity bit is always SET, but parity is not checked.

   Default is determined by interface configuration switches. If 8 bits per character, parity
   must be NONE, ODD, or EVEN.

Status 37   Returns inter-character time gap in character times.
(Async only)
Control 37   Sets inter-character time gap in character times.
(Async only)   Acceptable values: 1 thru 255 character times.
   0 = No gap between characters. **Reset Value = 0**

Status 38   Returns Transmit queue status.
   If returned value = 1, queue is empty, and there are no pending transmissions.

Status 39   Returns current Break time (in character times).
(Async only)
Control 39   Sets Break time in character times.
(Async only)   Acceptable values are: 2 thru 255.   **Reset Value = 4.**

The RS-232 Serial Interface	Chapter 14

Introduction

The HP 98626 Serial Interface is an RS-232C compatible interface used for simple asynchronous I/O applications such as driving line printers, terminals, or other peripherals where the more sophisticated capabilities of the HP 98628 Data Communications Interface (see Chapter 12) are not justified. It uses a UART (Universal Asynchronous Receiver and Transmitter) integrated circuit to generate the required async signals. The Series 200 computer must provide most control functions because the card does not have its own processor capability. Consequently, there is more interaction between the card and computer than when you use a more intelligent interface except for relatively simple applications. An HP 98644 Serial Interface is also available. While there are some differences in the two interface cards, you should be able to coordinate information from this chapter and information from the interface documentation.

The RS-232C interface standard[1] establishes electrical and mechanical interface requirements, but does not define the exact function of all the signals that are used by various manufacturers of data communications equipment and serial I/O devices. Consequently, when you plug your serial interface into an RS-232 connector, there is no guarantee the devices can communicate unless you have configured optional parameters to match the requirements of the device you are connecting to.

Asychronous Data Communication

The terms Asynchronous (Async for short) data communication and Serial I/O refer to a technique of transferring information between two communicating devices by means of bit-serial data transmission. This means that data is sent, one bit at a time, and that characters are not synchronized with preceding or subsequent data characters; that is, each character is sent as a complete entity without relationship to other events, before or after. Characters may be sent in close succession, or they may be sent sporadically as data becomes available. Start and stop bits are used to identify the beginning and end of each character, with the character data placed between them.

Character Format
Each character frame consists of the following elements:

- **Start Bit:** The start bit signals the receiver that a new character is being sent. Since the receiver knows how many bits per second are being transmitted (specified by the baud rate), it can determine the expected arrival time for all subsequent bits in that character frame. All other bits in a given frame are synchronized to the start bit.

---

[1] RS-232C is a data communication standard established and published by the Electronic Industries Association (EIA). Copies of the standard are available from the association at 2001 Eye Street N. W., Washington D. C. 20006. Its equivalent for European applications is CCITT V.24.

- **5-8 Character Data Bits:** The next bits are the binary code of the character being transmitted, consisting of 5, 6, 7, or 8 bits; depending on the application. The parity bit is not included in the character data bits.

- **Parity Bit:** The parity bit is optional, included only when parity is enabled.

- **Stop Bit(s):** One or more stop bits identify the end of each character. The serial interface has no provision for inserting time gaps between characters.

Here is a simple diagram showing the structure of an asynchronous character and its relationship to other characters in the data stream:
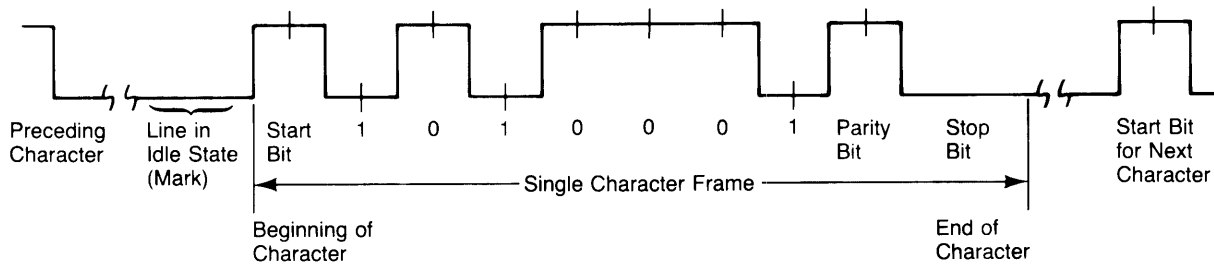
Parity

The parity bit is used to detect errors as incoming characters are received. If the parity bit does not match the expected sense, the character is assumed to be incorrectly received. The action taken when an error is detected depends upon how the interface and the desktop computer program are configured.

Parity sense is determined by system requirements. The parity bit may be included or omitted from each character by enabling or disabling the parity function. If the parity bit is enabled, four options are available. Parity is checked by the receiver for all parity options including ONE and ZERO. (The HP 98046 and HP 98628 Datacomm Interfaces do not check parity when parity is set to ONE or ZERO.)

Parity options include:

- **NONE**   Parity function is DISABLED, and the parity bit is omitted from each character frame.
- **ODD**   Parity bit is SET if there is an EVEN[1] number of ones in the data character. The receiver performs parity checks on incoming characters.
- **EVEN**   Parity bit is SET if there is an ODD[1] number of ones in the data character. The receiver performs parity checks on incoming characters.
- **ONE**   Parity bit is set for all characters. Parity is checked by the receiver on all incoming characters.
- **ZERO**   Parity bit is cleared, but present for all characters. Parity is checked by the receiver on all characters.

[1] Parity sense is determined by counting the number of ones in the character INCLUDING the parity bit. Consequently, the parity sense is reversed from the number of ones in a character without the parity bit.

Error Detection
Two types of incoming data errors can be detected by serial receivers:

- Parity errors are signalled when the parity bit does not match the number of ones, including the parity bit, even or odd as defined by interface configuration. When parity is disabled, no parity check is made.
- Framing errors are signaled when start and stop bits are not properly received during the expected time frame. They can be caused by a missing start bit, noise errors near the end of the character, or by improperly specified character length at the transmitter or receiver.

Two additional error types are detected by the receiver section of the serial interface:

- Overrun errors result when the desktop computer does not consume characters as fast as they arrive. The card provides only one character of buffer space, so the current character must be consumed by an ENTER before the next character arrives. Otherwise, the character is lost when the next character replaces it, and an error is sent to BASIC.
- Received BREAKs are detected as a special type of framing error. They generate the same type of BASIC error as framing errors.

Data Transfers Between Computer and Peripheral

Four statements are used to transfer information between your desktop computer and the interface card:

- The CONTROL statement is used to control interface operation and defines such parameters as baud rate, character format, or parity.
- The OUTPUT statement sends data to the interface which, in turn, sends the information to the peripheral device.
- The ENTER statement inputs data from the interface card after the interface has received it from the peripheral device.
- The STATUS statement is used to monitor the interface and obtain information about interface operation such as buffer status, detected errors, and interrupt enable status.

Since the interface has no on-board processor, ENTER and OUTPUT statements cause the computer to wait until the ENTER or OUTPUT operation is complete before continuing to the next line. For OUTPUT statements, this means that the computer waits until the last bit of the last character has been sent over the serial line before continuing with the next program statement.

Overview of Serial Interface Programming

Serial interface programming techniques are similar to most general I/O applications. The interface card is initialized by use of CONTROL statements; STATUS statements evaluate its readiness for use. Data is transferred between the desktop computer and a peripheral device by OUTPUT and ENTER statements. In most cases, default configuration switches on the interface card can be used to eliminate or significantly reduce the need for using CONTROL statements to initialize the card.

Due to the asynchronous nature of serial I/O operations, special care must be exercised to ensure that data is not lost by sending to another device before the device is ready to receive. Modem line handshaking can be used to help solve this problem. These and other topics are discussed in greater detail elsewhere in this chapter.

Initializing the Interconnection

Determining Operating Parameters

Before you can successfully transfer information to a device, you must match the operating characteristics of the interface to the corresponding characteristics of the peripheral device. This includes matching signal lines and their functions as well as matching the character format for both devices.

Hardware Parameters

To determine hardware operating parameters, you need to know the answer for each of the following questions about the peripheral device:

- Which of the following signal and control lines are actively used during communication with the peripheral?

  __ Data Set Ready (DSR)   __ Data Carrier Detect (DCD or CD)
  __ Clear to Send (CTS)   __ Ring Indicator (RI)

- What baud rate (line speed) is expected by the peripheral?

Character Format Parameters

To define the character format, you must know the requirements of the peripheral device for the following parameters:

- **Character Length:** How many data bits are used for each character, excluding start, stop, and parity bits?
- **Parity Enable:** Is parity enabled (included) or disabled (absent) for each character?
- **Parity Sense:** Is the parity bit, if enabled, ODD, EVEN, always ONE, or always ZERO?
- **Stop Bits:** How many stop bits are included with each character: 1, 1.5, or 2?

Using Interface Defaults to Simplify Programming

The serial interface includes three default configuration switch clusters in addition to the select code and interrupt level switches. Their functions are described in the following paragraphs.

Modem Line Disconnect Switches

The Modem Line Disconnect switches are used to connect or disconnect the following modem lines from the interface cable:

- Data Set Ready (DSR)
- Data Carrier Detect (DCD or CD)
- Clear to Send (CTS)
- Ring Indicator (RI)

When a given switch is in the CONNECT position, the corresponding modem line is connected from the peripheral device to the interface circuitry. When it is in the disconnected position, the modem line is disconnected, and the interface receiver input for that line is held HIGH (true). Any modem lines that are not actively used while communicating with the peripheral should be disconnected to minimize errors due to electrical noise in the cable. **Modem line disconnect switch settings cannot be altered under program control.** To reconfigure the switches, the interface must be removed from the computer, and the settings changed by hand.

Note

The Model 216 computer has an internal serial interface which has no switches. Because switch settings can vary, cable connections between the Model 216 and an external device can require some cross-wiring. Use of a breakout box can be helpful.

Baud Rate Select Switches

The rate at which data bits are transferred between the interface and the peripheral is called the baud rate. The interface card must be set to transmit and receive at the same rate as the peripheral, or data cannot be successfully transferred. To preset the baud rate, the Baud Rate Select switches can be set to any one of the following values:

Baud Rate	Switch Settings 3 2 1 0	Baud Rate	Switch Settings 3 2 1 0
50	0 0 0 0	1200	1 0 0 0
75	0 0 0 1	1800	1 0 0 1
110	0 0 1 0	2400	1 0 1 0
134.5	0 0 1 1	3600	1 0 1 1
150	0 1 0 0	4800	1 1 0 0
200	0 1 0 1	7200	1 1 0 1
300	0 1 1 0	9600	1 1 1 0
600	0 1 1 1	19200	1 1 1 1

Line Control Switches
The Line Control switches are used to preset character format and parity options. Functions are as follows:

Parity Sense (Switches 5&4)		Parity Enable (Switch 3)		Stop Bits (Switch 2)		Character Length (Switches 1&0)	
00	ODD parity	0	Disabled	0	1 stop bit	00	5 bits/char
01	EVEN parity	1	Enabled	1	1.5 stop bits	01	6 bits/char
10	Always ONE				if 5 bits/char	10	7 bits/char
11	Always ZERO				or 2 stop bits	11	8 bits/char
					if 6, 7, or 8		
					bits/char		

Bits 6 and 7 are reserved for future use.

Using Program Control to Override Defaults

You can override some of the interface default configuration options by use of CONTROL statements. This not only enables you to guarantee certain parameters, but also provides a means for changing selected parameters in the course of a running program. Control Register tables are listed at the end of this chapter as well as in the *BASIC Language Reference*. Refer to them as needed during the discussion which follows.

Interface Reset
Whenever an interface is connected to a modem that may still be connected to a telecommunications link from a previous session, it is good programming practice to reset the interface to force the modem to disconnect, unless the status of the link and remote connection are known. When the interface is connected to a line printer or similar peripheral, resetting the interface is usually unnecessary unless an error condition requires it.

When the interface is reset by use of a CONTROL statement to Control Register 0 with a non-zero value, the interface is restored to its power-up condition, except that the current character format is not altered, whether or not it is the same as the current default switch configuration. If you are not sure of the present settings, or if your application requires changing the configuration during program operation, you can use CONTROL statements to configure the interface. An example of where this may be necessary is when several peripherals share a single interface through a manually operated RS-232 switch such as those used to connect multiple terminals to a single computer port, or a single terminal to multiple computers.

Selecting the Baud Rate
In order to successfully transfer information between the interface card and a peripheral, the interface and peripheral must be set to the same baud rate. A CONTROL statement to register 3 can be used to set the interface baud rate to any one of the following values:

50	150	1200	4800
75	200	1800	7200
110	300	2400	9600
134.5 (or 134)	600	3600	19 200

For example, to select a baud rate of 3600, the following program statement is used:

```
1190   CONTROL Sc,3;3600
```

Use of values other than those shown may result in incorrect operation.

To verify the current baud rate setting, use a STATUS statement addressed to register 3. All rates are in baud (bits/second).

Setting Character Format and Parity
Control Register 4 overrides the Line Control switches that control parity and character format. To determine the value sent to the register, add the appropriate values selected from the following table:

Parity Sense		Parity Enable		Stop Bits		Character Length	
0	ODD Parity	0	Disabled	0	1 stop bit	0	5 bits/char
16	EVEN Parity	8	Enabled	4	1.5 stop bits	1	6 bits/char
32	Always ONE				if 5 bits/char	2	7 bits/char
48	Always ZERO				or 2 stop bits	3	8 bits/char
					if 6, 7, or 8		
					bits/char		

For example, to configure a character format of 8 bits per character, two stop bits, and EVEN parity, use the following CONTROL statement:

```
1200   CONTROL Sc,4;3+4+8+16
```

or

```
1200   CONTROL Sc,4;31
```

To configure a 5-bit character length with 1 stop bit and no parity bit, use the following:

```
1200   CONTROL Sc,4;0
```

Data Transfers

The serial interface card is designed for relatively simple serial I/O operations. It is not intended for sophisticated applications that use ON INTR statements extensively to service the interface. If your situation requires full interrupt capability such as in terminal emulator applications, use the HP 98628 Datacomm Interface instead. Limited ON INTR capabilities are provided by the serial interface for error trapping and other simple tasks.

Program Flow

When the interface is properly configured, either by use of default switches or CONTROL statements, you are ready to begin data transfers. OUTPUT statements are used to send information to the peripheral; ENTER statements to input information from the external device. Any valid OUTPUT or ENTER statement and variable(s) list may be used, but you must be sure that the data format is compatible with the peripheral device. For example, non-ASCII data sent to an ASCII line printer results in unpredictable behavior.

Various other I/O statements can be used in addition to OUTPUT and ENTER, depending on the situation. For example, the LIST statement can be used to list programs to an RS-232 line printer PROVIDED the interface is properly configured before the operation begins.

Data Output

To send data to a peripheral, use OUTPUT, OUTPUT USING, or any other similar or equivalent construct. Suppression of end-of-line delimiters and other formatting capabilities are identical to normal operation in general I/O applications. The OUTPUT statement hangs the computer until the last bit of the last character in the statement variable list is transmitted by the interface. When the output operation is complete, the computer then continues to the next line in the program.

Data Entry

To input data from a peripheral, use ENTER, ENTER USING, or an equivalent statement. Inclusion or elimination of end-of-line delimiters and other information is determined by the formatting specified in the ENTER statement. The ENTER statement hangs the computer until the input variables list is satisfied. To minimize the risk of waiting for another variable that isn't coming, you may prefer to specify only one variable for each ENTER statement, and analyze the result before starting the next input operation.

Be sure that the peripheral is not transmitting data to the interface while no ENTER is in progress. Otherwise, data may be lost because the card provides buffering for only one character. Also, interrupts from other I/O devices, or operator inputs to the computer keyboard can cause delays in computer service to the interface that result in buffer overrun at higher baud rates.

Modem Line Handshaking

Modem line handshaking, when used, is performed automatically by the computer as part of the OUTPUT or ENTER operation. If the modem line states have not been latched in a fixed state by Control Register 5, the following sequence of events is executed automatically during each OUTPUT or ENTER operation:

For OUTPUT operation,

1. Set Data Terminal Ready and Request-to-Send modem lines to active state.
2. Check Data Set Ready and Clear-to-Send modem lines to be sure they are active.
3. Send information to the interface and thence to the peripheral.
4. After data transfer is complete, clear Data Terminal Ready and Request-to-Send signals.

For ENTER operation,

1. Set Data Terminal Ready line to active state. Leave Request-to-Send inactive.
2. Check Data Set Ready and Data Carrier Detect modem lines to be sure they are active.
3. Input information from the interface as it is received from the peripheral.
4. After the input operation is complete, clear the Data Terminal Ready signal.

After a given OUTPUT or ENTER operation is completed, the program continues execution on the next line.

Control Register 5 can be used to force selected modem control lines to their active state(s). The Data Rate Select and Secondary Request-to-Send lines are set or cleared by bits 3 and 2 respectively. Request-to-send and Data Terminal Ready are held in their active states when bits 1 and 0 are true, respectively. If bits 1 and/or 0 are false, the corresponding modem line is toggled during OUTPUT or ENTER as explained previously.

Incoming Data Error Detection and Handling

The serial interface card can generate several errors that are caused when certain conditions are encountered while receiving data from the peripheral device. The UART detects a given error condition and sets the corresponding bit in Status Register 10. The card then generates a pending error to BASIC. Errors can be generated by any of the following conditions:

- Parity error. The parity bit on an incoming character does not match the parity expected by the receiver. This condition is most commonly caused by line noise. When this error occurs, bit 2 of Status Register 10 is set.
- Framing error. Start and stop bit(s) do not match the timing expectations of the receiver. This can occur when line noise causes the receiver to miss the start bit or obscures the stop bits. When this error is detected, bit 3 of Status Register 10 is set.
- Overrun error. Incoming data buffer overrun caused a loss of one or more data characters. This is usually caused when data is received by the interface, but no ENTER statement has been activated to input the information. Bit 1 of Status Register 10 is set when this error occurs.
- Break received. A BREAK was sent to the interface by the peripheral device. The desktop computer program must be able to properly interpret the meaning of a break and take appropriate action. When this condition occurs, bit 4 of Status Register 10 is set. Since a BREAK is detected as a special type of framing error, the framing error indicator, bit 3, is also set.

All UART status errors are generated by INCOMING data, never by outbound data. When a UART error occurs, the corresponding bit of Status Register 10 is set, and a pending error (ERROR 167: Interface status error) is sent to BASIC. BASIC processes the error according to the following rules:

- If an ENTER is in progress, the error is handled immediately as part of the ENTER process. An active ON ERROR causes the error trap to be executed. If no ON ERROR is active, the error is fatal and causes the program to terminate.

- If an OUTPUT is in progress, or if there is no current activity between the computer and interface, the error is flagged, but nothing is done by BASIC until an ENTER statement is encountered. When the computer begins execution of the ENTER statement, if an ON ERROR is active, the error trap is executed. If there is no active ON ERROR for that select code, the fatal ERROR 167 causes the BASIC program to terminate.

- If a STATUS statement is executed to Status Register 10 before an ENTER statement is encountered for that select code, the pending BASIC error is cleared, and the program continues as if no error had been generated. Whenever a STATUS statement is executed to Status Register 10, bits 1 through 4 of the register are cleared and the data is destroyed. If you need to perform multiple operations (such as IF BIT tests) on the register contents, be sure to store the information in a variable before you use it.

Trapping Serial Interface Errors

Pending BASIC errors can be trapped by using an ON ERROR statement in conjunction with an error trapping service routine to evaluate the error condition. Here is an example technique:

```
1200   Sc=9                      ! Set serial interface select code.
1210   ON ERROR GOTO Error       ! Set up error trap routine.
   .
   .
1400   ENTER Sc;A$               ! Input line of data from interface.
   .
   .
1530 Error:          ! Error trap routine:
1535     IF ERRN<>167 THEN Other_error
1540     STATUS Sc,10;Uart_error      ! Get UART error information.
1550     IF BIT (Uart_error,1) THEN Overrun    ! Overrun error.
1560     IF BIT (Uart_error,2) THEN Parity     ! Parity error.
1570     IF BIT (Uart_error,4) THEN Break      ! BREAK received.
1580     IF BIT (Uart_error,3) THEN Framing     ! Framing error.
1590 Other:                                    ! Other error type.
   .
   .
1650 Overrun:   ! Overrun error routine:
   .
   .
1700 Parity:    ! Parity error routine:
   .
   .
1750 Framing:   ! Framing error routine:
   .
   .
1800 Break:     ! BREAK received routine:
   .
   .
1850 Other_error: ! Not error 167. Process accordingly.
```

This example is not intended to show a specific application, but only to illustrate the technique for trapping interface errors. Only UART errors are shown in this example, but the technique is valid for other errors related to a given interface.

Note that in this example, the UART error information is checked for a BREAK before looking at the framing error bit. When a break is received, both the BREAK and framing error bits are set. Consequently, if the error check sequence were reversed, it would be necessary to check for a BREAK whenever a framing error is processed. Reversing the order eliminates an extra step by making it unnecessary to check for framing errors when a BREAK occurs. That is because whenever the BREAK is processed, the framing error is also cleared, making it unnecessary to perform any operations related to framing errors that are handled by the BREAK routine.

Special Applications

This section provides advanced programming information for applications requiring special techniques.

Sending BREAK Messages

A BREAK is a special character transmission that usually indicates a change in operating conditions. Interpretation of break messages varies with the application. To send a break message, send a non-zero value to Control Register 1 as follows (Sc is the interface select code):

```
1640   CONTROL Sc,1;1       ! Send a BREAK to peripheral.
```

Using the Modem Control Register

Control Register 5 controls various functions related to modem operation. Bits 0 thru 3 control modem lines, and bit 4 enables a self-test loopback configuration.

Modem Handshake Lines (RTS and DTR)

As explained earlier in this chapter, Request-to-send and Data Terminal Ready lines are set or cleared at the beginning and end of each OUTPUT or ENTER operation. In some cases, it may be advantageous or necessary to maintain either or both in an active state. This is done by setting bit 1 or 0 respectively in Control Register 5 as follows:

```
1650   CONTROL Sc,5;2       ! Set RTS line only and hold active.
1660   CONTROL Sc,5;1       ! Set DTR line only and hold active.
1670   CONTROL Sc,5;3       ! Set both RTS and DTR lines active.
1680   CONTROL Sc,5;0       ! Return to normal modem line handshake.
```

When RTS and/or DTR are set by Control Register 5, they are NOT toggled during OUTPUT or ENTER operations, but remain constantly in an active state until the control register is cleared by a CONTROL statement or an interface reset from Control Register 0 or the computer keyboard (SHIFT PAUSE or Reset).

Programming the DRS and SRTS Modem Lines

Bits 3 and 2 of Control Register 5 control the present state of the Data Rate Select (DRS) and Secondary Request-to-send (SRTS) lines, respectively. When either bit is set, the corresponding modem line is activated. When the bit is cleared, so is the modem line. To set both lines, the following statement or its equivalent can be used:

```
1690  CONTROL Sc,5;8+4   ! Set DRS and SRTS lines.
```

These lines are also cleared by a CONTROL statement to Control Register 5 with bits 2 and 3 cleared, or by an interface reset.

Configuring the Interface for Self-test Operations

Self-test programs can be written for the serial interface. Prior to testing the interface, it must be properly configured. Using bit 4 of Control Register 5, you can rearrange the interconnections between input and output lines on the interface, enabling the interface to feed outbound data to the inbound circuitry.

When LOOPBACK is enabled (bit 4 is set), the UART output is set to its MARK state and sent to the Transmitted Data (TxD) line. The output of the transmitter shift register is then connected to the input of the receiver shift register, causing outbound data to be looped back to the receiver. In addition, the following modem control lines are connected to the indicated modem status lines:

Modem Control Line		Modem Status Line	
DTR	Data Terminal Ready	CTS	Clear-to-send
RTS	Request-to-send	DSR	Data Set Ready
DCD	Data Carrier Detect	DRS	Data Rate Select
SRTS	Secondary RTS	RI	Ring Indicator

When loopback is active, receiver and transmitter interrupts are fully operational. Modem control interrupts are then generated by the modem control outputs instead of the modem status inputs. Refer to serial interface hardware documentation for information about card hardware operation.

READIO and WRITEIO Register Operations

For those cases where you need to write special interface driver routines, the interface card provides registers that can be accessed by use of READIO and WRITEIO statements. These capabilities are intended for use by experienced programmers who understand the inherent programming complexities that accompany this versatility.

Some registers are read/write; that is, both READIO and WRITEIO operations can be performed on a given register. Writing places a new value in the register; a read operation returns the current value. All registers have 8 bits available, and accept values from 0 through 255 unless noted otherwise. When the value of a given bit is 1, the bit is set; otherwise, it is zero (cleared or inactive).

Note

Some READIO and WRITEIO registers are similar in structure and function to Status and Control Registers. However, their interaction with the desktop computer operating system is considerably different. To prevent incorrect program operation, do NOT intermix the use of STATUS/CONTROL registers and READIO/WRITEIO registers in a given program.

Interface Registers

READIO and WRITEIO registers 1, 3, 5, and 7 access interface registers. Their functions are as follows:

Register 1: Interface Reset and ID

READIO of Register 1 returns the interface ID value: 2 for the HP 98626 Serial Interface. WRITEIO to Register 1 with any value, 1 thru 255, resets the interface as when using a CONTROL statement to Control Register 0.

Register 3: Interrupt Control

Only the upper four bits of Register 3 are used. Bits 5 and 4 return the setting of the Interrupt Level switches on the interface. Their values are as follows:

00	Interrupt Level 3	10	Interrupt Level 5
01	Interrupt Level 4	11	Interrupt Level 6

Bit 6 is set when an interrupt request is originated by the UART. No machine interrupt can occur unless bit 7, Interrupt Enable is set by a WRITEIO statement. Only bit 7 can is affected by WRITEIO statements. During READIO, bit 7 returns the current enable value; bits 6 thru 4 return interrupt request and level information.

Register 5: Optional Circuit and Baud Rate Control

WRITEIO to bits 7 and 6 control the state of optional circuit drivers 3 and 4, respectively. READIO returns current values of the respective drivers, plus the following:

Bit 5   Optional Circuit Receiver 2 state.
Bit 4   Optional Circuit Receiver 3 state.
Bits 3-0   Current Baud Rate switch setting (not necessarily the current UART baud rate) as follows:

Setting	Baud Rate	Setting	Baud Rate
0000	50	1000	1200
0001	75	1001	1800
0010	110	1010	2000
0011	134.5	1011	2400
0100	150	1100	3600
0101	200	1101	4800
0110	300	1110	7200
0111	600	1111	9600

Note that WRITEIO to this register can NOT be used to set the baud rate. Use Register 23, bit 7 and Registers 17 and 19 instead.

Register 7: Line Control Switch Monitor

READIO to this register enables you to input the present settings of the Line Control switches that preset default character format and parity. Bit functions are included in the table earlier in this chapter under Using Interface Defaults to simplify programming. Bits 7 thru 0 correspond to switches 7 thru 0, respectively. WRITEIO operations to this register are meaningless.

UART Registers
Addresses 17 through 29 access UART registers. They are used to directly control certain UART functions. The function of Registers 17 and 19 are determined by the state of bit 7 of Register 23.

Register 17: Receive Buffer/Transmitter Holding Register

When bit 7 of Register 23 is clear (0), this register accesses the single-character receiver buffer by use of READIO. A WRITEIO statement places a character in the transmitter holding register.

The receiver and transmitter are doubly buffered. When the transmitter shift register becomes empty, a character is transferred from the holding register to the shift register. You can then place a new character in the holding register while the preceding character is being transmitted. Incoming characters are transferred to the receiver buffer when the receiver shift register becomes full. You can then input the character (READIO) while the next character is being constructed in the shift register.

Registers 17 and 19: Baud Rate Divisor Latch

When bit 7 of Register 23 is set (1), Registers 17 and 19 access the 16-bit divisor latch used by the UART to set the baud rate. Register 17 forms the lower byte; Register 19 the upper. The baud rate is determined by the following relationship:

$$\text{Baud Rate} = 153\ 600/\text{Baud Rate Divisor}$$

To access the Baud Rate Divisor latch, set bit 7 of Register 23. This disables access to the normal functions of Registers 17 and 19, but preserves access to the other registers. When the proper value has been placed in the latch, be sure to clear bit 7 of Register 23 to return to normal operation.

Register 19: Interrupt Enable Register

When bit 7 of Register 23 is clear (0), this register enables the UART to interrupt when specified conditions occur. Only bits 0 thru 3 are used. WRITEIO establishes a new value for each bit; READIO returns the current register value. Interrupt enable conditions are as follows:

Bit 3   **Enable Modem Status Change Interrupts**, when set, enables an interrupt whenever a modem status line changes state as indicated by Register 29, bits 0 thru 3.

Bit 2   **Enable Receiver Line Status Interrupts**, when set, enables interrupts by errors, or received BREAKs as indicated by Register 27, bits 1 thru 4.

Bit 1   **Enable Transmitter Holding Register Empty Interrupt**, when set, allows interrupts when bit 5 of Register 27 is also set.

Bit 0   **Enable Receiver Buffer Full Interrupts**, when set, enables interrupts when bit 0 of Register 27 is also set.

Register 21: Interrupt Identification Register

This register identifies the cause of the highest-priority, currently-pending interrupt. Only bits 2, 1, and 0 are used. Bit 0, if set, indicates no interrupt pending. Otherwise an interrupt is pending as defined by bits 2 and 1. Causes of pending interrupts in order of priority are as follows:

Bits 2&1   Interrupt cause

11   Receiver Line Status interrupt (highest priority) is caused when bit 2 of Register 19 is set and a framing, parity, or overrun error, or a BREAK is detected by the receiver (indicated by bits 1 thru 4 of Register 27). The interrupt is cleared by reading Register 27.

10   Receive Buffer Register Full interrupt is generated when bit 0 of Register 19 is set and the Data Ready bit (bit 0) of Register 27 is active. To clear the interrupt, read the receiver buffer, or write a zero to bit 0 of Register 27.

01   Transmitter Holding Register Empty interrupt occurs when bit 1 of Register 19 is set and bit 5 of Register 27 is set. The interrupt is cleared by writing data into the transmitter holding register (Register 17 with bit 7 of Register 23 clear) with a WRITEIO statement, or by reading this register (Interrupt Identification).

00   Modem Line Status Change interrupt occurs when bit 3 of Register 19 is set and a modem line change is indicated by one or more of bits 0 thru 3 of Register 29. To clear the interrupt, read Register 29 which clears the status change bits.

Register 23: Character Format Control Register

This register is functionally equivalent to Control and Status Register 4 except for bits 6 and 7. WRITEIO sets a new character format; READIO returns the current character format setting.

Bit 7   **Divisor Latch Access Bit**, when set, enables you to access the divisor latches of the Baud Rate generator during read/write operations to registers 17 and 19.

Bit 6   **Set BREAK**, when set, holds the serial line in a BREAK state (always zero), independent of other transmitter activity. **This bit must be cleared to disable the break and resume normal activity.**

Bits 5,4   **Parity Sense** is determined by both bits 5 and 4. When bit 5 is set, parity is always ONE or ZERO. If bit 5 is not set, parity is ODD or EVEN as defined by bit 4. The combinations of bits 5 and 4 are as follows:

00	ODD parity	10	Always ONE
01	EVEN parity	11	Always ZERO

Bit 3   **Parity Enable**, when set, sends a parity bit with each outbound character, and checks all incoming characters for parity errors. Parity is defined by bits 4 and 5.

Bit 2   **Stop Bit(s)** are defined by a combination of bit 2 and bits 1 & 0.

Bit 2	Character Length	Stop Bits
0	5, 6, 7, or 8	1
1	5	1.5
1	6, 7, or 8	2

Bits 1&0 **Character Length** is defined as follows:

Bits 1&0	Character Length
00	5 bits
01	6 bits
10	7 bits
11	8 bits

Register 25: Modem Control Register

This is a READ/WRITE register. READIO returns current control register value. WRITEIO sets a new value in the register. This register is equivalent to interface Control Register 5.

Bit 4   **Loopback,** when set, enables a loopback feature for diagnostic testing. Serial line is set to MARK state, UART receiver is disconnected, and transmitter output shift register is connected to receiver input shift register. Modem line outputs and inputs are connected as follows: DTR to CTS, RTS to DSR, DRS to DCD, and SRTS to RI. Interrupts are enabled, with interrupts caused by modem control outputs instead of inputs from modem.

Bit 3   **Data Rate Select** controls the OCD1 driver output. 1 = Active, 0 = Disabled.

Bit 2   **Secondary Request-to-Send** controls the OCD2 driver output. 1 = Active, 0 = Disabled.

Bit 1   **Request-to-Send** controls the RTS modem control line state. When bit 1 = 1, RTS is always active. When bit 1 = 0, RTS is toggled by the OUTPUT statement as described earlier in this chapter.

Bit 0   **Data Terminal Ready** holds the DTR modem control line active when the bit is set. If not set, DTR is controlled by the OUTPUT or ENTER statement as described earlier.

Bits 7, 6, and 5 are not used.

Register 27: Line Status Register

Bit 7   Not used.

Bit 6   **Transmitter Shift Register Empty** indicates no data present in transmitter shift register.

Bit 5   **Transmitter Holding Register Empty** indicates no data present in transmitter holding register. The bit is cleared whenever a new character is placed in the register.

Bit 4   **Break Indicator** indicates that the received data input remained in the spacing (line idle) state for longer than the transmission time of a full character frame. This bit is cleared when the line status register is read.

Bit 3   **Framing Error** indicates that a character was received with improper framing; that is, the start and stop bits did not conform with expected timing boundaries.

Bit 2   **Parity Error** indicates that the received character did not have the expected parity sense. This bit is cleared when the register is read.

Bit 1   **Overrun Error** indicates that a character was destroyed because it was not read from the receiver buffer before the next character arrived. This bit is cleared by reading the line status register.

Bit 0   **Data Ready** indicates that a character has been placed in the receiver buffer register. This bit is cleared by reading the receiver buffer register, or by writing a zero to this bit of the line status register.

Register 29: Modem Status Register

Bit 7   **Data Carrier Detect**, when set, indicates DCD modem line is active.

Bit 6   **Ring Indicator**, if set, indicates that the RI modem line is active.

Bit 5   **Data Set Ready**, if set, indicates that the DSR modem line is active.

Bit 4   **Clear-to-send**, if set, indicates that CTS is active.

Bit 3   **Change in Carrier Detect**, when set, indicates that the DCD modem line has changed state since the last time the modem status register was read.

Bit 2   **Trailing Edge of Ring Indicator** is set when the RI modem line changes from active to inactive state.

Bit 1   **Delayed Data Set Ready** is set when the DSR line has changed state since the last time the modem status register was read.

Bit 0   **Change in Clear-to-send**, if set, indicates that the CTS modem line has changed state since the last time the register was read.

Cable Options and Signal Functions

The HP 98626A Serial Interface is available with RS-232C DTE and DCE cable configurations. The DTE cable option consists of a male RS-232C connector and cable designed to function as Data Terminal Equipment (DTE) when used with the serial interface. This means that the cable and connector are wired so that signal paths are correctly routed when the cable is connected to a peripheral device wired as Data Communication Equipment (DCE), such as a modem. The cables are designed so that you can write programs that work for both DCE and DTE connections without requiring modifications to accommodate equipment changes.

The DCE cable option includes a female connector and cable wired so that the interface and cable behave like normal DCE. This means that signals are routed correctly when the female cable connector is connected to a male DTE connector.

Line printers and other peripheral devices that use RS-232C interfacing are frequently wired as DTE with a female RS-232C chassis connector. This means that if you use a male (DTE) cable option to connect to the female DTE device connector, no communication can take place because the signal paths are incompatible. To eliminate the problem, use an adapter cable to convert the female RS-232C chassis connector to a cable connector that is compatible with the male or female interface cable connector. The HP 13242 adapter cable is available in various configurations to fit most common applications. Consult cable documentation to determine which adapter cable to use.

The DTE Cable

The signals and functions supported by the DTE cable are shown in the signal identification table which follows. The table includes RS-232C signal identification codes, CCITT V.24 equivalents, the pin number on the interface card rear panel connector, the RS-232C connector pin number, the signal mnemonic used in this manual, whether the signal is an input or output signal, and its function.

RS-232 DTE (male) Cable Signal Identification Tables

Signal RS-232C	V.24	Interface Pin #	RS-232C Pin #	Mnemonic	I/O	Function
AA	101	24	1	–	–	Safety Ground
BA	103	12	2		Out	Transmitted Data
BB	104	42	3		In	Received Data
CA	105	13	4	RTS	Out	Request to Send
CB	108	44	5	CTS	In	Clear to Send
CC	107	45	6	DSR	In	Data Set Ready
AB	102	48	7	–	–	Signal Ground
CF	109	46	8	DCD	In	Data Carrier Detect
SCF (OCR2)	122	47	12	SDCD	In	Secondary DCD
SCA (OCD2)	120	15	19	SRTS	Out	Secondary RTS
CD	108.1	14	20	DTR	Out	Data Terminal Ready
CE (OCR1)	125	9	22	RI	In	Ring Indicator
CH (OCD1)	111	40	23	DRS	Out	Data Rate Select

Optional Circuit Driver/Receiver Functions

Not all signals from the interface card are included in the cable wiring. RS-232C provides for four optional circuit drivers and two receivers. Only two drivers and two receivers are supported by the DCE and DTE cable options. They are as follows:

	Drivers		Receivers
Name	Function	Name	Function
OCD1	Data Rate Select	OCR1	Ring Indicator
OCD2	Secondary Request-to-send	OCR2	Secondary Data Carrier Detect
OCD3	Not used		
OCD4	Not used		

If your application requires use of OCD3 or OCD4, you must provide your own interface cable to fit the situation.

The DCE Cable

The DCE cable option is designed to adapt a DTE cable and serial or data communications interface to an identical interface on another desktop computer. It is also used with the serial interface to simulate DCE operation when driving a peripheral wired for DTE operation. The DCE cable is equipped with a female connector. Since most DTE peripherals are also equipped with female connectors (pin numbering is the same as the standard male DTE connector), an adapter (such as the HP 13242M) is used to connect the two female connectors as explained earlier.

Note

Not all RS-232C devices are wired the same. To ensure proper operation, you must know whether the peripheral device is wired as DTE or DCE. The interface cable option and associated adapter cable, if needed, must be configured to properly mate with the female DTE chassis connector.

The following schematic diagram shows the input and output signals for the serial interface and how they are connected to a DCE peripheral.
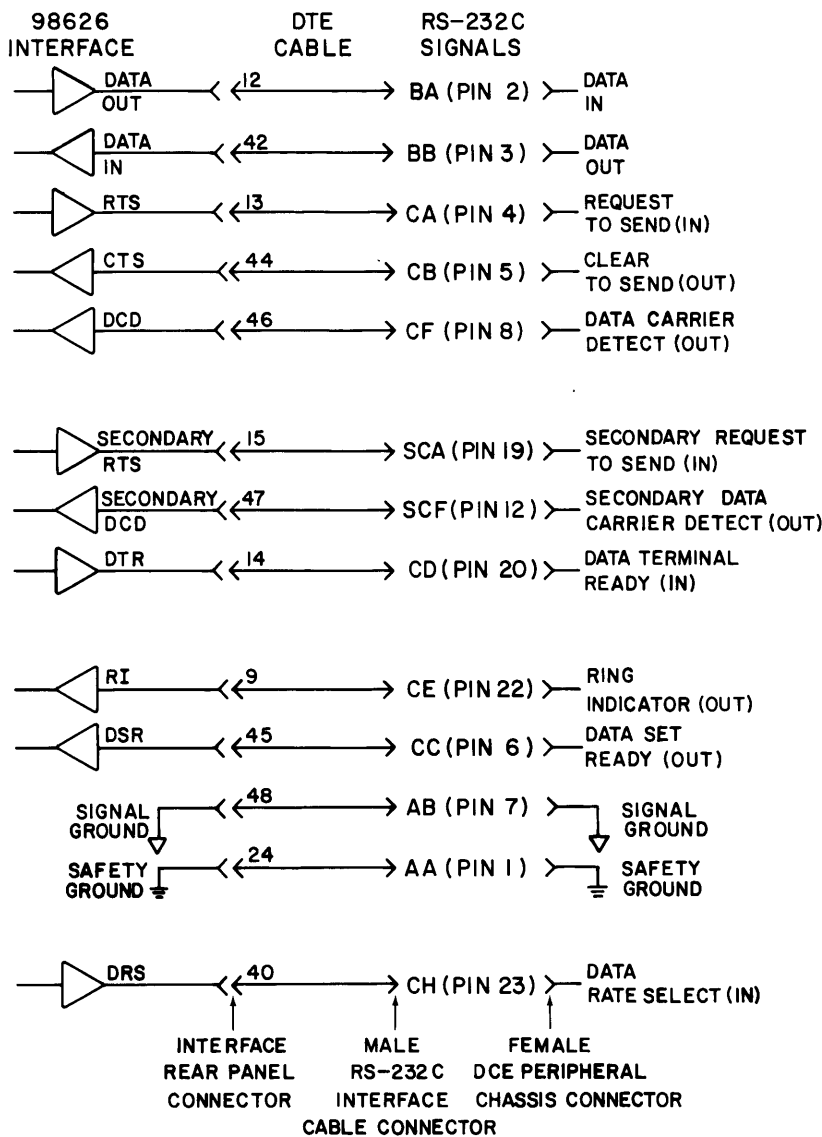
DTE Cable Interconnection Diagram

This diagram shows an HP 13242M adapter cable connected to a DCE interface cable and a DTE peripheral. Note that RTS is connected to CTS in the DCE cable. If your peripheral uses RTS/CTS handshaking, a different adapter cable must be used with the appropriate DTE or DCE interface cable option.
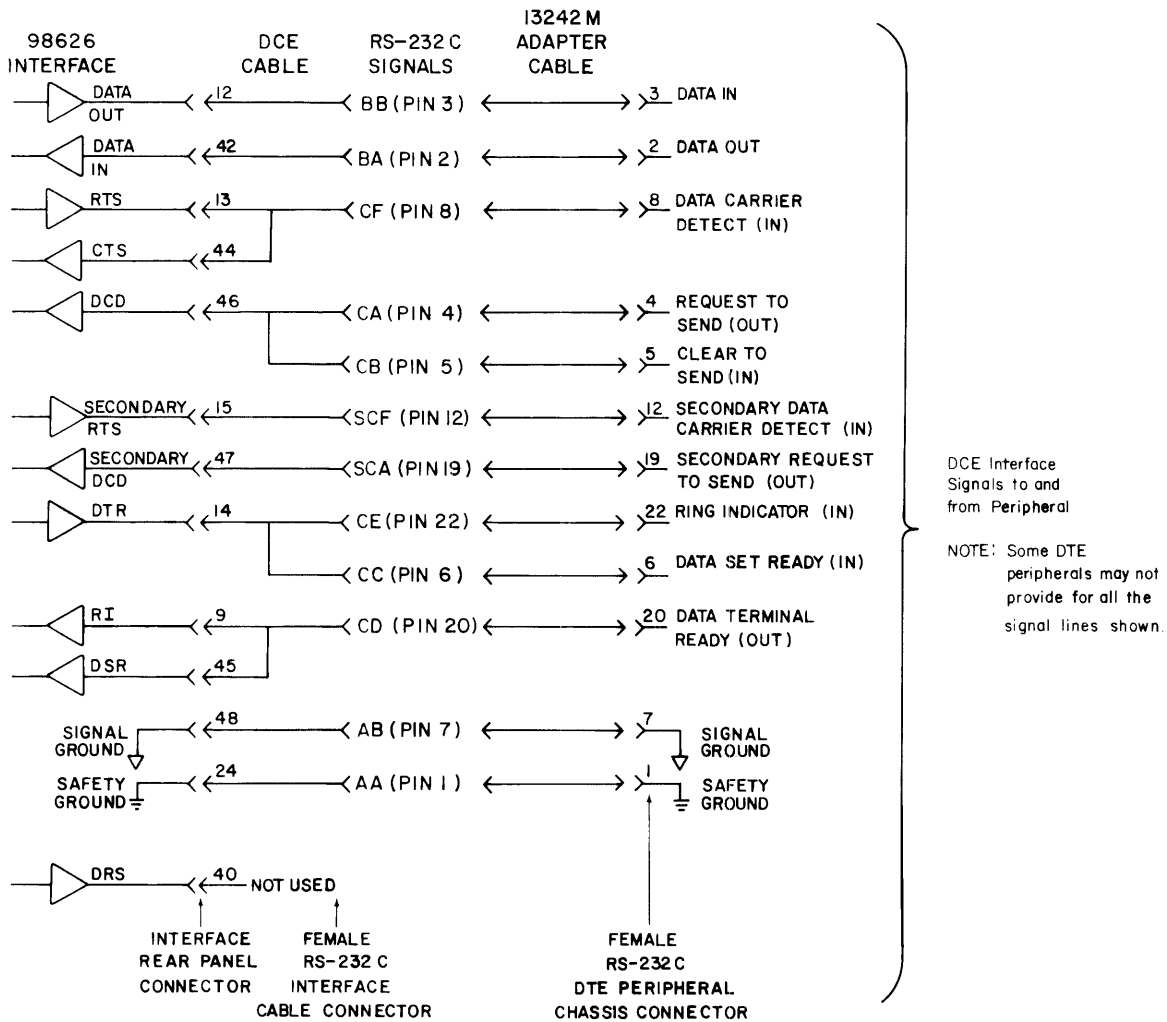
DCE Cable Interconnection Diagram

RS-232C/CCITT V24[1]

The following table provides information about each data communications interface function. The pin assignments are also shown. Not all of the functions provided by RS-232C are implemented. The functions denoted with an * **are implemented.**

<p align="center">**RS-232C/CCITT V24[1]**</p>

RS-232C	CCITT V24	Signal Name
*Pin 1	101	PROTECTIVE GROUND. Electrical equipment frame and ac power ground.
*Pin 2	103	TRANSMITTED DATA. Data originated by the terminal to be transmitted via the sending modem.
*Pin 3	104	RECEIVED DATA. Data from the receiving modem in response to analog signals transmitted from the sending modem.
*Pin 4	105	REQUEST TO SEND. Indicates to the sending modem that the terminal is ready to transmit data.
*Pin 5	106	CLEAR TO SEND. Indicates to the terminal that its modem is ready to transmit data.
*Pin 6	107	DATA SET READY. Indicates to the terminal that its modem is not in a test mode and that modem power is ON.
*Pin 7	102	SIGNAL GROUND. Establishes common reference between the modem and the terminal.
*Pin 8	109	DATA CARRIER DETECT. Indicates to the terminal that its modem is receiving carrier signals from the sending modem.
Pin 9		Reserved for test.
Pin 10		Reserved for test.
Pin 11		Unassigned.
*Pin 12	122	SECONDARY DATA CARRIER DETECT. Indicates to the terminal that its modem is receiving secondary carrier signals from the sending modem.
Pin 13	121	SECONDARY CLEAR TO SEND. Indicates to the terminal that its modem is ready to transmit signals via the secondary channel.

These signals are commonly used for 3 wire (no modem) links.

[1] International Telephone and Telegraph Consultative Committee European standard.

RS-232C/CCITT V24 (Cont'd)

RS-232C	CCITT V24	Signal Name
Pin 14	118	SECONDARY TRANSMITTED DATA. Data from the terminal to be transmitted by the sending modem's channel.
*Pin 15	114	TRANSMITTER SIGNAL ELEMENT TIMING. Signal from the modem to the transmitting terminal to provide signal element timing information.
Pin 16	119	SECONDARY RECEIVED DATA. Data from the modem's secondary channel in response to analog signals transmitted from the sending modem.
*Pin 17	115	RECEIVER SIGNAL ELEMENT TIMING. Signal to the receiving terminal to provide signal element timing information.
Pin 18		Unassigned.
*Pin 19	120	SECONDARY REQUEST TO SEND. Indicates to the modem that the sending terminal is ready to transmit data via the secondary channel.
*Pin 20	108.2	DATA TERMINAL READY. Indicates to the modem that the associated terminal is ready to receive and transmit data.
Pin 21	110	SIGNAL QUALITY DETECTOR. Signal from the modem telling whether a defined error rate in the received data has been exceeded.
*Pin 22	125	RING INDICATOR. Signal from the modem indicating that a ringing signal is being received over the line.
*Pin 23	111	DATA SIGNAL RATE SELECTOR. Selects one of two signaling rates in modems having two rates.
*Pin 24	113	TRANSMIT SIGNAL ELEMENT TIMING. Transmit clock provided by the terminal.
Pin 25		Unassigned.

Summary of RS-232 Serial Status and Control Registers

General Notes: Most Control registers accept values in the range of zero through 255. Some registers accept only specified values as indicated, or higher values for baud rate settings. Values less than zero are not accepted. Higher-order bits not needed by the interface are discarded if the specified value exceeds the valid range.

Reset value is the default value used by the interface after a reset or power-up until the value is overridden by a CONTROL statement.

Status 0 Card Identification
Value returned: 2 (if 130 is returned, the Remote jumper wire has been removed from the interface card).

Control 0 Interface Reset
Any value from 1 thru 255 resets the card. Execution is immediate; any data transfers in process are aborted and any buffered data is destroyed. A value of 0 causes no action.

Status 1 Interrupt Status
Bit 7 set: Interface hardware interrupt to CPU enabled.
Bit 6 set: Card is requesting interrupt service.
Bits 5&4:   00   Interrupt Level 3
             01   Interrupt Level 4
             10   Interrupt Level 5
             11   Interrupt Level 6
Bits 3 thru 0 not used.

Control 1 Transmit BREAK
Any non-zero value sends a 400 millisecond BREAK on the serial line.

Status 2 Interface Activity Status
Bit 7 thru 3 are not used.
Bit 2 set: Handshake in progress. This occurs only during multi-line function calls.
Bit 1 set: Firmware interrupts enabled (ENABLE INTR active for this select code).
Bit 0:   Reserved for future use.

Status 3 Current Baud Rate
Returns one of the values listed under Control Register 3.

Control 3 Set New Baud Rate
Use any one of the following values:

50	150	1200	3600
75	200	1800	4800
110	300	2000	7200
134.5 (or 134)	600	2400	9600

Status 4 Current Character Format

See Control Register 4 for function of individual bits.

Control 4 Set New Character Format

Parity Sense[1] (Bits 5&4)	Parity Enable (Bit 3)	Stop Bits (Bit 2)	Character Length (Bits 1&0)
00 ODD parity 01 EVEN parity 10 Always ONE 11 Always ZERO	0 = Parity OFF 1 = Parity ON	0 = 1 stop bit 1 = 2 stop bits[2]	00 5 bits 01 6 bits 10 7 bits 11 8 bits

Bits 7 and 6 are reserved for future use.

Status 5 Current Status of Modem Control Lines

Returns CURRENT line state values. See Control Register 5 for function of each bit.

Control 5 Set Modem Control Line States

Sets Modem Control lines or interface state as follows:

Bit 4 set: Enables loopback mode for diagnostic tests.

Bit 3 set: Set Data Rate Select modem line to active state.

Bit 2 set: Set Secondary Request-to-Send modem line to active state.

Bit 1 set: Force Request-to-Send modem line to fixed active state.

Bit 1 clear: Toggle RTS line as in normal OUTPUT operations.

Bit 0 set: Force Data Terminal Ready modem line to fixed active state.

Bit 0 clear: Toggle DTR line as in normal OUTPUT and ENTER operations.

Status 6 Data In

Reads character from input buffer. Buffer contents is not destroyed. but bit 0 of Status Register 10 is cleared.

Control 6 Data Out

Sends character to transmitter holding register. This register is sometimes used to transmit protocol control characters or other characters without using OUTPUT statements. Modem control lines are not affected.

Status 7 Optional Receiver/Driver Status

Returns current value of optional circuit drivers or receivers as follows:

Bit 3: Optional Circuit Driver 3 (OCD3).

Bit 2: Optional Circuit Driver 4 (OCD4).

Bit 1: Optional Circuit Receiver 2 (OCR2).

Bit 0: Optional Circuit Receiver 3 (OCR3).

Other bits are not used (always 0).

Control 7 Set New Optional Driver States

Sets (bit = 1) or clears (bit = 0) optional circuit drivers as follows:

Bit 3: Optional Circuit Driver 3 (OCD3).

Bit 2: Optional Circuit Driver 2 (OCD2).

Other bits are not used.

---

[1] Parity sense valid only if parity is enabled (bit 3 = 1). If parity is disabled, parity sense is meaningless.

[2] If character length is 5 bits, bit 2 = 1 sends 1.5 stop bits with each character frame.

Status 8 Current Interrupt Enable Mask

Returns value of interrupt mask associated with most recent ENABLE INTR statement. Bit functions are as follows:

Bit 3: Enable interrupt on modem line change. Status Register 11 shows which modem line has changed.

Bit 2: Enable interrupt on UART status error. This bit is used to trp ERROR 167 caused by UART error conditions. Status Register 10, bits 4 thru 1 show cause of error.

Bit 1: Enable interrupt when Transmitter Holding Register is empty.

Bit 0: Enable interrupt when Receiver Buffer is full.

Status 9: Cause of Current Interrupt

Returns cause of interrupt as follows:

Bits 2&1: Return cause of interrupt

11 = UART error (BREAK, parity, framing, or overrun error). See Status Register 10.

10 = Receiver Buffer full. Cleared by STATUS to Register 6.

01 = Transmitter Holding Register empty. Cleared by CONTROL to Register 6 or STATUS to Register 9.

00 = Interrupt caused by change in modem status line(s). See Status Register 11.

Bit 0: Set when no active interrupt requests from UART are pending. Clear until all pending interrupts have been serviced.

Status 10 UART Status

Bit set indicates UART status or detected error as follows:

Bit 7: Not used.

Bit 6: Transmit Shift Register empty.

Bit 5: Transmit Holding Register empty.

Bit 4: Break received.

Bit 3: Framing error detected.

Bit 2: Parity error detected.

Bit 1: Receive Buffer Overrun error.

Bit 0: Receiver Buffer full.

Status 11 Modem Status

Bit set indicates that the specified modem line or condition is active.

Bit 7: Data Carrier Detect (DCD) modem line active.

Bit 6: Ring Indicator (RI) modem line active.

Bit 5: Data Set Ready (DSR) modem line active.

Bit 4: Clear-to-Send (CTS) modem line active.

Bit 3: Change in DCD line state detected.

Bit 2: RI modem line changed from true to false.

Bit 1: Change in DSR line state detected.

Bit 0: Change in CTS line state detected.

Powerfail Protection	Chapter
	15

Some Series 200 computers have an optional capability of up to about one minute of powerfail protection. This chapter describes the capabilities provided by this optional internal interface, which has been **permanently assigned to interface select code 5**.

This optional feature is discussed in this Interfacing Techniques manual because of the nature of its access from BASIC programs. If you need additional explanation regarding interface registers or interface interrupt events, refer to Chapters 6 and 7 of this manual, respectively.

Note

The powerfail option is not available for all Series 200 computers. Also, not all Series 200 computers use the same monitors. The actual parameters you need may differ from those shown in some of the example programs. See the documentation for your hardware system for details.

Overview of Capabilities Provided

The powerfail protection provided by the internal battery-backup circuitry is as follows.

- A period of operation **after powerfail** may be specified. (See the hardware documentation for Option 050 for the Model 226 or 236 computers).

- The interface may optionally interrupt the computer when a powerfail has occurred. A delay time before interrupt may also be programmed to allow the computer to ignore power "glitches".

- The program can read both the powerfail interrupt cause and determine current powerfail status information, including ac power status, battery time remaining, and time elapsed since power was returned.

- The real-time clock and 64 bytes of memory registers are maintained after power has been down for greater than one minute.

The Computer's Reaction to Powerfails

There are two general categories of computer reactions to powerfail situations. The default response is to continue running as before the failure for up to one minute. The alternate response is to interrupt the current routine's execution to service the failure. In either case, the computer beeps and the following warning message is displayed on the CRT when the power-fail is detected.

```
Power failed
```

If power remains off for more than one minute, or if the computer turns itself off, only a real-time clock and 64 bytes of low-power memory registers are maintained. If power is restored, the computer powers on in its normal powerup sequence.

Continuous-Memory Registers

The sixty-four, single-byte registers on the interface are maintained after power has failed. The contents of these registers can be written with CONTROL statements and read with STATUS statements. The registers are numbered 8 through 71.

Real-Time Clock

The clock on the powerfail interface is read at powerup and is used to set the BASIC system clock. However, the system clock, not the powerfail clock, is read by the T I ME DA TE function.

Executing either SET TIME or SET TIMEDATE sets **both** clocks to the specified value. Thus, the two clocks may drift apart temporarily but may be synchronized by setting time with either of these statements. See Chapter 9 of *BASIC Programming Techniques* for further detail.

Powerfail-Protection Timers

Three additional timers are used by the interface to keep track of times between different powerfail events. These timers allow the program to keep track of Powerfail events so that the desired service response may be initiated.

When a powerfail occurs, the **Powerfail Timer** is cleared and begins to count the seconds elapsed since the powerfail occurred. After waiting the **Powerfail Delay Time**, the interface may generate a Powerfail interrupt, if enabled to do so. If and when the Powerfail Timer timer reaches the value of the **Protection Time**, the computer automatically powers down.

When power is returned, the **Power Back Timer** is cleared and begins counting seconds elapsed since the power back occurred. When this timer reaches the value of the **Power Back Delay**, the computer is no longer in the Powerfail State; a Power Back interrupt is generated, if enabled.

When a powerfail occurs, the **Overheat Protection Timer** begins to increment, counting the seconds elapsed since the powerfail event occurred. When power is restored, this timer is **decremented one second for every two seconds that power is back**. If power remains or long enough, the timer decrements to 0. However, if the timer reaches 60 seconds, the computer automatically powers down. These actions ensure that the fan adequately cools the computer during continuous power fluctuations.

Further description of delay times, timer actions, and enabling interrupt events are described in the remainder of this chapter.

Interrupt Events

Interrupts can be generated by the powerfail-protection controller when **three different events are sensed**: when power fails, when power is returned, and when approximately one second of battery power remains. Enabling these events to initiate interrupts and typical responses to these events are explained in this and in the following section.

Setting Up and Enabling Interrupts

The desired interrupt condition(s) may be enabled by specifying the appropriate numeric mask value. The bits of the Interrupt Enable register enable the following interrupts.

Powerfail Interrupt Enable 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		One Second Left	Power Is Back	Power Has Failed
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

One Second Left — When this bit is set (1), an interrupt to the computer is generated when approximately one second of battery power remains.

Power Is Back — When this bit is set (1), an interrupt to the computer is generated when power has been returned (after a previous powerfail).

Power Has Failed — When this bit is set (1), an interrupt to the computer is generated when a powerfail has been detected.

The branch to the powerfail service routine is set up and enabled in the same manner as are other interrupt service routines. A typical example is as follows.

```
200    ON INTR 5 GOSUB Power_down
210    Mask=1   ! Enable Powerfail Interrupt.
210    ENABLE INTR 5;Mask
```

Service Routines

The service routine must determine which type of event initiated the interrupt branch. The bits of the Interrupt Cause register have the same definitions as those of the Interrupt Enable Mask register.

STATUS Register 1   **Powerfail Interrupt Cause**
Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
			Not Used		One Second Left	Power Is Back	Power Has Failed
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

```
100     STATUS 5,1;Interrupt_cause
```

If more than one interrupt cause has occurred, more than one bit will be set in this register. Also, the register's contents must be stored in a variable which is not used until all causes have been determined, because **reading this register clears its contents**.

Also keep in mind that when the "One Second Left" bit is a 1, the computer will power down **regardless** of whether on not power is back before the end of the one second.

The action performed by the service routine is usually to store critical data. The internal disc drives remain fully operational for this purpose. External drives usually lose power when the computer loses its power; if so, they should not be used for this purpose. Other external devices may also be affected by the failure and therefore may not respond to the request to transfer the data. Therefore, all attempts to communicate with external devices should have ON TIMEOUT branches set up and enabled so that the program will not spend the entire minute waiting for the device to respond.

Powerfail Status and Timers

The Powerfail Status register and Timer registers provide useful information describing the state of computer power. The following example service routine reads these STATUS registers and displays the information on the CRT.

```
100     ON INTR 5 GOSUB Pfail_service
110     ENABLE INTR 5;7   ! Enable all three causes,
120     !
130     Pback_delay=300   ! Delay 3 s before Pback interrupt,
140     Protection=2000   ! 20 s max, of Pfail protection,
150     Pfail_delay=100   ! Delay 1 s before Pfail interrupt,
160     CONTROL 5,5;Pback_delay,Protection,Pfail_delay
170     !
180     LOOP
190        CONTROL 1;1,1   ! Upper-left corner,
200        OUTPUT 1;Number
210        Number=Number+1
220     END LOOP
```

```
230    !
240 Pfail_service: CONTROL 1;1,3  ! Begin on third line,
250                  OUTPUT 1;"  Powerfail Interface Registe
rs"
260                  OUTPUT 1;" ----------------------------
---"
270                    !
280    REPEAT
290      CONTROL 1;1,5  ! Begin printing on line 5,
300      STATUS 5,3;Pf_status
310        Pfail=BIT(Pf_status,0)
320        Ac_down=BIT(Pf_status,1)
330        Batt_on=BIT(Pf_status,2)
340        One_sec=BIT(Pf_status,3)
350        S_test=BIT(Pf_status,7)
360        OUTPUT 1
370        OUTPUT 1;"STATUS Register 3 - Powerfail Status:"
380        OUTPUT 1;" Test Fail   1 Sec,     Batt, On     Ac
Down    In Pfail"
390        OUTPUT 1 USING "#,5X,D,5X";S_tst,One_sec,Batt_on
,Ac_down,Pfail
400        OUTPUT 1 USING "/"
410        !
420      STATUS 5,4;Ovheat
430        OUTPUT 1;"STATUS Register 4 - Overheat Timer:   "
;
440        OUTPUT 1 USING "DD.D,/";Ovheat/100
450        !
460      STATUS 5,5;Pback
470        OUTPUT 1;"STATUS Register 5 - Power Back Timer:"
;
480        OUTPUT 1 USING "DD.D,/";Pback/100
490        !
500      STATUS 5,6;Pf_timer
510        OUTPUT 1;"STATUS Register 6 - Powerfail Timer:  "
;
520        OUTPUT 1 USING "DD.D,/";Pf_timer/100
530        !
540      STATUS 5,4;Ov_heat
550    UNTIL Ov_heat=0     ! UNTIL Overheat timer expires,
560    !
570    ENABLE INTR 5              ! Use same mask,
580    RETURN
590    !
600    END
```

Type in and run the program. Alternately remove and replace the power cord while watching the status values and timers change. You are highly encouraged to experiment with the parameters until you are familiar with how the computer responds to power failures. The next section presents several simple examples of service routines.

Typical Service Routines

The Powerfail Protection option allows programming several types of service responses. A few typical examples are shown in this section. All STATUS and CONTROL registers are summarized at the end of the chapter.

Using the Continuous-Memory Registers

The most common function of service routines is to store any critical data and then turn the computer off to conserve battery power. The following simple example shows the use of the continuous-memory registers for storing a message.

```
100    ON INTR 5 GOTO Pfail_serve
110    ENABLE INTR 5;1    ! Pfail interrupts only.
120    !
130    ! Use defaults of: 500 ms    Pback Delay,
140    !                   60  s     Protection Time,
150    !                  100 ms     Pfail Delay.
160    !
170    LOOP
180       DISP Number
190       Number=Number+1
200    END LOOP
210    !
220    STOP
230    !
240 Pfail_serve:  ! Write message in Cont-Mem. Registers.
250               !
260       Message$="Adios, amigos."
270       Message$=Message$&CHR$(10) ! Add LF.
280       No_bytes=LEN(Message$)
290       !
300       FOR Reg=8 TO 8+No_bytes-1
310          CONTROL 5,Reg;NUM(Message$[Reg-7;1])
320       NEXT Reg
330       !
340       CONTROL 5;1 ! Shut down when finished.
350       !
360    END
```

Type in the program and press (RUN). The CRT shows a counter running continuously. Unplugging the power cord initiates the Powerfail interrupt after the default delay of 100 milliseconds. Thus, if power had failed for a duration of less than 100 milliseconds, the interrupt would not have been generated. Similarly, the Power Back Delay determines how long the computer will delay after power has been restored before generating a Power Back interrupt, when enabled.

The program did not allow the Powerfail Timer to reach the default Protection Time (60 seconds). Instead, it powered itself down after storing a message in the registers in order to save battery power. If power is subsequently restored, the computer powers on in the normal powerup sequence. If an Autostart routine exists, it will be run automatically.

The following program shows a method for reading the message stored in the continuous-memory registers by the preceding program. The program makes use of the fact that the message was terminated by a line-feed character, CHR$(10).

```
100     PRINTER IS 1
110     !
120     ! Read message in Continuous-Memory Registers.
130     DIM Registers$[64],Message$[64]
140     !
150     FOR Register=8 TO 71       ! Read all 64 registers.
160        STATUS 5,Register;Byte
170        Registers$[Register-7]=CHR$(Byte)
180     NEXT Register
190     !
200     ENTER Registers$;Message$ ! Enter and stop at LF.
210     !
220     PRINT Message$
230        !
240     END
```

Storing Data on Disc

Service routines can be programmed to take many other actions, such as to store data on an internal disc. The following program shows a technique for storing the ALPHA and GRAPHICS displays and the value of the clock at the time the powerfail occurred. This program requires GRAPH BIN file.

```
100     INTEGER Crt_graphics(1:12480) ! (1:7500) for Model 226.
110     DIM Crt_alpha$(1:57)[80]        ! [50] for Model 226.
120     !
130     ON INTR 5 GOTO Pfail_serve
140     ENABLE INTR 5;1   ! Pfail interrupts only.
150     !
160     Pback_delay=100   ! Delay 1 s before Pback interrupts
        .
170     Protection=3000   ! 30 s max. of Protection Time.
180     Pfail_delay=200   ! Delay 2 s before Pfail interrupts
        .
190     CONTROL 5,5;Pback_delay,Protection,Pfail_delay
200     !
210     FOR Crt_line=1 TO 57
220        OUTPUT 1;"Output Area line";Crt_line
230     NEXT Crt_line
240     !
250     GCLEAR
260     GRAPHICS ON
270     FRAME
280     MOVE 50,50
290     LABEL "GRAPHICS DISPLAY"
300     !
310     LOOP
320        DISP Number
330        Number=Number+1
340     END LOOP
```

```
350    !
360    STOP
370    !
380 Pfail_serve: ! First, store GRAPHICS display.
390      GSTORE Crt_graphics(*)
400      !
410      ! Then store ALPHA display.
420      STATUS 1,3;Lines_above
430      CONTROL 1;1,-Lines_above+1 ! Move print position
440                                 ! to "top" of display.
450      ENTER 1 USING "K";Crt_alpha$(*) ! Enter screen.
460      !
470      ON ERROR GOTO Already
480         CREATE BDAT "Pfail_data:INTERNAL,4,1",116
490 Already: OFF ERROR ! File already created.
500      ASSIGN @File TO "Pfail_data:INTERNAL,4,1"
510      OUTPUT @File;Crt_graphics(*),Crt_alpha$(*)
520      !
530      CONTROL 5;1 ! Shut down when finished.
540    END
```

The INTEGER array used to store the graphics display was dimensioned for the Model 236's display (12 480 INTEGER elements). Exactly 7 500 INTEGER elements are required to store the Model 226's graphics display.

The size of the BDAT file was chosen for the "worst case" storage requirement. In order to calculate the maximum number of of disc sectors required to store both displays, you must determine three facts: the maximum number of data elements to be stored, the data type of each item, and the number of bytes required to store one element of each data type.

The Model 236 display's Output-Area memory can hold up to 57 lines of 80 characters each (4 560 bytes). The graphics display requires 12 480 INTEGERs (24 960 bytes). A total of 29 520 bytes of storage is required. Since BDAT files contain default records of 256 bytes each, the file "Pfail_data" was dimensioned to 116 256-byte records.

The following program gives a method of restoring the alpha and graphics displays and real-time clock. Actual programs would probably also restore other variables and resume program execution that was interrupted by the powerfail. This program requires GRAPH BIN file.

```
100    ! This program for use on a Model 236; change
110    ! array sizes and msus for use on a Model 226.
120    !
130    INTEGER Graphics(1:12480) ! (1:7500) for Model 226.
140    DIM Crt_alpha$(1:57)[80]  ! [50] for Model 226.
150    !
160    ASSIGN @File TO "Pfail_data:INTERNAL,4,1"
170    !                    ":INTERNAL,4,0" for Model 226.
180    ENTER @File;Graphics(*)
190    ENTER @File;Crt_alpha$(*)
200    ENTER @File;Clock
210    !
```

```
220    GRAPHICS ON
230    GLOAD Graphics(*)
240    !
250    OUTPUT 1;Crt_alpha$(*)
260    !
270    SET TIME Clock
280    DISP "Powerfail occurred at";Clock
290    !
300    !
310    END
```

A very important consideration for the powerfail service routine is that it has enough battery time to store all the specified data. If there is insufficient battery time to allow storing all desired data, the service routine should be able to record exactly how far it got into the backup when battery power went down. The next example shows how to enable interrupts to signal that power is back or that only one second of battery power is left.

Power-Is-Back and One-Second-Left Interrupts

The powerfail-interface controller has the ability to sense when power is back and when approximately one second of battery power remains; it can optionally generate interrupts to the BASIC program when these events occur. The following example program shows how to enable and service these types of interrupts.

```
100    COM Important_data$(1:8192)[28]
110    DIM Random$[28]
120    !
130    ON INTR 5,14 CALL Pfail_response
140    ENABLE INTR 5;1 ! "Power Has Failed" interrupts.
150    !
160    !
170    REPEAT


                        Main Program.



370    UNTIL Error<1.E-12
380    !
390    END
400    !
```

```
410   ! ******** Powerfail Service Routine ********
420   SUB Pfail_response
430   COM Important_data$(1:8192)[28]
440   DIM Message$[64]
450   !
460   ! Set up and enable service routine for
470   ! "One-Sec-Left" and "Power-Back" interrupts;
480   ! priority 15 allows data storage to be interrupted.
490   ON INTR 5,15 GOSUB Stop_storing
500   ENABLE INTR 5;4+2
510   !
520   ! Assume BDAT file (1024 records) exists.
530   ASSIGN @Storage TO "PFAIL_DATA"
540   ! Store elements individually to permit interrupts.
550   FOR Element=1 TO 8192
560     OUTPUT @Storage;Important_data$(Element)
570   NEXT Element
580   !
590   ! Power Down after all data stored.
600   CONTROL 5;1
610   !
620   ! ********* New service routine. *********
630 Stop_storing: STATUS 5,1;Intr_cause
640                    !
650   IF BIT(Intr_cause,2) THEN   ! One Second Left.
660     ! Define Message.
670     Message$="Only the first "&VAL$(Element)
680     Message$=Message$&" elements have been stored."
690     Message$=Message$&" Error="&VAL$(Error)
700     Message$=Message$&CHR$(10) ! End with LF.
710     ! Write to Continuous-Memory Regs.
720     FOR Reg=8 TO LEN(Message$)+7
730       CONTROL 5,Reg;NUM(Message$[Reg-7;1])
740     NEXT Reg
750     ! Power Down.
760     CONTROL 5;1
770   END IF
780   !
790   IF BIT(Intr_cause,1) THEN   ! Power Is Back.
800     ! Re-enable "Power Has Failed" interrupts.
810     ENABLE INTR 5;1
820     ! Then return to interrupted context.
830     SUBEXIT
840   END IF
850   !
860   SUBEND ! ********************************
```

The service routine first enables two types of interrupts; one is generated when power is back after the powerfail, and the other is generated when approximately one second of battery power remains. Then, the service routine attempts to store the specified data. Notice that the service routine stores the data one item at a time so that either interrupt may be serviced while the data are being stored.

If the Power-Is-Back interrupt is generated, the service routine ends and returns to the main program. You may want to expand the service routine to sense recurring power flutuations and to respond accordingly. If the One-Second-Left interrupt is generated, the program stores a message to show how much of the desired data have been stored. Keep in mind that once this interrupt is generated, the computer powers down, **regardless** of whether power is restored before the end of the one second.

Summary of Powerfail
Status and Control Registers

This section lists all STATUS and CONTROL registers of the Powerfail-Protection Interface, which is permanently assigned to interface select code 5.

STATUS Register 0 — Card Identification is always 5.

CONTROL Register 0 — Shut Down. Any non-zero value written to this register will turn off both battery and ac-line power to the computer, which conserves battery power after the service routine has finished responding to the powerfail. If ac-line power is on when this statement is executed, the computer will be turned back on in the normal powerup sequence.

STATUS Register 1   **Powerfail Interrupt Cause**

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not Used					One Second Left	Power Is Back	Power Has Failed
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

CONTROL Register 1 — Not Used.

STATUS Register 2 — Interrupt Mask has bit definitions identical to the preceding register (Powerfail Interrupt Cause).

CONTROL Register 2 — Not Used.

STATUS Register 3   **Powerfail Status**

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Failed Self Test	Not Used			One Second Left	Currently Using Battery	Ac Is Down	In the Powerfail State
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Bit 7 — Failed Self Test indicates the outcome of the self test: a 1 indicates failure, and 0 indicates successful results.

Bit 3 — One Second Left indicates that approximately one second of battery power remains. The computer will automatically power itself down, even if power is restored before one second has expired.

Bit 2 — Currently Using Battery indicates whether or not the battery is being used: 1 indicates it is currently being used for computer power, and 0 indicates that it is not.

Bit 1 — Ac Is Down indicates the current status of ac-line power: a 1 indicates that ac power is completely gone. If bit 2 is a 1 and this bit is 0, the battery is being used because ac power is not completely gone but has dropped below an acceptable level; in this case, a "brown-out" condition is indicated.

Bit 0 — In the Powerfail State indicates whether or not the computer is currently in the Powerfail State: a 1 indicates Powerfail State, and 0 indicates that the computer is not currently in the Powerfail State. The Powerfail State is exited when power is back and the Power Back Timer reaches the value of the Power Back Delay.

CONTROL Register 3 — Not Used.

STATUS Register 4 — Overheat Protection Timer contains the amount of battery time used during this Powerfail State (in tens of milliseconds). For every second the power is down, it must be back for two seconds to ensure adequate cooling for the machine. Thus, the value of this register bounds the maximum amount of time that can be obtained from the battery, even though 60 seconds may have been specified as the protection time (CONTROL Register 6).

CONTROL Register 4 — Not Used.

STATUS Register 5 — Power Back Timer contains the time elapsed since power was restored after the last powerfail (in tens of milliseconds).

CONTROL Register 5 — Power Back Delay. The value of this register determines the amount of time (in tens of milliseconds) that the computer will delay, after power is back, before leaving the powerfail state (i.e., before generating a "Power Is Back" interrupt). The power-on default value is 50 (500 milliseconds).

STATUS Register 6 — Powerfail Timer contains the time elapsed since the last powerfail (in tens of milliseconds).

CONTROL Register 6 — Protection Time. The value of register determines the maximum amount of time (in tens of milliseconds) that the computer is to have battery backup. Power-on default is 6000 (60 seconds).

STATUS Register 7 — Not Used.

CONTROL Register 7 — Powerfail Delay Timer. The contents of this register determine the amount of time (in tens of milliseconds) that the Powerfail-Protection Interface will wait, after a powerfail, before generating a "Power Has Failed" interrupt. Power-on default is 10 (100 milliseconds).

STATUS Registers 8 thru 71 — Continuous-Memory Registers contain the 64 bytes of data written by the last CONTROL statement directed to these registers.

CONTROL Registers 8 thru 71 — Continuous-Memory Registers. These sixty-four, single-byte registers can be filled with any desired data, one byte (ASCII character) per register.

The GPIO Interface	Chapter 16

Introduction

This chapter should be used in conjunction with the *HP 98622A GPIO Interface Installation* manual. **The best way to use these two documents is to read this chapter before attempting to configure and connect the interface** according to the directions given in the installation manual. The reason for this order of use is that knowing how the interface works and how it is driven by BASIC programs will help you to decide how to connect it to your peripheral device. If you only need to use the interface to connect the computer to simple devices (such as printers), the examples at the end of the chapter (to be added) may be enough to "get you up and running".

The HP 98622 Interface is a very flexible parallel interface that allows you to communicate with a variety of devices. The interface sends and receives up to 16 bits of data with a choice of several handshake methods. External interrupt and user-definable signal lines are provided for additional flexibility. The interface is known as the General-Purpose Input/Output (GPIO) Interface for these reasons. This chapter describes the use of the interface's features from BASIC programs.

Use of some statements or suggestions for interfacing requires that you load the TRANS BIN file.

Interface Description

The main function of any interface obviously to transfer data between the computer and a peripheral device. This section briefly describes the interface lines and how they function. Using the lines from BASIC programs is more fully described in subsequent sections.

The GPIO Interface provides **32 lines for data input and output**: 16 for input (DI0 — DI15), and 16 for output (DO0 — DO15).
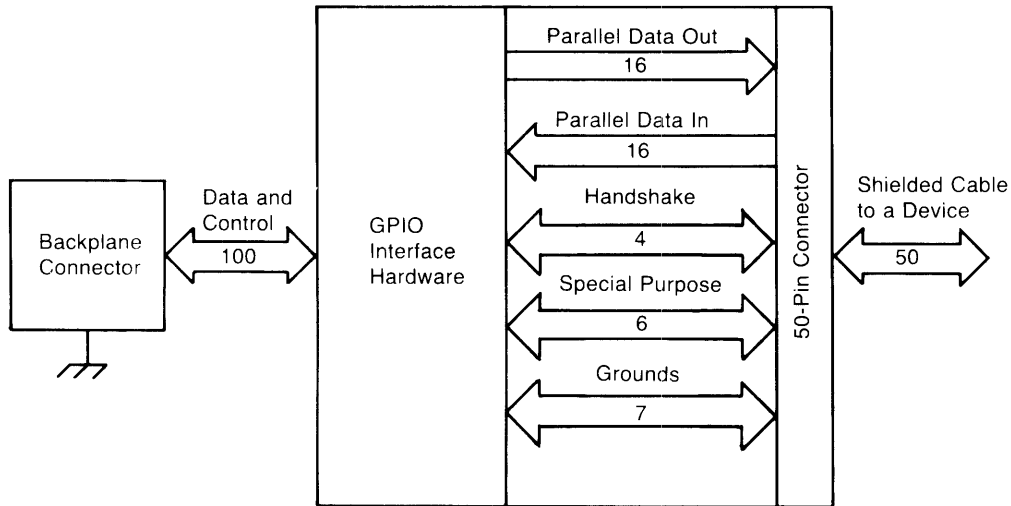
Block Diagram of the GPIO Interface

Three lines are dedicated to **handshaking** the data from source to destination device. The Peripheral Control line (PCTL) is controlled by the interface and is used to initiate data transfers. The Peripheral Flag line (PFLG) is controlled by the peripheral device and is used to signal the peripheral's readiness to continue the transfer process. The Input/Output line (I/O) is used to indicate direction of data flow.

One line is used to signal External Interrupt Requests to the computer (EIR). The interface must be enabled to initiate interrupt branches for the interface to detect this request. The state of the line can also be read by the program.

Four general-purpose lines are available for any purpose that you may desire: two are controlled by the computer and sensed by the peripheral (CTL0 and CTL1), and two are controlled by the peripheral device and sensed by the computer (STI0 and STI1).

Both Logic Ground and Safety Ground are provided by the interface. Logic Ground provides the reference point for signals, and Safety Ground provides earth ground for cable shields.

Interface Configuration

This section presents a brief summary of selecting the interface's configuration-switch settings. It is intended to be used as a checklist and to begin to acquaint you with programming the interface. **Refer to the installation manual for the exact location and setting of each switch.**

The following sample program checks a few of these switch settings on a GPIO Interface already installed in the computer and displays the settings. However, many of the settings cannot be determined from BASIC programs. If any of the displayed settings are different than desired, or if any settings are not already known, refer to the installation manual for switch locations and settings.

```
100    PRINTER IS 1    ! Select printer device,
110    PRINT CHR$(12) ! Clear screen,
120    !
130    DISP "Enter GPIO Interface Select Code (CONT=12)"
140    OUTPUT 2 USING "#,DD";12
150    ENTER 2;Isc
160    DISP
170    !
180    ASSIGN @Gpio TO Isc ! FORMAT ON default,
190    !
200    ! Read STATUS registers 0 and 1,
210    STATUS Isc;Card_id,Intr_stat
220    !
230    ! Is this card a GPIO?
240    IF Card_id<>3 THEN
250       PRINT "The interface at select code";Isc
260       PRINT "is not a GPIO Interface,"
270       PRINT "Program stopped,"
280       STOP
290    ELSE
300       PRINT "The card ID of the GPIO at"
310       PRINT "interface select code";Isc
320       PRINT "is";Card_id
330    END IF
340    PRINT
350    !
360    ! Calculate hardware interrupt priority,
370    Bits_5_and_4=BINAND(Intr_stat,32+16)
380    Switches=Bits_5_and_4 DIV 16
390    Hd_prior=Switches+3
400    PRINT "Hardware Interrupt Priority is";Hd_prior
410    PRINT
420    !
430    END
```

Interface Select Code

In BASIC, allowable interface select codes range from 8 through 31; codes 1 through 7 are already used for built-in interfaces. The GPIO interface has a factory default setting of 12, which can be changed by re-configuring the "SEL CODE" switches on the interface.

Hardware Interrupt Priority

Two switches are provided on the interface to allow selection of hardware interrupt priority. The switches allow hardware priority levels 3 through 6 to be selected. **Hardware priority** determines the order in which simultaneously occurring interrupt events are **logged**, while **software priority** determines the order in which interrupt events are **serviced** by the BASIC program[1].

Data Logic Sense

The data lines of the interface are **normally low-true**; in other words, when the voltage of a data line is low, the corresponding data bit is interpreted to be a 1. This logic sense may be changed to high-true with the Option Select Switch. Setting the switch labeled "DIN" to the "0" position selects high-true logic sense of Data In lines. Conversely, setting the switch labeled "DOUT" to the "1" position inverts the logic sense of the Data Out lines. The default setting is "1" for both.

Data Handshake Methods

This section describes the data handshake methods available with the GPIO Interface. A general description of the handshake modes and clock sources is given first. A more detailed discussion of each handshake is then given to allow you to choose the handshake mode, clock source, and handshake-line logic sense that is compatible with your peripheral device.

As a brief review, a data handshake is a method of synchronizing the transfer of data from the sending to the receiving device. In order to use any handshake method, **the computer and peripheral device must be in agreement as to how and when several events will occur.** With the GPIO Interface, the following events must take place to synchronize data transfers; the first two are optional.

- The computer may optionally be directed to perform a one-time "OK check" of the peripheral before beginning to transfer any data.
- The computer may also optionally check the peripheral to determine whether or not the peripheral is "ready" to transfer data.
- The computer must indicate the direction of transfer and then initiate the transfer.
- During OUTPUT operations, the peripheral must read the data sent from the computer while valid; similarly, the computer must clock the peripheral's data into the interface's Data In registers while valid during ENTER operations.
- The peripheral must acknowledge that it has received the data.

Handshake Lines
The GPIO handshakes data with three signal lines. The Input/Output line, **I/O**, is driven by the computer and is used to signal the direction of data transfer. The Peripheral Control line, **PCTL**, is also driven by the computer and is used to initiate all data transfers. The Peripheral Flag line, **PFLG**, is driven by the peripheral and is used to acknowledge the computer's requests to transfer data.

Handshake Logic Sense
Logic senses of the PCTL and PFLG lines are selected with switches of the same name. The logic sense of the I/O line is High for ENTER operations and Low for OUTPUT operations; this logic sense cannot be changed. The available choices of handshake logic sense and handshake modes allow nearly all types of peripheral handshakes to be accommodated by the GPIO Interface.

Handshake Modes
There are two general handshake modes in which the PCTL and PFLG lines may be used to synchronize data transfers: Full-Mode and Pulse-Mode Handshakes. If the peripheral uses pulses to handshake data transfers **and** meets certain hardware timing requirements, the Pulse-Mode Handshake may be used. The Full-Mode Handshake should be used if the peripheral does not meet the Pulse-Mode timing requirements.

The handshake mode is selected by the position of the "HSHK" switch on the interface, as described in the installation manual. Both modes are more fully described in subsequent sections.

Data-In Clock Source
Ensuring that the data are valid when read by the receiving device is slightly different for OUTPUT and ENTER operations. During OUTPUTs, the interface generally holds data valid while PCTL is in the Set state, so the peripheral must read the data during this period. During ENTERs, the data must be held valid by the peripheral until the peripheral signals that the data are valid (which clocks the data into interface Data In registers) or until the data is read by the computer. The point at which the data are valid is signalled by a transition of PFLG. The PFLG transition that is used to signal valid data is selected by the "CLK" switches on the interface. Subsequent diagrams and text further explain the choices.

Optional Peripheral Status Check
Many peripheral devices are equipped with a line which is used to indicate the device's current "OK-or-Not-OK" status. If this line is connected to the Peripheral Status line (PSTS) of the GPIO Interface, and the computer may determine the status of the peripheral device by checking the state of PSTS. The logic sense of this line may be selected by setting the "PSTS" switch.

If enabled, the computer performs a **one-time check** of the Peripheral Status line (PSTS) **before initiating any transfers** as part of the data-transfer handshake. If PSTS indicates "Not OK," Error 172 is reported; otherwise, the transfer proceeds normally. If this feature is not enabled, this one-time check is never made. This feature is available with both Full-Mode and Pulse-Mode Handshakes. See "Using the PSTS Line" for further details.

Full-Mode Handshakes
The Full-Mode Handshake mode is described first for two reasons. The first reason is that the PCTL and PFLG transitions must always occur in the order shown, so only one sequence of peripheral handshake responses needs to be shown. Secondly, this mode will generally work when the Pulse-Mode Handshake may not be compatible with the peripheral's handshake signals. The Pulse-Mode Handshake is described in the next section.

The following diagrams show the order of events of the Full-Mode OUTPUT and ENTER Handshakes. These drawings are not drawn to any time scale: only the order of events is important. The I O line has been omitted to simplify the diagrams; in all cases, it is driven Low before any OUTPUT is initiated by the computer and High before any ENTER is initiated.
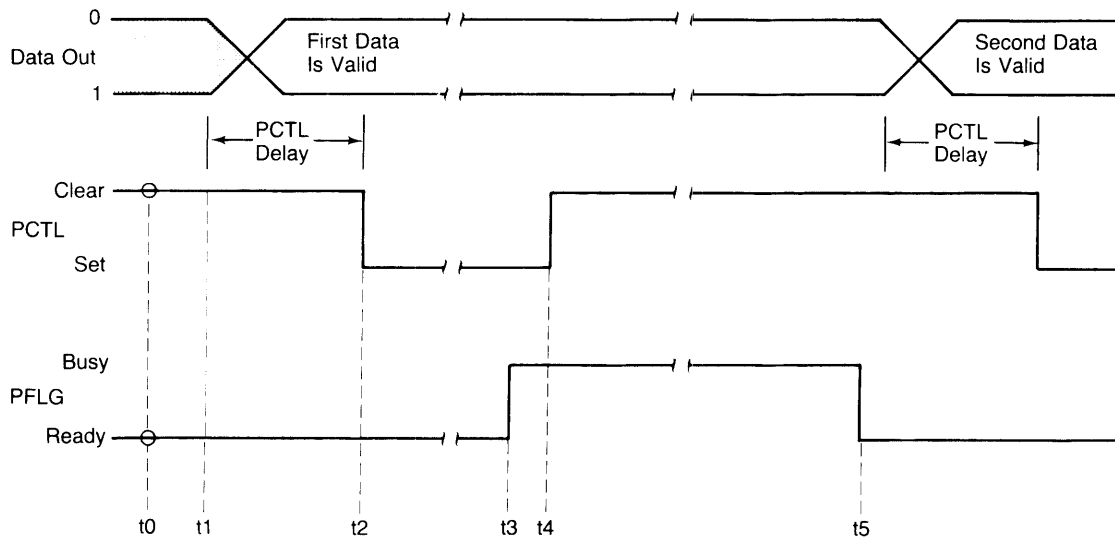
Diagram of Full-Mode OUTPUT Handshakes

With Full-Mode Handshakes. the computer first checks to see that the peripheral device is Ready before initiating the transfer of each byte/word (t0); with this handshake mode, the peripheral indicates **Ready when both PCTL is Clear and PFLG is Ready**. If the peripheral does not indicate Ready, the computer waits until a Ready is indicated.

When a Ready is sensed, the computer places data on the Data Out lines (t1) and drives the I/O line Low (not shown). The interface then waits the PCTL Delay time before initiating the transfer by placing PCTL in the Set state (t2).

The peripheral acknowledges the computer's request by placing the PFLG line Busy (t3); this PFLG transition automatically Clears the PCTL line (t4). However. the computer cannot initiate further transfers until the peripheral is Ready with Full-Mode Handshake; the peripheral is not Ready until both PCTL is Clear and PFLG is Ready (t5).

The data on the Data Out lines is held valid from the time PCTL is Set until after the peripheral indicates Ready. The peripheral may read the data any time within this time period.

The PCTL and PFLG lines are used in the same manner in Full-Mode ENTER Handshakes as in Full-Mode OUTPUT Handshakes. However, there are three options available as to when the peripheral's data may be valid: at the Ready-to-Busy transition of PFLG (BSY clock source), at the Busy-to-Ready transition of PFLG (RDY clock source), and when the Data In lines are read with a STATUS statement (READ clock source). The first two of these options are shown in the following two diagrams; the READ clock source is discussed later in "Designing Your Own Transfers".
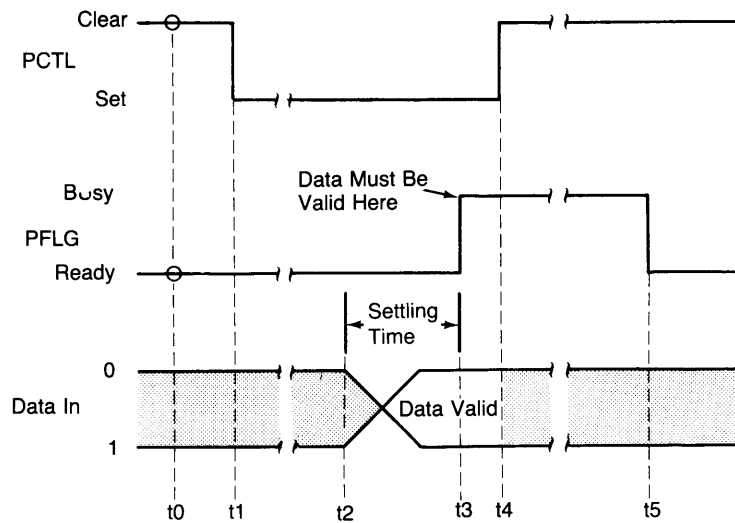
Full-Mode ENTER Handshake with BSY Clock Source

As with Full-Mode OUTPUT Handshakes, the computer first checks to see if the peripheral is Ready (t0); since PCTL is Clear and PFLG is Ready, the handshake may proceed. The computer places the I/O line in the High state (not shown) and then initiates the handshake by placing PCTL in the Set state (t1).

With the "BSY" clock source, the PFLG transition to the Busy state clocks the peripheral's data into the interface's Data-In registers; consequently, the peripheral must place data on the Data-In lines (t2), allowing enough time for the data to settle before placing PFLG in the Busy state (t3). This PFLG transition to the Busy state automatically Clears PCTL (t4). The next handshake may be initiated when PFLG is placed in the Ready state by the peripheral (t5).
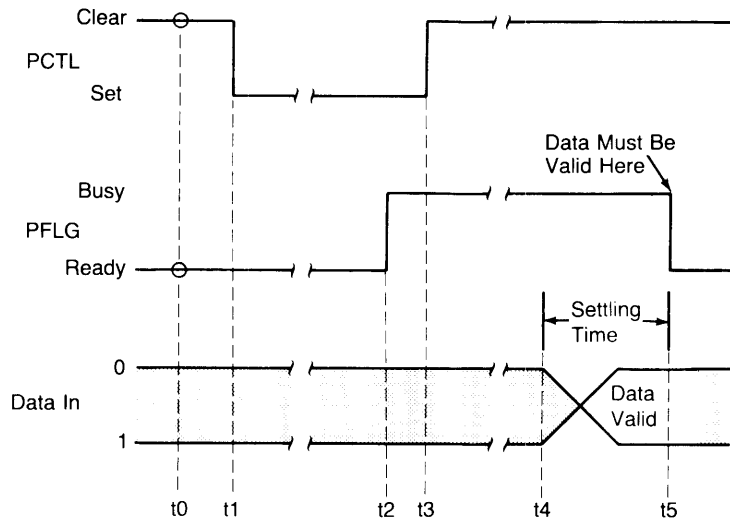
Full-Mode ENTER Handshake with RDY Clock Source

As with other Full-Mode Handshakes, the computer first checks to see if the peripheral is ready (t0). Since PCTL is Clear and PFLG is Ready, the computer may drive the I/O line High (not shown) and initiate the handshake by placing PCTL in the Set state (t1).

The peripheral may acknowledge by placing PFLG Busy (t2), which automatically Clears PCTL (t3). Unlike the previous example, this transition does not clock data into the interface Data-In registers. With the "RDY" clock source, the peripheral must place the data on the Data-In lines (t4), allowing enough time for the data to settle before placing PFLG in the Ready state (t5). The computer may then initiate a subsequent transfer.

Pulse-Mode Handshakes
The following drawings show the order of handshake-line events during Pulse-Mode Handshakes. Notice that the **main difference** between Full-Mode and Pulse-Mode Handshakes is that the **PFLG is not checked for Ready before the computer initiates Pulse-Mode Handshakes**; the computer may initiate a subsequent data transfer as soon as the PCTL line is Cleared by the Ready-to-Busy transition of PFLG.

Two cycles of data transfers are shown in these diagrams to illustrate that the computer need not wait for the PFLG = Ready indication with the Pulse-Mode Handshake. The first cycle shown in each diagram is a typical example of the first transfer of an I/O statement. The dashed PFLG line at the beginning of the second cycle shows that computer disregards whether or not PFLG is in the Ready state before the next transfer is initiated.

This absense of the PFLG check allows a **potentially higher data-transfer rate** than possible with the Full-Mode Handshake; however, in some cases, it also places additional timing restrictions on the peripheral's response time, as described in the text.
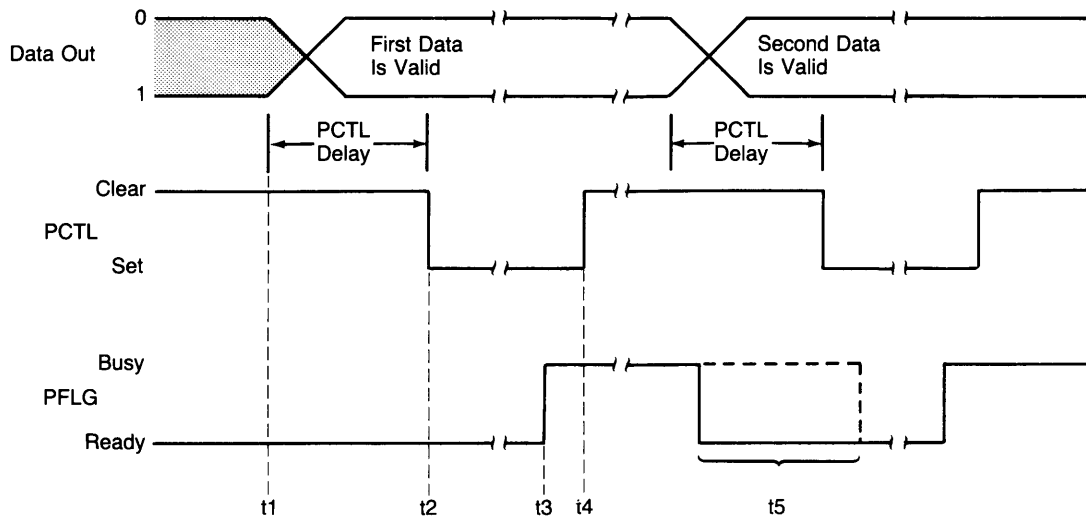
Busy Pulses With Pulse-Mode OUTPUT Handshake

The PFLG line is not checked for Ready before the computer drives the I/O line Low (not shown) and places data on the Data-Out lines (t1). A PCTL Delay time later, the interface initiates the transfer by placing PCTL in the Set state (t2).

The peripheral acknowledges by placing PFLG Busy (t3); this transition automatically Clears PCTL (t4). The dashed PFLG line shows that the computer may initiate another transfer any time after PCTL is Clear, possibly before the peripheral places PFLG in the Ready state (t5).

The Busy Pulse shown in the diagram is identical to the PFLG's response during the previous Full-Mode handshake; however, the Pulse-Mode Handshake works properly with this type of pulse **only** if the peripheral reads the data by the time PCTL is Clear (data should be read between t2 and t3). If the peripheral has not read the data by the time that PCTL is Clear, it might erroneously read the data for the second transfer, since the computer might have already changed the data and initiated the second transfer.
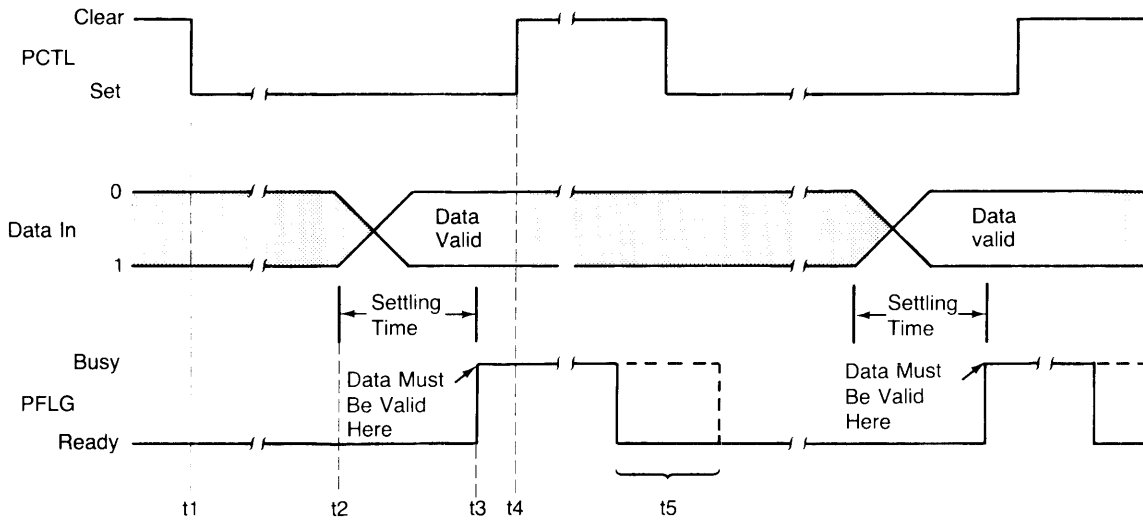
Busy Pulses With Pulse-Mode ENTER Handshakes (BSY Clock Source)

The computer does not have to check for PFLG to be Ready before placing I/O in the High state (not shown) and initiating the transfer by placing PCTL in the Set state (t1).

The peripheral must place data on the Data In lines (t2), allowing enough tome for the data to settle before placing PFLG in the Busy state (t3). This Ready-to-Busy transition of PFLG automatically Clears PCTL. The dashed PFLG signal shows that the next transfer may be initiated before PFLG indicates Ready.
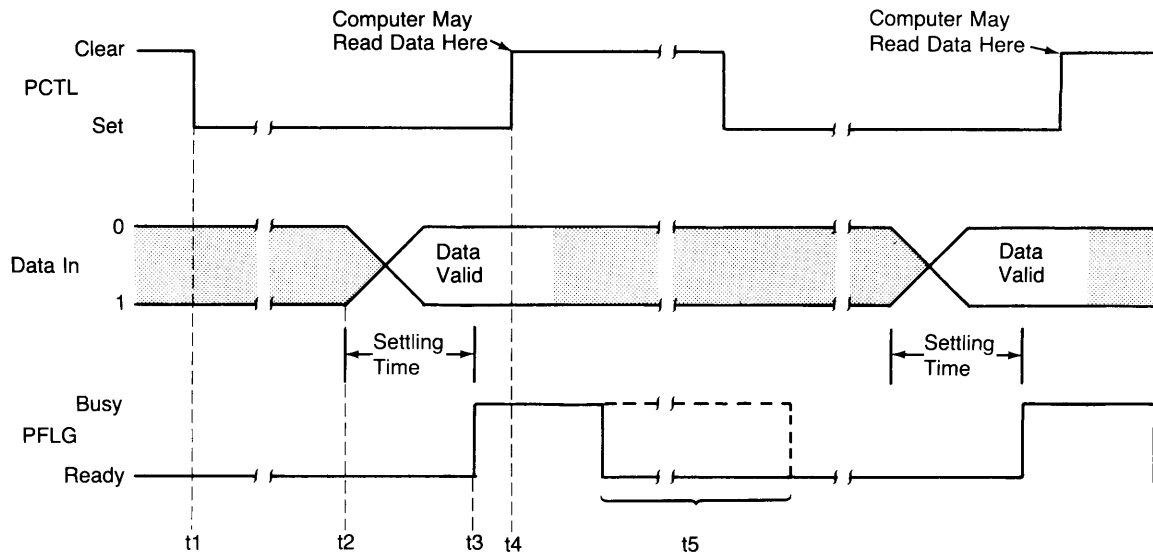
Busy Pulses With Pulse-Mode ENTER Handshakes (RDY Clock Source)

The computer does not have to check for PFLG to be Ready before placing I/O in the High state (not shown) and initiating the transfer by placing PCTL in the Set state (t1).

The peripheral must place data on the Data In lines (t2), allowing enough time for the data to settle before placing PFLG Busy (t3). This requirement **may seem contradictory**, since the clock source is the Busy-to-Ready transition of PFLG. However, with Pulse-Mode handshakes, the peripheral is assumed to be Ready whenever PCTL is Clear; consequently, the computer may read the data any time after PCTL is cleared by the Ready-to-Busy transition of PFLG. The PFLG transition to Busy Clears PCTL (t4), after which the peripheral may place PFLG Ready (t5).

Note

In order to use this type of pulse with the Pulse-Mode Handshake and RDY clock source, the peripheral must adhere to the stated timing restrictions.
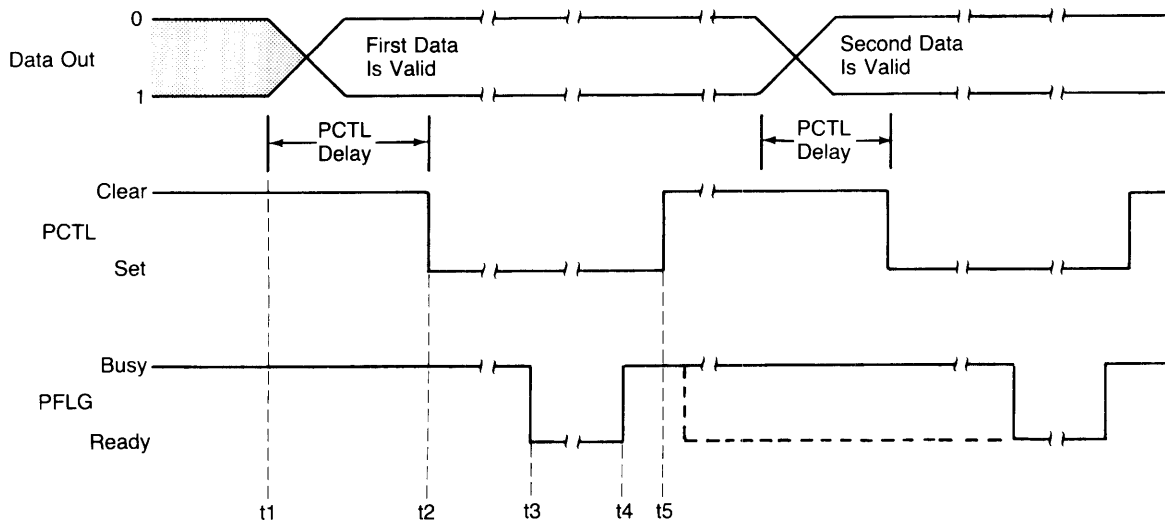
Ready Pulses With Pulse-Mode OUTPUT Handshakes

The PFLG line is not checked for Ready before the computer drives the I/O line Low (not shown) and places data on the Data Out lines (t1). A PCTL Delay time later the interface initiates the transfer by placing PCTL in the Set state (t2).

The peripheral later acknowledges by placing PFLG in the Ready state (t3). The handshake is completed by the peripheral placing PFLG in the Busy state (t4), which automatically Clears PCTL (t5).

If the peripheral uses the type of Ready pulses shown, either the Pulse-Mode handshake with default PFLG logic sense or Full-Mode handshake with inverted PFLG logic sense may be used. With this type of pulse, the data being output may be read by the peripheral as long as PCTL is Set.
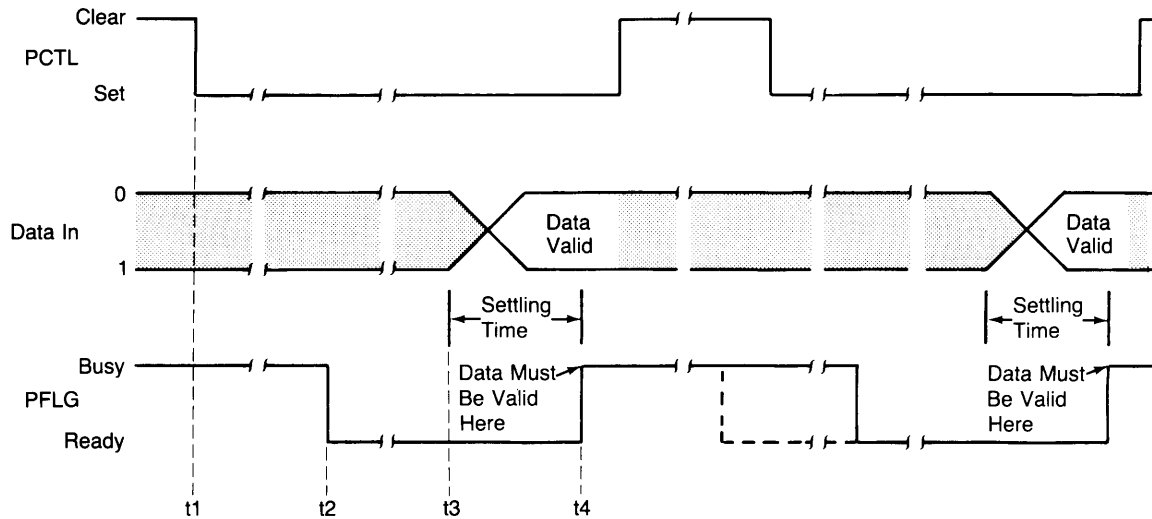
Ready Pulses With Pulse-Mode ENTER Handshakes (BSY Clock Source)

The computer does not have to check for PFLG to be Ready before placing I/O in the High state (not shown) and initiating the transfer by placing PCTL in the Set state (t1).

The peripheral acknowledges by placing PFLG in the Ready state (t2). The peripheral must place data on the Data In lines (t3), allowing enough time for the data to settle before placing PFLG in the Busy state (t4). With this type of pulse, events t2 and t3 may also occur in the reverse order.

The Ready-to-Busy transition of PFLG automatically Clears PCTL (t4). The dashed PFLG signal shows that the state of PFLG is not checked before the computer initiates a subsequent transfer.
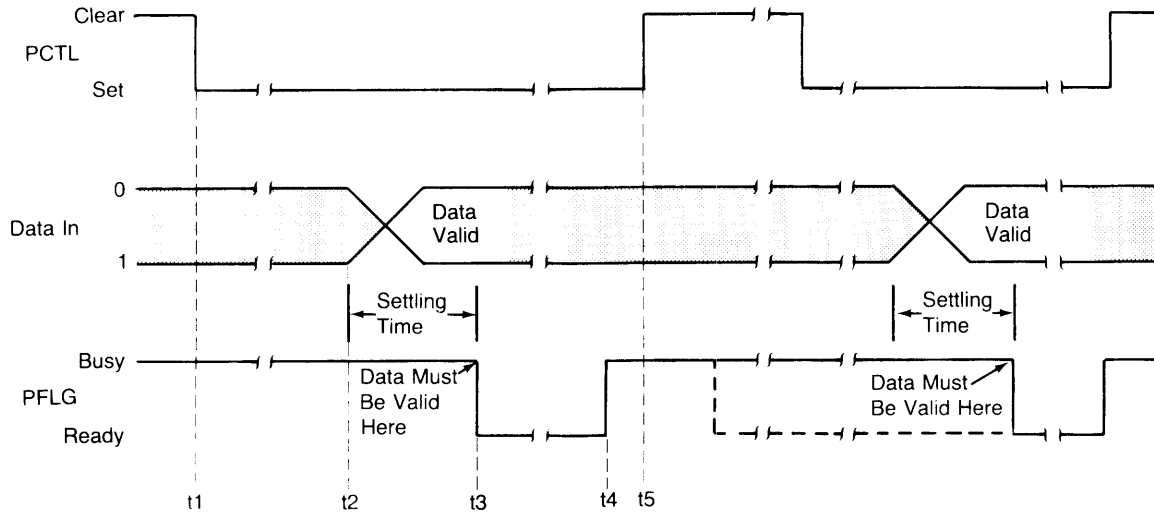
Ready Pulses With Pulse-Mode ENTER Handshakes (RDY Clock Source)

The computer does not have to check for PFLG to be Ready before placing I O in the High state (not shown) and initiating the transfer by placing PCTL in the Set state (t1).

The peripheral must place data on the Data In lines (t2), allowing enough time for the data to settle before placing PFLG Ready (t3). The peripheral places PFLG in the Busy state (t4), which automatically Clears PCTL (t5).

Interface Reset

The interface should always be reset before use to ensure that it is in a known state. All interfaces are automatically reset by the computer at certain times: when the computer is powered on, when the (RESET) key is pressed, and at other times described in the Reset Table[1]. The interface may be optionally reset at other times under control of BASIC programs. Two examples are as follows:

```
Gpio=12
CONTROL Gpio,0;1

Reset=1
CONTROL Gpio;Reset
```

The following action is invoked whenever the GPIO Interface is reset:

- The Peripheral Reset line (PRESET) is pulsed Low for at least 15 microseconds.
- The PCTL line is placed in the Clear state.
- If the DOUT CLEAR jumper is installed, the Data Out lines are all cleared (set to logic 0).
- The interrupt enable bit is cleared, disabling subsequent interrupts until re-enabled by the program.

The following lines are **unchanged** by a reset of the GPIO Interface:

- The CTL0 and CTL1 output lines.
- The I/O line.
- The Data Out lines, if the DOUT CLEAR jumper is not installed.

---

[1] The Reset Table is given in the Useful Tables of the *BASIC Language Reference.*

Outputs and Enters through the GPIO

This section describes techniques for outputting and entering data through the GPIO Interface. The mechanism by which data are communicated are the electrical signals on the data lines. The actual signals that appear on the data lines depend on three things: the data currently being transferred, how this data is being represented, and the logic sense of the data lines.

Brief explanations of ASCII and internal data representation are given in Chapter 2. Complete details of the freefield convention and effects of IMAGE specifiers during OUTPUT and ENTER statements are described in Chapters 4 and 5, respectively. The section of Chapter 10 called "The FORMAT OFF Attribute" describes how internal-form data are represented during OUTPUT and ENTER. This section gives simple examples of how several representations are implemented during OUTPUTs and ENTERs through the GPIO Interface.

ASCII and Internal Representations

When data are moved through the GPIO Interface, the **data are generally sent one byte at a time, with the most significant byte first**. This byte-mode transfer is independent of whether FORMAT ON or FORMAT OFF is the I/O path attribute. However, there are **two exceptions**; data are represented by words when the "W" image specifier is used and when numeric data are moved with reads of STATUS register 3 and writes to CONTROL register 3. The following diagrams illustrate which data lines are used during byte and word transfers.

```
         GPIO                Peripheral
       Interface               Device
     DO15 — DO8
         or             Upper 8 bits are not used
     DI15 — DI8          (all 0's during byte transfers).


     DO7 — DO0
         or             Only lower 8 bits are used.
     DI7 — DI0
```
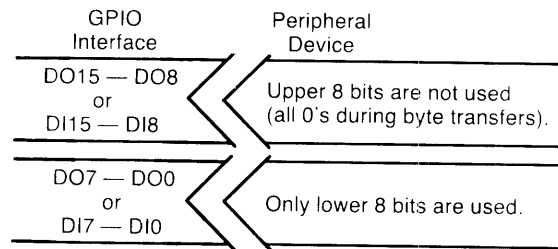
Diagram of Byte Transfers

Example Statements that Output Data Bytes

The following diagrams show the actual logic signals that appear on the least significant data byte (DO7 thru DO0) as the result of the corresponding OUTPUT statement: the most significant byte is always zeros with byte transfers. The actual logic levels depend on how the data lines are configured (i.e., as Low-true or High-true).

```
ASSIGN @Gpio TO 12
OUTPUT @Gpio;"ASCII"
```

Signal Line DO7 .... DO0	ASCII Char.
0 1 0 0 0 0 0 1	A
0 1 0 1 0 0 1 1	S
0 1 0 0 0 0 1 1	C
0 1 0 0 1 0 0 1	I
0 1 0 0 1 0 0 1	I
0 0 0 0 1 1 0 1	C_R
0 0 0 0 1 0 1 0	L_F

Signal Line DO7 .... DO0								ASCII Char.
0	0	1	0	1	1	0	1	–
0	1	1	0	0	1	0	0	4
0	0	1	0	1	1	1	0	.
0	0	1	1	0	0	0	0	0
0	0	1	1	0	0	0	0	0
0	0	0	0	1	1	0	1	C_R
0	0	0	0	1	0	1	0	L_F

```
Gpio=12
Number=-4
OUTPUT Gpio USING "MD.DD";Number
```

Signal Line DO7 .... DO0								ASCII Char.
0	1	0	1	0	1	0	1	U
0	1	0	0	1	1	0	0	L

```
ASSIGN @Gpio TO 12;FORMAT OFF
Integer_1=256*85+76
OUTPUT @Gpio;Integer_1
```

Signal Line DO7 .... DO0								ASCII Char.
0	0	0	0	0	0	0	0	N_u
0	0	0	0	0	0	0	0	N_u
0	0	0	0	0	0	0	0	N_u
0	0	0	0	0	1	0	0	E_t
0	0	1	1	0	0	0	1	1
0	0	1	1	0	0	1	0	2
0	0	1	1	0	0	1	1	3
0	0	1	1	0	1	0	0	4

```
ASSIGN @Gpio TO 12;FORMAT OFF
String$="1234"
OUTPUT @Gpio;String$
```

Example Statements that Enter Data Bytes

The following diagrams show the variable values that result from the logic signals being present during the corresponding ENTER statement on the least significant data byte (DI7 thru DI0); the most significant byte is always ignored with byte transfers. The actual logic levels required depend on how the data lines are configured (i.e., as Low-true or High-true).

Signal Line DI7 .... DI0								ASCII Char.
0	1	0	0	0	0	0	1	A

```
ENTER @Gpio USING "#,B";Byte
DISP "Value entered=";Byte

   Value entered= 65
```

Signal Line DI7 .... DI0								ASCII Char.
0	1	1	1	0	0	1	0	r
0	1	1	1	0	1	0	1	u
0	1	1	0	1	1	1	1	o
0	1	1	0	1	0	1	1	k
0	0	1	1	1	1	1	1	?
0	0	0	0	1	0	1	0	L_F

```
ENTER 12;String$
DISP "String entered= ";String$

   String entered= ruok?
```

	Signal Line						ASCII
	DI7			....		DI0	Char.

```
REAL Number            0 1 0 0   0 0 0 0    @
ASSIGN @Gpio TO 12     0 0 0 0   0 0 0 0    NU
ENTER @Gpio;Number     0 0 0 0   0 0 0 0
DISP "Number=";Number  0 0 0 0   0 0 0 0    ,
                       0 0 0 0   0 0 0 0    ,
Number= 2              0 0 0 0   0 0 0 0    ,
                       0 0 0 0   0 0 0 0
                       0 0 0 0   0 0 0 0    NU
```

```
GPIO                  Peripheral
Interface             Device

DO15 — DO8            Upper 8 bits are used only when:
  or
DI15 — DI8           1. Writing to CONTROL register 3
                        (reading from STATUS register 3).
                     2. The "W" image specifier is used.


DO7 — DO0
  or                 Lower 8 bits are used for ALL data
DI7 — DI0            transfers.
```
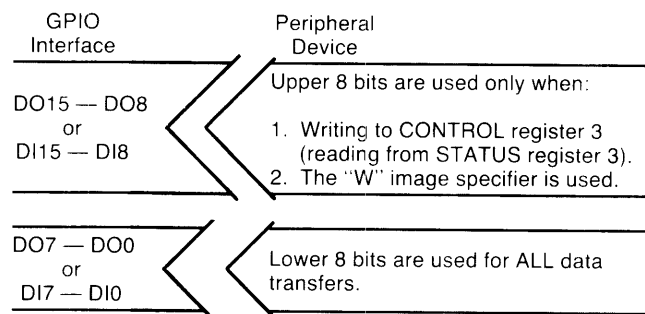
Diagram of Word Transfers

Example Statements that Output Data Words[1]

The following diagrams show the logic signals that appear on the Data Out lines as a result of the corresponding BASIC statements and numeric values. All numeric values are first rounded to an INTEGER value before being placed on the Data Out lines. The actual logic level that appears on each line depends on how the lines have been configured (i.e., as High-true or Low-true).

```
Word=3*256+3
OUTPUT @Gpio USING "#,W";Output_word
```

Signal Lines			
DO15 ...... DO8		DO7 .... DO0	
0 0 0 0 0 0 1 1		0 0 0 0 0 0 1 1	

```
Output_16_bits=-1
CONTROL Gp_isc,3;Output_16_bits
```

Signal Lines			
DO15 ...... DO8		DO7 .... DO0	
1 1 1 1 1 1 1 1		1 1 1 1 1 1 1 1	

It is important to note that no output handshake is executed when the CONTROL statement is executed; only the states of the Data Out lines and the I/O line are affected. Handshake sequence, if desired, must be performed by BASIC statements in the program. See "Designing Your Own Transfers" for design suggestions.

[1] Data are automatically sent as words when using an I/O path with the WORD attribute. See Chapter 10 for further information.

Example Statements that Enter Data Words[1]

The following diagrams show the variable values that result from entering the logic signals on the Data In lines. Note that all sixteen-bit values entered are interpreted as INTEGER values.

Signal Lines		
DI15 .... DI8	DI7 .... DI0	
0 0 0 0 0 0 0 1	1 1 1 1 1 1 1 1	

```
ENTER 12 USING "#,W";Enter_16_bits
DISP "INTEGER entered=";Enter_16_bits
```

```
INTEGER entered= 511
```

Signal Lines		
DI15 .... DI8	DI7 .... DI0	
1 1 1 1 1 1 1 0	0 0 0 0 0 0 0 0	

```
STATUS Gp_isc,3;Enter_16_bits
DISP "INTEGER entered=";Enter_16_bits
```

```
INTEGER entered= -512
```

It is important to note that no enter handshake is performed when the STATUS statement is executed. The only actions taken are the I/O line being placed in the High state and the Data In registers being read. If an enter handshake is required, it must be performed by the BASIC program. See "Designing Your Own Transfers" for design suggestions.

Remember also that the Data In Clock source is solely determined by the switch setting on the interface card. Thus, when the STATUS statement is used to read the Data In lines, the data on the lines may or may not be clocked into the registers when the statement is executed. If the data are to be clocked in by the STATUS statement, the "READ" clock source must be selected. See the installation manual for further details.

GPIO Timeouts

Timeout events were generally discussed in Chapter 7. However, specific details of the affects of the time parameter on the event's occurrence were not described. This section explains how the time parameter is measured and describes typical service routines.

Timeout Time Parameter

There are two general time intervals measured and compared to the specified TIMEOUT time. The first interval is measured between the computer initiating the first handshake (PCTL = Set) and the peripheral signalling Ready (with the PFLG line). If the peripheral does not indicate readiness[2] by the specified TIMEOUT time parameter, a TIMEOUT event occurs.

The time elapsed during each handshake is also measured and compared to the TIMEOUT time. The timing begins when the transfer is initiated (PCTL Set by the computer) and, in general, ends when the peripheral responds on the PFLG line.

---

[1] Data are automatically received as words when using an I/O path with the WORD attribute. See Chapter 10 for further information.

[2] The computer optionally reads the state of the PSTS line before initiating the transfer. See "Using the PSTS Line" for further details.

Keep in mind that the TIMEOUT time parameter specifies the **minimum** time that the computer will wait before initiating the ON TIMEOUT branch. However, the computer may occasionally wait an additional 25% of the specified time parameter before initiating the branch. For instance, if a time of 0.4 seconds is specified, the computer will wait at least 0.4 seconds for the handshake to be completed, but it may occasionally wait up to 0.5 seconds before initiating the ON TIMEOUT branch.

Timeout Service Routines

The service routine usually responds by determining if the peripheral is functioning properly ("OK") or is down ("not OK"). The simplest action that might be taken by the computer is to read the state of the PSTS signal line, as shown in the following service routine.

```
100    Gpio=12
110    ON TIMEOUT Gpio,.08 GOSUB Gpio_down
         :
200    OUTPUT Gpio;String$
210    ! Next line.
         :
300 Gpio_down: STATUS Gpio,5;Periph_status
310                 Psts=BIT(Periph_status,3)  ! Read PSTS.
320                 IF NOT Psts THEN
330                    PRINT "GPIO interface is "
340                    PRINT "non-functional"
350                    PRINT "Program paused."
360                    PAUSE
370                 ELSE
380                    ! Take other action.
390                 END IF
400       RETURN
```

A TIMEOUT has been set up to occur if the peripheral takes approximately more than .08 second to complete its response during a data transfer; how the peripheral completes its response depends on the handshake mode currently selected. With Pulse-Mode Handshakes, the peripheral completes its response by using PFLG to Clear PCTL; with Full-Mode Handshakes, the response is complete **only** after PCTL has been Cleared **and** PFLG is in the Ready state.

When a TIMEOUT occurs, the computer automatically executes an Interface Reset; the PCTL line is Set and then Cleared, and the PRESET line is pulsed Low. See the section called "Interface Reset" for further effects. The Service routine checks the PSTS line to see if the peripheral is OK or not OK. If not OK, a message is displayed and the program is paused; if OK, program execution is returned to the line following that in which the TIMEOUT occurred. Service routine may be programmed to attempt the transfer again, if desired; however, the automatic Reset performed when the TIMEOUT occurred may make this type of response difficult to implement.

Using Alternate Data Representations

As with any other interface, representations other than the ASCII or internal representations may sometimes be more meaningful to the peripheral. This section briefly describes a few techniques for implementing alternate data representations.

BCD Representation

With OUTPUT and ENTER statements, numeric values are either represented by ASCII characters or by one of the internal representations (INTEGER or REAL). Another common method of representing numeric data is to use a four-bit, binary-coded decimal (BCD) characters. Only ten of the available sixteen bit patterns need to be used to represent decimal digits "0" through "9". The remaining six patterns can be used for sign, decimal point, exponent, and other special characters, as required by the application.

The following bit patterns have been chosen arbitrarily to correspond to numeric characters[1]. Note that this representation cannot be used if more than six other characters are to be represented.

Decimal Digit	Bit Pattern MSB			LSB	Other Character	Bit Pattern MSB			LSB
0	0	0	0	0	Line-Feed	1	0	1	0
1	0	0	0	1	+	1	0	1	1
2	0	0	1	0	,	1	1	0	0
3	0	0	1	1	−	1	1	0	1
4	0	1	0	0	E	1	1	1	0
5	0	1	0	1	.	1	1	1	1
6	0	1	1	0					
7	0	1	1	1					
8	1	0	0	0					
9	1	0	0	1					

The following subprogram assumes that BCD numbers are to be entered through the GPIO Interface. Sixteen BCD characters are represented by four 16-bit words from the peripheral. The sixteen four-bit BCD characters have the following general format.
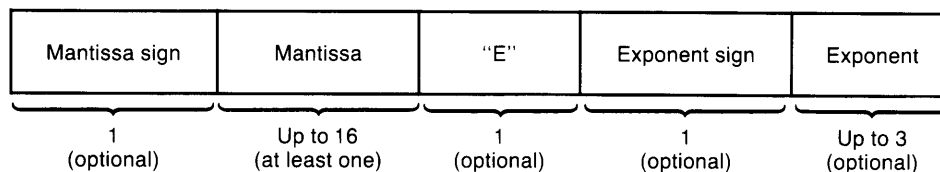
Mantissa sign	Mantissa	"E"	Exponent sign	Exponent
1 (optional)	Up to 16 (at least one)	1 (optional)	1 (optional)	Up to 3 (optional)

---

[1] This is also the data representation used by the HP 98623 BCD Interface. See Chapter 17 for further information.

Each BCD character is represented by four bits of data. The first word entered contains the four most significant BCD characters, and the last word contains the least significant. The program changes the BCD characters to their ASCII representation and then uses the number builder to generate the corresponding numeric value.

```
100    ASSIGN @Gpio TO 12
110    !
120    ! Define conversion string.
130    Conv$="0123456789"&CHR$(10)&"+,-E,"
140    !
150    CALL Enter_bcd(@Gpio,Conv$,Number)
160    OUTPUT 1;"The BCD number is ";Number
170    !
180    END
190    !
200    !
210    SUB Enter_bcd(@Device,Conv$,Number)
220    COM /Enter_bcd/ INTEGER Word(1:4)
230    !
240    ! Enter 4 words (=16 BCD digits).
250    ENTER @Device USING "#,W";Word(*)
260    !
270    FOR W=1 TO 4 ! Process four words.
280    !
290    ! Shift right multiples of four bits.
300       FOR Bits_rt=12 TO 0 STEP -4
310          Shifted_word=SHIFT(Word(W),Bits_rt)
320          Four_lsb=BINAND(Shifted_word,15) ! Mask MSB's.
330          Ascii_char$=Conv$[Four_lsb+1;1] ! LSB's = index.
340          Number$=Number$&Ascii_char$
350       NEXT Bits_rt
360       !
370    NEXT W
380    !
390    ENTER Number$;Number ! Use number builder.
400    SUBEND ! Returns BCD number as "Number".
```

Character Conversions

One of the most common needs of a computer is to convert[1] certain unused or disallowed bit patterns into meaningful or allowed bit patterns. A typical example is to change the radix character from a decimal point to a comma. For instance, the following ASCII characters represent the same number.

U.S. Representation	European Representation
1,234,567.89	1.234.567,89

A remedy is needed to allows these types of numbers to be entered through the number builder. To enter a number with the preceding European format, the commas must be changed to periods and the periods changed to spaces. The following routine changes the numeric radix from the European to the US convention when numeric data are entered through the GPIO.

```
100    ! Generate string with no conversions,
110    DIM Conv$[256]
120    FOR Code=0 TO 255
130       Conv$[Code+1]=CHR$(Code)
140    NEXT Code
150    !
160    ! Then define the conversions,
170    Conv$[NUM(",")+1;1]=" "  ! Change "," to " "
180    Conv$[NUM(",")+1;1]=","  ! Change "," to ","
190    !
200    !
210    Number$="123,456,789"
220    PRINT "Before conversion  ";Number$
230    CALL Convert(Conv$,Number$)
240    PRINT "After conversion  ";Number$
250    !
260    END
270    !
280    !
290    SUB Convert(Conv$,Data$)
300    !
310    FOR Char_pos=1 TO LEN(Data$)
320       Index=NUM(Data$[Char_pos])+1
330       Data$[Char_pos;1]=Conv$[Index;1]
340    NEXT Char_pos
350    !
360    ! Returns Data$ with converted characters,
370    SUBEND
```

If more characters are to be converted, simply change the default (standard ASCII) character in Conv$ to the desired code. The speed of the conversion is not affected by the number of characters to be converted. This routine works for either input or output, but the characters to be converted must be in a string variable.

---

[1] This is also the data representation used by the HP 98623 BCD Interface. See Chapter 17 for further information.

GPIO Interrupts

This section describes the types of and techniques for using the interrupts available on the GPIO Interface.

Types of Interrupt Events

The GPIO Interface can sense **two interrupt events**: the first is the interface becoming "Ready" for suqsequent handshakes, and the second is the External Interrupt Request line (EIR) being driven to logic low by the peripheral. As with all interfaces, both events initiate identical computer responses — the service routine must be able to determine which of these interrupts have occurred if both are enabled to initiate interrupts.

Both of these types of interrupt events are **level-sensitive**; in other words, the signal that caused the event should be maintained until the program has time to determine which event has caused the interrupt. Further explanation follows in this section.

Setting Up and Enabling Events

When either event occurs, the interrupt is logged by the BASIC operating system. After logging the occurrence, any further interrupts from the GPIO Interface are automatically disabled until specifically enabled by a program. All further computer responses to either event depend entirely on the BASIC program currently in memory.

The following program segment shows the steps involved in setting up and enabling Ready interrupts.

```
100     Gpio=12
110     ON INTR Gpio GOSUB Gpio_serv
120     !
130     Mask=2
140     ENABLE INTR Gpio;Mask
```

The value of the interrupt mask determines which, if any, of the GPIO interrupt events are to be enabled to initate the corresponding branch. Bits of the Interrupt Mask register have the following definitions.

Interrupt Enable Register   **(ENABLE INTR)**

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
			Not Used			Enable Interface Ready Interrupts	Enable EIR Interrupts
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Interface Ready — Setting this bit (1) enables an interrupt to initiate the ON INTR branch when the interface detects that it is Ready to handshake data. If Full-Mode Handshake is selected (with the Option Select switch), the Ready event is PCTL = Clear **and** PFLG = Ready. With Pulse-Mode Handshake, the event is PCTL = Clear (independent of the state of PFLG).

External Interrupt Request — Setting this bit (1) enables an interrupt to initiate the ON INTR branch when the interface senses an External Interrupt Request (EIR line = Low).

Interrupt Service Routines

If both events are enabled, the service routine must be able to differentiate between the two. And, if both have occurred, the service routine must be able to service both causes. The following registers contain the current state of the Interface Ready flag and EIR signal lines, from which the interrupt cause(s) may be determined.

Status Register 4   **Interface Ready**

Interface is ready for a subsequent data transfer; 1 = Ready, 1 = Busy.

Status Register 5   **Peripheral 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	0	PSTS Ok	EIR Line Low	STI1 Line Low	STI0 Line Low
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

As mentioned in preceding paragraphs, these two interrupt causes are both **level-sensitive** events, not edge-triggered events. This fact has **two important implications**. The **first** is that, for an event to be recognized, the corresponding signal line must be held in the interrupting state until the computer can interrogate the line's logic state. If the signal line's state is changed before the service routine checks the line, the interrupt may be "missed". This will happen only if both events are enabled; if only one event is enabled, determining the cause may not be necessary.

The **second implication** is that the service routine must be able to acknowledge the request in order for the peripheral device to remove the request. If the request is not removed after service, the same request may be serviced more than once.

The following program shows a simple example of servicing an External Interrupt Request. Note that only EIR-type interrupts have been enabled and that the peripheral device provides its own interrupt cause with signals on the STI0 and STI1 lines.

```
100     PRINTER IS 1
110     Gpio=12
120     CONTROL Gpio;1 ! Reset Interface,
130     !
140     ON INTR Gpio GOSUB Gpio_serv
150     ENABLE INTR Gpio;1 ! Enable EIR-type only,
160     !
170     ! Show concurrent processing,
180 Loop: Counter=Counter+1
190         DISP Counter
200         GOTO Loop
210     !
220     STOP
230     !
240 Gpio_serv: !
250     STATUS Gpio,5;Periph_status   ! Check EIR line,
260     IF BIT(Periph_status,2) THEN ! EIR interrupt,
270     !
280       IF BIT(Periph_status,0) THEN ! STI0=True,
290         BEEP
300         PRINT "Improper value; type in correct"
310         PRINT "value, and press ENTER,"
320         PRINT
330         ENTER 2;Value
340         OUTPUT Gpio;Value
350       END IF
360       !
370       IF BIT(Periph_status,1) THEN ! STI1=True,
380         BEEP
390         PRINT "Reading of:";Reading;" out of range"
400         PRINT "No other action will be taken,"
410         PRINT
420         WAIT 2
430         BEEP
440       END IF
450       !
460     END IF
470     !
480     ! Put Ready service routine here,
490     !
500     !
510     ENABLE INTR Gpio        ! Use same mask,
520     RETURN
530     !
540     END
```

A slightly different method that peripherals use to communicate the cause of their interrupt request is to place the interrupt cause on the data lines concurrent with the interrupt request. The service routine can determine the cause by reading STATUS register 3 and take the appropriate action.

Notice that the service routine indicates a likely place for a Ready-interrupt service routine. The Service routine must check for the Ready condition, acknowledge the interrupt, and then take the desired action. In this case, no service action has been defined because Ready interrupts have not been enabled. The next section provides an example of a Ready interrupt service routine.

Designing Your Own Transfers

Other specialized methods of handshaking data can be designed according to your specific needs. The methods of synchronizing data transfers are as flexible as the GPIO Interface hardware. However, the general techniques will probably still require the fundamental handshake features: initiation by the sending device, acknowledgement from the receiving device, and agreement as to when the data are valid. The TRANSFER statement can be used to transfer data. See Chapter 11 for further information.

A wide choice of initiating events is available; obvious possibilities include use of the PCTL, EIR, or CTL0 (or CTL1) lines to signal the start of the transfer. Data can be placed on the Data Out lines by writing to CONTROL register 3, or data can be clocked into the Data In registers by reading STATUS register 3. Sensing acknowledgement from the peripheral can be accomplished by reading the state of such lines as PFLG, PSTS, EIR, or STI0 (or STI1).

The feature common to all of these methods is that each byte (or word) of data must be transferred individually. If an entire block of data is to entered or output, the BASIC program that implements the transfer must keep a "pointer" to which byte/word is to be transferred.

Full Handshake Transfer

The following program implements a handshake similar to the Full OUTPUT Handshake by controlling the PCTL and sensing the PFLG and PCTL lines. Timeout capability can easily be included in the routine, if so desired.

```
100    DATA 65,66,67,68,69
110    !
120    STATUS 12,5;Periph_status     ! Chech PSTS.
130    IF BIT(Periph_status,3) THEN  ! PSTS True.
140    !
150      FOR Char=1 TO 5
160          READ Code
170 Wait:    STATUS 12,4;Interface_ready
180          IF NOT Interface_ready THEN Wait
190 Output: CONTROL 12,3;Code        ! Data onto lines.
200          CONTROL 12,1;1          ! Set PCTL.
210      NEXT Char
220      !
230    ELSE                          ! PSTS False.
240      PRINT "Peripheral error"
250      PAUSE
260    END IF
270    !
280    END
```

Notice that each byte of data must be output separately and that the program must keep track of which byte, of several, is to be sent. Keep in mind that the data written to CONTROL register 3 is **16-bit words**; in this case, the most significant eight bits (byte) is all zeros. Also, using FOR...NEXT loops to index each byte/word to be sent may not be the most expedient way of sending data, so your particular application may use alternative methods handling the data.

The following subprogram implements a handshake similar to the Full ENTER handshake.

```
170    SUB Enter_word(@Device,Data_word)
180    !
190 Wait1: STATUS 12,4;Interface_ready
200          IF NOT Interface_ready THEN Wait1
210          STATUS 12,3;Dummy_read  ! I/O High.
220          CONTROL 12,1;1          ! Set PCTL.
230 Wait2: STATUS 12,4;Interface_ready
240          IF NOT Interface_ready THEN Wait2
250          STATUS 12,3;Data_word   ! Enter word.
260          !
270      SUBEND
```

The appropriate Data-In Clock source should be selected to ensure the data are clocked into the registers when valid. Refer to the installation manual for further details.

Interrupt Transfers

The interrupt capabilities of the GPIO Interface can be used to synchronize the transfer of data between the computer and peripheral. These examples describe simple methods of synchronizing the transfer of data by using both the EIR and the PFLG line. See the section of this chapter called "GPIO Interrupts" for further explanation of GPIO interrupts in general.

General interrupt transfers through the GPIO Interface involve the following elements:

- placing data on (or reading data from) the data lines
- signaling to the peripheral device to initiate the transfer
- continuing processing until an interrupt is received, at which time the handshake is finished and transfer of the next byte/word can be initiated.

Examples of using Ready interrupts to implement interrupt transfers are given in the remainder of this section.

Ready Interrupt Transfers

The Ready interrupt event occurs when the GPIO Interface becomes "Ready". Whether or not the GPIO Interface is Ready depends on the currently selected handshake mode. If Full-Mode Handshake is selected, the interface is Ready if **both** the PFLG line is Ready and the PCTL line is Clear; if Pulse-Mode is selected, the interface is Ready if PCTL is in the Clear state, regardless of the state of PFLG. The following program shows how to implement Ready interrupt transfers.

```
100     PRINTER IS 1
110     Gpio=12
120     CONTROL Gpio;1 ! Reset Interface.
130     ON INTR Gpio GOSUB Ready_xfer
140     !
150     DIM Data_out$[256]
160     Data_out$="123ABC"
170     Pointer=1
180     Size=LEN(Data_out$)
190     !
200     ! Initiate the transfer.
210     GOSUB Ready_xfer
220     !
230     ! Show concurrent processing.
240 Loop: Counter=Counter+1
250         DISP Counter
260         GOTO Loop
270     !
280     STOP
290     !
300     ! The branch to this subroutine is initiated
310     ! first by the program, but thereafter by
320     ! Ready Interrupt events.
330     !
340 Ready_xfer: !
350                     !
360     IF Pointer<=Size THEN
370         Byte_out=NUM(Data_out$[Pointer;1])
380         PRINT Data_out$[Pointer;1];" sent"
390         CONTROL Gpio,3;Byte_out ! Place data on lines.
400         Pointer=Pointer+1
410         CONTROL Gpio,1;1          ! Set PCTL.
420         ENABLE INTR Gpio;2        ! Enable Ready INTR's.
430         RETURN
440         !
450     ELSE
460         DISABLE INTR Gpio         ! Disable after done.
470         RETURN
480         !
490     END IF
500     !
510     !
520     END
```

Interrupt transfers that use the EIR line are similar to Ready interrupt transfers. The main difference is that the interrupt-initiating event is the EIR line, rather than the PCTL line (and PFLG if in Full Handshake mode) indicating Interface Ready.

Using the Special-Purpose Lines

Four special-purpose signal lines are available for a variety of uses. Two of these lines are available for output (CTL0 and CTL1), and the other two are used as inputs (STI0 and STI1).

Driving the Control Output Lines

Setting bits 0 and 1 of GPIO CONTROL register 2 places a logic low on CTL0 and CTL1, respectively. The definition of this CONTROL register is shown in the following diagram.

Control Register 2   **Peripheral Control**

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not Used					PSTS Error (1 = Report; 0 = Ignore)	Set CTL1 (1 = Low; 0 = High)	Set CTL0 (1 = Low; 0 = High)
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

```
Ctl0=0   ! Clear.
Ctl1=1   ! Set.
CONTROL 12,2;Ctl1*2+Ctl0
```

As indicated in the diagram, setting a bit in the register places the corresponding line Low, while clearing the bit places a logic High on the line. The logic polarity of these signals cannot be changed. The signal remains on these lines until another value is written into the CONTROL register, and Reset has no effect on the state of either line.

Interrogating the Status Input Lines

The state of both status input lines STI0 and STI1 are determined by reading bits 0 and 1 of STATUS register 5, respectively. A logic "1" in a bit position indicates that the corresponding line is at logic Low, and a "0" indicates the opposite logic state. This logic polarity cannot be changed. The definition of GPIO STATUS register 5 is shown below.

Status Register 5   **Peripheral 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	0	PSTS Ok	EIR Line Low	STI1 Line Low	STI0 Line Low
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

```
STATUS 12,5;P_status
Sti0=BIT(P_status,0)
Sti1=BIT(P_status,1)
```

Reading this register returns a numeric value that reflects the logic states of these lines **at the instant the computer reads the interface lines**; the state of these lines are not latched by any internal or external event.

Using the PSTS Line

The Peripheral Status line (PSTS) is generally used to indicate whether or not the peripheral device is functional. The current state of PSTS may be checked by reading STATUS Register 5 (bit 3). It may also optionally be checked automatically at the **beginning** of an OUTPUT or ENTER statement; normally, it is not checked. This feature is only enabled by by setting Bit 2 of CONTROL register 2.

Control Register 2   **Peripheral Control**

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not Used					PSTS Error (1 = Report; 0 = Ignore)	Set CTL1 (1 = Low; 0 = High)	Set CTL0 (1 = Low; 0 = High)
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

When Bit 2 is set and PSTS is false at the beginning of either an OUTPUT or ENTER statement, Error 172 (Peripheral error) is reported. The error must be trapped with ON ERROR, since it generates no INTR or TIMEOUT branch.

Summary of GPIO Status and Control Registers

Status Register 0   Card identification = 3

Control Register 0   Reset interface if non-zero

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
Interrupts Are Enabled	An Interrupt Is Currently Requested	Interrupt Level Switches (Hardware Priority)		Burst-Mode DMA	Word-Mode DMA	DMA Channel 1 Enabled	DMA Channel 0
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Control Register 1   Set PCTL if non-zero

Status Register2

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	Handshake In Process	Interrupts Are Enabled	Reserved For Future Use
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Control Register 2   **Peripheral Control**

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not Used					PSTS Error (1 = Report; 0 = Ignore)	Set CTL1 (1 = Low; 0 = High)	Set CTL0 (1 = Low; 0 = High)
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Status Register 3   Data In (16 bits)

Control Register 3   Data Out (16 bits)

Status Register 4   **Interface Ready**

Interface is Ready for a subsequent data transfer;   1 = Ready, 0 = Busy.

Status Register 5
Most Significant Bit

Peripheral Status

Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	PSTS Ok	EIR Line Low	STI1 Line Low	STI0 Line Low
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Interrupt Enable Register
Most Significant Bit

(ENABLE INTR)

Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not Used						Enable 1Interface Ready Interrupts	Enable EIR Interrupts
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Summary of GPIO READIO and WRITEIO Registers

This section describes the GPIO Interface's READIO and WRITEIO registers. Keep in mind that these registers should be used **only** when you know the exact consequences of their use, as using some of the registers improperly may result in improper interface behavior. If the desired operation can be performed with STATUS or CONTROL, you should not use READIO or WRITEIO.

GPIO READIO Registers

Register 0 — Interface Ready
Register 1 — Card Identification
Register 2 — Undefined
Register 3 — Interrupt Status
Register 4 — MSB of Data In
Register 5 — LSB of Data In
Register 6 — Undefined
Register 7 — Peripheral Status

READIO Register 0   **Interface Ready**
A 1 indicates that the interface is Ready for subsequent data transfers, and 0 indicates Not Ready.

READIO Register 1   **Card Identification**
This register always contains 3, the identification for GPIO interfaces.

READIO Register 3   **Interrupt Status**

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Interrupts Are Enabled	An Interrupt Is Currently Requested	Interrupt Level Switches (Hardware Priority)		Burst-Mode DMA	Word-Mode DMA	DMA Channel 1 Enabled	DMA Channel 0 Enabled
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

READIO Register 4   **MSB of Data In**

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DI15	DI14	DI13	DI12	DI11	DI0	DI9	DI8
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

READIO Register 5 **LSB of Data In**

Most Significant Bit Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DI7	DI6	DI5	DI4	DI3	DI2	DI1	DI0
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

READIO Register 7 **Peripheral 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	0	PSTS Ok	EIR Line Low	STI1 Line Low	STI0 Line Low
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

GPIO WRITEIO Registers

WRITEIO Register 0 — Set PCTL
WRITEIO Register 1 — Reset Interface
WRITEIO Register 2 — Interrupt Mask
WRITEIO Register 3 — Interrupt and DMA Enable
WRITEIO Register 4 — MSB of Data Out
WRITEIO Register 5 — LSB of Data Out
WRITEIO Register 6 — Undefined
WRITEIO Register 7 — Set Control Output Lines

WRITEIO Register 0 **Set PCTL**

Writing any non-zero numeric value to this register places PCTL in the Set state; writing zero causes no action.

WRITEIO Register 1 **Reset Interface**

Writing any non-zero numeric value to this register resets the interface.

WRITEIO Register 2   Interrupt 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						Enable Interface Ready Interrupts	Enable EIR Interrupts
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

WRITEIO Register 3   Interrupt and DMA Enable

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Enable Interrupts	Not Used			Enable Burst-Mode DMA	Enable Word-Mode DMA	Enable DMA Channel 1	Enable DMA Channel 0
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

WRITEIO Register 4   MSB of Data Out

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DO15	DO14	DO13	DO12	DO11	DO10	DO9	DO8
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

WRITEIO Register 5   LSB of Data Out

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DO7	DO6	DO5	DO4	DO3	DO2	DO1	DO0
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

WRITEIO Register 7 — Set Control Output Lines

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not Used						Set CTL1 (1 = Low; 0 = High)	Set CTL0 (1 = Low; 0 = High)
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

| The BCD Interface | Chapter 17 |

This chapter should be used in conjunction with the *HP 98623 BCD Interface Installation Note*. The best way to use these two documents is to first read the section of this chapter called "Brief Description of Operation" to see how the interface works with the BASIC language. Within this section is information about the interface's modes of operation that will help you to understand how you might use the interface for your application. Second, read "Configuring the Interface" while referring to the Installation Note as necessary to configure and connect the interface according to your application's needs. The reason for this order is that you will probably be able to configure and use the interface once you know a little about how it works.

The main section of the chapter presents several techniques for using the interface to move data between the computer and peripheral devices using BASIC programs.

Brief Description of Operation

The HP 98623 Interface consists of data registers and handshake circuitry required to transfer data to and from the computer using either BCD or binary data formats. The interface cable contains the following sixty-four conductors:

- forty data, two sign, and one overload signal lines used to enter data from the peripheral
- eight lines used to output data to the peripheral
- two sets of handshake lines (two wires per set)
- one reset line to the peripheral device
- one five-volt logic line
- five logic (signal) grounds and two safety (shield) grounds

Data Representations and Formats

The BCD interface can be used to transfer data using one of two data representations: BCD (binary-coded decimal) and binary representations. BCD is the default data representation; the binary representation may be selected by software (as described in the configuration section).

The BCD Data Representation

When the BCD representation is in use, data lines are handled in groups of four lines each, with each group representing one BCD digit. The sixteen possible combinations of logic states and corresponding characters which each four-line group may represent are as follows:

Data Line Logic Sense (MSB) (LSB)	Character Represented	Data Line Logic Sense (MSB) (LSB)	Character Represented
0 0 0 0	0	1 0 0 0	8
0 0 0 1	1	1 0 0 1	9
0 0 1 0	2	1 0 1 0	line-feed
0 0 1 1	3	1 0 1 1	+
0 1 0 0	4	1 1 0 0	.
0 1 0 1	5	1 1 0 1	—
0 1 1 0	6	1 1 1 0	E
0 1 1 1	7	1 1 1 1	.

When the BCD representation is in use, the data lines are read sequentially in groups of four lines each. For each four-bit BCD character read, a corresponding ASCII character (listed above) is generated. Operating system "drivers" control both the sequence of reading the BCD-character groups and the generation of the appropriate ASCII character which each group represents. The sequence used by the drivers and the resultant numeric value entered depends on which BCD format is currently in use: Standard or Optional format.

Standard Format

The Standard BCD format is used to connect **one peripheral** to the computer. The data lines are arranged as follows to form two numbers: one mantissa sign bit, eight BCD mantissa characters, one exponent sign bit, and one BCD exponent character form the first number; one overload-indicator bit and one BCD character are combined to form the second number.

The following diagram shows how the signal lines are organized in Standard format (i.e., the order in which the lines are read with ENTER statements). The notation used with these diagrams is as follows: SGN1, SGN2, and OVLD are individual signal lines, while DI1 through DI10 are groups of four lines each. The signal lines of group DIx (in which x denotes one of the BCD characters 1 through 10) consist of DIx-8, DIx-4, DIx-2, and DIx-1; the 8, 4, 2, and 1 prefixes are used to denote the binary-weighted significance of each line.

Standard Format (Read One BCD Device)

Signal Name	SGN1	DI1	DI2	DI3	DI4	DI5	DI6	DI7	DI8		SGN2	DI9		OVLD	DI10	
Info.	Mant. Sign	MSD	◄—						—► LSD	Exp. Char.	Exp. Sign	Exp. Digit	Comma	0 = $\overline{OVLD}$ 8 = OVLD	Fn. Digit	Line-Feed
BCD Char. (Pos. True)	+1011 −1101	0000 thru 1111	◄—						—► 0000 thru 1111	1110	+1011 −1101	0000 thru 1111	1100	0000 1000	0000 thru 1111	1010
ASCII Char.	+ −	X	X	X	X	X	X	X	X	E	+ −	X	,	0 or 8	X	LF

Let's take a closer look at how data are entered into the computer with a BASIC-language ENTER statement while using the Standard format. (Standard format is selected when the Peripheral Status Switch marked "OF" is in the "ON" position; further details will be given in the subsequent configuration section.) Suppose the following logic signals are present on the lines from the peripheral device:

Signal Name	SGN1	DI1-8	DI1-4	DI1-2	DI1-1	DI2-8	DI2-4	DI2-2	DI2-1	DI3-8	DI3-4	DI3-2	DI3-1	DI4-8	DI4-4	DI4-2	DI4-1	DI5-8	DI5-4	DI5-2	DI5-1	DI6-8	DI6-4	DI6-2	DI6-1	DI7-8	DI7-4	DI7-2	DI7-1	DI8-8	DI8-4	DI8-2	DI8-1	SGN2	DI9-8	DI9-4	DI9-2	DI9-1	OVLD	DI10-8	DI10-4	DI10-2	DI10-1
Logic Level	1	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0	0	1	0	0	1	0	0	0	1	0
BCD Character	−	1				2				3				4				5				6				7				8				+	9				0	2			

Number = −1.2345678E+16
Function = 2

Let's further assume the following: the Peripheral Status Switch settings are DATA = ON, SGN1 = ON, SGN2 = ON, OVLD = ON; and the following ENTER statement has been executed (with the BCD Interface as the source):

```
ENTER Bcd;Number,Function
```

The ENTER statement is executed as follows. The computer first initiates a handshake with the CTLA signal (handshake operation is also described in the configuration section). The peripheral responds to the request by placing data on the lines and then completing the handshake. The states of all data lines are now stored in registers on the interface (i.e, the data signals are "latched").

The Standard-format driver reads the state of the SGN1 line and generates an ASCII "+" character. The "number builder" routine of the free-field ENTER statement (described in Chapter 5) is used to construct the number as characters are entered for the variable Number.

The BCD digits DI1 through DI8 are then read and used to form the mantissa. The "E" character is generated automatically by the driver, after which it reads the SGN2 line and generates a " – " character. BCD digit DI9 is read; the driver generates a "3" for the exponent character. A comma is automatically generated by the driver, terminating entry into Number. The number builder then constructs the internal representation of – 1.2345678E + 16, which is placed in Number.

Since one additional numeric variable has been specified in the ENTER statement, the computer continues to enter characters from the interface. The OVLD signal line is read, and a "0" is generated and entered. BCD digit DI10 is read, and the resultant ASCII "2" is entered by the number builder. The driver automatically generates the line-feed character, which terminates both entry of characters into the Function variable and the ENTER statement. The variable Function is assigned a value of 2, and the ENTER has finished execution. Further examples of sending and receiving data through the BCD Interface are given in the main section of this chapter.

Optional Format

With the Optional format, **two peripherals** may be connected to the interface. One four-digit and one five-digit mantissa are generated with this format. The signal lines are organized as follows with Optional format:

Optional Format (Read Two BCD Devices)

Signal Name	First Device (FD)						Second Device (SD)									
	SGN1	DI4	DI2	DI6	DI8		SGN2	DI10	DI1	DI5	DI3	DI7		OVLD	DI9	
Info.	Mant. Sign	MSD	←——→		LSD	Comma	Mant. Sign	MSD	←——→		LSD	Exp. Char.	FD	SD	Line Feed
BCD Char. (Pos. True)	+ 1011 1101	0000 thru 1111	←——→		0000 thru 1111	1100	+ 1011 – 1101	0000 thru 1111	←——→		0000 thru 1111	1110	0000 thru 1000	0000 thru 1000	1010
ASCII Char	+ –	X	X	X	X	,	+ –	X	X	X	X	X	E	0 or 8	0 or 8	LF

Let's take a closer look at how data are entered into the computer by a BASIC-language ENTER statement while using the Optional format ("$\overline{OF}$" = OFF). Suppose the following logic signals are present on the lines from the peripheral device:

Signal Name	SGN1	DI4-8 DI4-4 DI4-2 DI4-1	DI2-8 DI2-4 DI2-2 DI2-1	DI6-8 DI6-4 DI6-2 DI6-1	DI8-8 DI8-4 DI8-2 DI8-1	SGN2	DI10-8 DI10-4 DI10-2 DI10-1	DI1-8 DI1-4 DI1-2 DI1-1	DI5-8 DI5-4 DI5-2 DI5-1	DI3-8 DI3-4 DI3-2 DI3-1	DI7-8 DI7-4 DI7-2 DI7-1	OVLD	DI9-8 DI9-4 DI9-2 DI9-1
Logic Level	1	0 1 0 0	0 0 1 0	0 1 1 0	1 0 0 0	0	0 0 0 0	0 0 0 1	0 1 0 1	0 0 1 1	0 1 1 1	1	0 0 0 1
BCD Character	–	4	2	6	8	+	0	1	5	3	7	8	1

Number_1 = – 4268
Number_2 = 1.537E + 84

Let's further assume that the Peripheral Status Switches are set as follows: DATA = ON, SGN1 = ON, SGN2 = ON, OVLD = ON; and that the following ENTER statement has been executed (with the BCD Interface as the source):

```
ENTER Bcd;Number_1,Number_2,
```

The computer initiates a handshake with both peripherals by using the $\overline{\text{CTLA}}$ and $\overline{\text{CTLB}}$ signals (handshake operation is described in the configuration section). Both peripherals must respond to the request by placing data on the lines and then completing the handshake. The states of all data lines from the first device are now stored in registers on the interface (i.e, the data signals are "latched").

As with Standard format, the Optional-format driver reads the states of the signal lines from the peripheral and generates the appropriate ASCII characters. The computer uses the "number builder" routine of the free-field ENTER statement (described in Chapter 5) to enter the ASCII characters from the interface and to generate the internal representation of the number(s) represented by the BCD characters.

In this example, the logic state of SGN1 (1, or True) is read by the driver, which generates a " − " character (see table). The BCD digits DI4, DI2, DI6, and DI8 are read, and corresponding characters are generated. The comma (generated by the driver) terminates entry into the first numeric variable, called Number_1. In this case, the value assigned to Number_1 is − 4268.

Since another number has been specified in the ENTER statement, the computer continues to enter characters through the interface until the line-feed is entered. A value of 1.537E + 84 is assigned to the variable Number_2. The line-feed character (also generated by the driver) terminates both entry of characters into Number_2 and the ENTER statement. Further examples of entering data through this interface are given in in the main section of this chapter.

The Binary Data Representation
A binary data representation is available on the HP 98623 BCD Interface. With this representation, the forty data lines (groups DI1 through DI10) are treated as five individual data bytes which can be entered using ENTER or STATUS statement(s).

The Binary Mode
Unlike the BCD representation, the Binary Mode has no Standard and Optional format; thus, the setting of the Option Format switch has no effect while in the Binary Mode.

To select the Binary Mode, write a non-zero numeric value into BCD Control register 3; the following statement shows a typical method.

```
CONTROL 11,3;1
```

To see how the ENTER statement enters data through the BCD Interface while in Binary Mode, let's suppose the logic signals on the data lines are as follows.

Signal Name	DI1-8 DI1-4 DI1-2 DI1-1 DI2-8 DI2-4 DI2-2 DI2-1	DI3-8 DI3-4 DI3-2 DI3-1 DI4-8 DI4-4 DI4-2 DI4-1	DI5-8 DI5-4 DI5-2 DI5-1 DI6-8 DI6-4 DI6-2 DI6-1	DI7-8 DI7-4 DI7-2 DI7-1 DI8-8 DI8-4 DI8-2 DI8-1	DI9-8 DI9-4 DI9-2 DI9-1 DI10-8 DI10-4 DI10-2 DI10-1
Logic Level	0 0 1 1 0 0 0 1	0 0 1 1 0 0 1 0	0 0 1 1 0 0 1 1	0 1 0 0 0 1 0 1	0 0 1 1 0 1 0 1
Decimal Value	49	50	51	69	53
ASCII Character	1	2	3	E	5

Let's make the same assumptions that have been made in the previous examples: the logic sense of the data lines is positive-true (the "DATA" switch is set to "ON"). Assume that the following ENTER statement has been executed.

```
ENTER Bcd USING "B";Byte1,Byte2,Byte3,Byte4,Byte5
```

The Control signal line ($\overline{CTLA}$) is placed in the Set state by the computer to signal to the peripheral that a data transfer is to take place. The peripheral responds on the Data Flag line ($\overline{DFLGA}$), completing the handshake and clocking ("latching") the data on the lines into interface registers.

The Binary-Mode driver begins reading the line states as bytes in the order DI1 through DI10; the first byte contains DI1 as the most significant bits and DI2 as the least significant bits. The second byte contains DI3 and DI4, and so forth. In this case, the values 49, 50, 51, 69, and 53 are given to the variables Byte1 through Byte5, respectively.

In this example, the "B" image is used to direct the computer to enter the data on the input signal lines as bytes. A line-feed character is generated by the driver to terminate the ENTER statement.

As another example, suppose that the data on the input lines and the switch settings are as in the preceding example. Let's look at how the computer would enter the data with the following statement.

```
ENTER Bcd;Number
```

As in the preceding example, the ENTER statement latches the data into the interface registers with the same handshake. The Binary-Mode driver begins reading the line states as bytes in the order DI1 through DI10; the first byte contains DI1 as the most significant bits and DI2 as the least significant bits. The second byte contains DI3 and DI4, and so forth. In this case, the characters "123E5" are entered, followed by a line-feed generated by the driver. In this case, the variable Number receives a value of 1.23E + 7.

Alternate Methods of Entering Data

As with other interfaces, the data signal lines' logic states can be read with STATUS statements. However, no handshake is performed with this method of entering data.

With the BCD Interface, STATUS registers 5 through 9 contain digits DI1 through DI10, and STATUS register 4 contains SGN1, SGN2, and OVLD. Examples are given in the main section of this chapter.

Outputting Data

Data may be output through the BCD Interface by using the OUTPUT statement. Data are sent through the eight output lines in byte-serial fashion. The eight lines are called DO-7 through DO-0, in which DO-7 is the most significant bit. Numeric data are sent with the most significant digits first; string data are sent with the lowest-subscripted string characters sent first. Representation depends on whether FORMAT ON or FORMAT OFF is in effect.

Let's look at how data are output through the BCD Interface with the following OUTPUT statement.

```
OUTPUT 11;"A2C"
```

With OUTPUT, each byte is transferred under control of a handshake which is identical to a corresponding ENTER handshake. The Binary-Mode driver does not send four-bit BCD digits, it sends entire bytes of data; so the driver does not perform any ASCII-to-BCD translation. The items specified in the OUTPUT list are evaluated and sent to the BCD Interface byte-serially. The following diagram shows the logic signals that appear on the Data Output signal lines:

ASCII/Char.	Decimal Value	DO-7	DO-6	DO-5	DO-4	DO-3	DO-2	DO-1	DO-0
A	65	0	1	0	0	0	0	0	1
2	50	0	0	1	1	0	0	1	0
C	67	0	1	0	0	0	0	1	1
C_R	13	0	0	0	0	1	1	0	1
L_F	10	0	0	0	0	1	0	1	0

Notice that the free-field convention is used, since the free-field form of the OUTPUT statement was used. The CR-LF (default) EOL sequence is sent after all items have been output. The same data may be sent with the following statement.

```
OUTPUT 11 USING "#,B";65,50,67,13,10
```

Other examples are given in the main section of the chapter.

Configuring the Interface

This section describes the range of or recommended interface's switch settings for use with BASIC language. The switch locations are described in the *HP 98623 BCD Interface Installation Note*.

Determining Interface Configuration

If the interface is already installed in a computer which currently has the BASIC-language system resident, you can determine the configuration by running the following program. If the interface is not yet installed, you may want to check the switch settings as you read this section to see that they are set for use with your particular application.

```
100    PRINTER IS 1
110    PRINT CHR$(12)      ! Clear screen.
120    !
130    DISP "Enter select code of BCD Interface."
140    ENTER 2;Isc
150    DISP
160    !
170    ON ERROR GOTO Skip_status ! Skip if bad isc.
180    STATUS Isc;Id
190 Skip_status:    OFF ERROR
200    !
210    PRINT "The Interface at select code ";Isc;
220    IF Id=4 THEN
230        PRINT "is a BCD Interface."
240    ELSE
250        PRINT "is NOT a BCD Interface."
260        PRINT "Program terminated."
270        STOP
280    END IF
290    PRINT
300    !
310    CONTROL Isc;1        ! Reset interface.
320    !
330    STATUS Isc,1;Intr_status
340    Mask=2 5+2^4         ! Mask out all but bits 5 and 4.
350    Bits_set=BINAND(Intr_status,Mask)
360    Hd_prior=(Bits_set MOD 16)+3  ! Shift Rt. and add 3.
370    PRINT "Hardware priority (Interrupt Level) is ";Hd_prior;"."
380    PRINT
390    !
400    STATUS Isc,3;Binary_mode
410    IF Binary_mode THEN
420        PRINT "Binary mode selected."
430    ELSE
440        STATUS Isc,4;Switches
450        IF BIT(Switches,7)=1 THEN
460            PRINT "BCD mode, Optional format selected (2 devices),"
470        ELSE
480            PRINT "BCD mode, Standard format selected (1 device),"
490        END IF
500    END IF
510    PRINT
520    !
530    PRINT "Logic sense of signals:"
540    IF BIT(Switches,6)=1 THEN
550        PRINT "     Input data:   Low=1, High=0."
560    ELSE
570        PRINT "     Input data:   Low=0, High=1."
580    END IF
590    !
600    IF BIT(Switches,5)=1 THEN
610        PRINT "     SGN1:   High=""+"", Low=""-""."
620    ELSE
```

```
630        PRINT "     SGN1:   High=""-"", Low=""+"","
640    END IF
650    !
660    IF BIT(Switches,4)=1 THEN
670        PRINT "     SGN2:   High=""+"", Low=""-"","
680    ELSE
690        PRINT "     SGN2:   High=""-"", Low=""+"","
700    END IF
710    !
720    IF BIT(Switches,3)=1 THEN
730        PRINT "     OVLD:   High=0, Low=8,"
740    ELSE
750        PRINT "     OVLD:   High=8, Low=0,"
760    END IF
770    PRINT
780    !
790    END
```

Setting the Interface Select Code

The interface's select code setting determines the value of the interface select code parameter in which is used in ENTER and OUTPUT statements to specify the interface through which data is to be sent. The allowable range is 8 through 31, since internal interfaces already use select codes 1 through 7. Keep in mind that **no two interfaces should be set to the same select code.**

The default select code is 11. If a different select code is desired, set the switches as described in the installation note.

Setting the Hardware Priority (Interrupt Level)

The hardware priority assigned to an interface determines the order in which the interrupts from the interface are logged by the system. The software priority of interrupts determines the order of interrupt service, which is independent of this hardware priority.

A default setting of 3 is generally used. See the installation note for switch location and settings.

Setting the Peripheral Status Switches

The peripheral status switches are used to select the format of BCD data and the logic sense of data input lines. The $\overline{OF}$ switch selects between the Optional BCD format and the Standard BCD format. Set the switch to ON (default) if Standard is desired, or to OFF if Optional format is desired. The setting of this switch is irrelevant if the interface is only to be used in the Binary mode.

The **DATA** switch determines the logic sense of all 40 data input lines. If set to ON, positive-true logic is used (logic high is a 1). If set to OFF, negative-true logic is used; (logic low is a 1).

The **SGN1** and **SGN2** switches determine the logic sense of the respective sign-bit signal lines. If set to ON, a logic high signifies a " − " and logic low signifies a " + ". If set to OFF, a logic high signifies a " + " and logic low signifies a " − ".

The **OVLD** switch determines the logic sense of the OVLD signal line. If set to ON, a logic high is entered as an "8" and a low is entered as a "0". If set to OFF, a logic high is entered as a "0" and low is an "8".

Setting the Handshake Configuration

The handshake used by the BCD Interface is a two-wire handshake that synchronizes the exchange of data in one of two general manners: Type 1 timing or Type 2 timing. Type 1 and Type 2 timing differ in when the peripheral's data are clocked (latched) into the interface's data registers.

The logic sense of both the Control lines from the computer ($\overline{\text{CTLA}}$ and $\overline{\text{CTLB}}$) and Data Flag lines from the peripheral ($\overline{\text{DFLGA}}$ and $\overline{\text{DFLGB}}$) are switch-selectable.

Type 1 Timing

With Type 1 handshake timing, the Busy-to-Ready transition of the peripheral's data flag line ($\overline{\text{DFLGA}}$ or $\overline{\text{DFLGB}}$) Clears the Control line ($\overline{\text{CTLA}}$ or $\overline{\text{CTLB}}$) from the computer and clocks the data into the interface's Data In registers. The following timing diagram shows an example of how this sequence of events takes place. Note that the CTLA and DFLGA switches are set to OFF (Low-true).
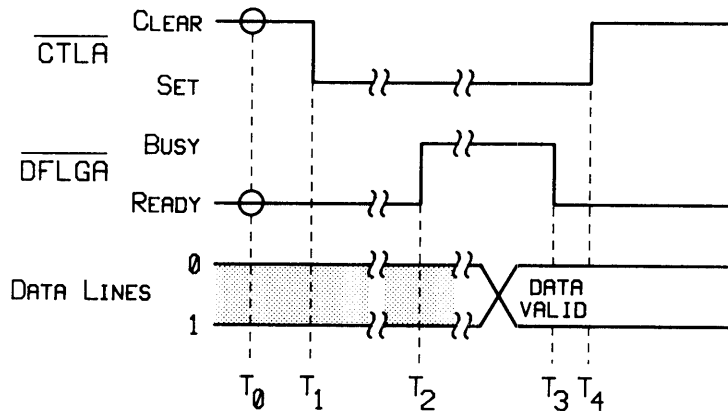
Type 1 Handshake Timing Diagram

At time t0, $\overline{\text{CTLA}}$ is Clear and $\overline{\text{DFLGA}}$ is Ready, indicating that a transfer may be initiated. At time t1, the computer initiates the handshake. At t2, the peripheral responds by placing $\overline{\text{DFLGA}}$ Busy. The peripheral then places the data on the data lines. When data have settled, the peripheral completes the handshake by placing $\overline{\text{DFLGA}}$ Ready, which also Clears $\overline{\text{CTLA}}$ and clocks the data into the interface registers (at time t4). Another handshake cycle may then be initiated by the computer.

Note

If only one peripheral is connected to the interface, connect the $\overline{\text{CTLB}}$ line to the $\overline{\text{DFLGB}}$ line and set both CTLB and DBLGB switches to the OFF positions. If this is not done, the handshake cannot be completed.

Type 2 Timing

With Type 2 handshake timing, the Ready-to-Busy transition of the peripheral's data flag line (DFLGA or DFLGB) Clears the Control line from the computer; however, the Busy-to-Ready transition still clocks the data into the interface's Data In registers. The following timing diagram shows an example of how this sequence of events takes place. Note that the CTLA and DFLGA switches are set to OFF (Low-true).
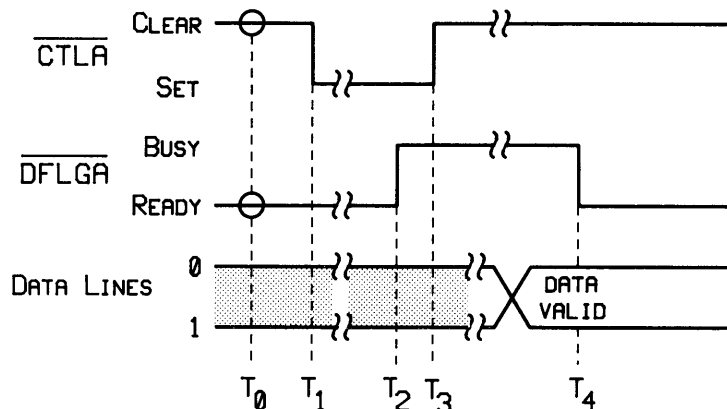
Type 2 Handshake Timing Diagram

At time t0, $\overline{\text{CTLA}}$ is Clear and $\overline{\text{DFLGA}}$ is Ready, indicating that a transfer may be initiated. At time t1, the computer initiates the handshake. At t2, the peripheral responds by placing $\overline{\text{DFLGA}}$ Busy, which also Clears $\overline{\text{CTLA}}$. When ready, the peripheral places $\overline{\text{DFLGA}}$ Ready (at time t4), which also clocks the data into the interface registers. Another handshake cycle may then be initiated by the computer.

Note

If only one peripheral is connected to the interface, connect the $\overline{\text{CTLB}}$ line to the $\overline{\text{DFLGB}}$ line and set both CTLB and DFLGB switches to the OFF positions. If this is not done, the handshake cannot be completed.

Configuring the Cable

The installation note describes how to connect the cable wires. Any unused lines should be connected as follows: connect the line to the "+5 Ref" signal line if the line is to be read as high, or to logic ground if the line is to be read as low. With lines such as SGN1, SGN2, and OVLD the line may be tied either to ground or to +5V, because the logic-sense switch allows either sense to be selected independent of other signals.

Note

Be sure to follow the recommendations in the installation note **exactly** to ensure signal integrity and operator safety.

Interface Reset

The interface should always be reset to ensure that it will be in a known state before use. All interfaces are automatically reset by the computer at certain times: when the computer is powered on, when the (RESET) key is pressed, and at other times described in the Reset Table (in the Useful Tables). The interface may also be reset by BASIC programs, as in the following examples.

```
Bcd=11
CONTROL Bcd;1


Reset_value=1
CONTROL Bcd,0;Reset_value


RESET Bcd
```

The following action is take when the BCD Interface is reset:

- The peripheral reset signal line ($\overline{\text{PRESET}}$) is pulsed low for at least 15 microseconds.
- The $\overline{\text{CTLA}}$ and $\overline{\text{CTLB}}$ handshake lines are Cleared.
- The Data Out register is cleared (set to all 0's).
- The Interrupt Enable bit is cleared, disabling subsequent interrupts until re-enabled by the program, and the Interrupt Request bit is set.

The state of the BCD/Binary Mode register (STATUS and CONTROL Register 3) is unchanged by the Interface Reset.

Entering Data Through the BCD Interface

This section describes BASIC programming techniques useful for entering data through the BCD Interface. Several examples of entering data were given in the first section to show how the interface works in BCD Mode with Standard and Optional formats and Binary Mode. This section gives additional general techniques for entering data from peripheral devices. If you need further explanation of how the ENTER statement works, refer to Chapter 5; Chapter 6 discusses the STATUS statement.

The diagrams and corresponding BASIC-language statements in this section show how data on the interface signal lines get read by the ENTER statement and corresponding values assigned to BASIC-language variable(s). The notation used in the examples is that the name of the interface signal line (or group of lines) is shown above the decimal value and ASCII character that the driver produced by reading the line(s). The logic sense of the lines is not shown here; see the preceding configuration section for a description of selecting the logic sense of the interface signals.

As an example, the following drawing shows that an ASCII " + " was generated by the driver when it read the SGN1 signal line; similarly, the four signals of the group DI5 produced a period character. The driver produces the "E" (exponent), comma, and line-feed characters automatically.

SGN1	DI1	DI2	DI3	DI4	DI5	DI6	DI7	DI8		SGN2	DI9		OVLD	DI10	
+	1	2	3	4	.	6	7	8	E	−	6	,	0	4	LF

The following statements show how the preceding data might be entered and the resultant values assigned to program variables.

```
Bcd=11
ENTER Bcd;Number,Function
PRINT "Number=   ";Number
PRINT "Function= ";Function
```

The following display is the result of executing the preceding statements.

```
Number=   1.234678E-3
Function= 4
```

Entering Data from One Peripheral

There are several methods of entering data through the BCD Interface when only one peripheral device is connected. The Standard BCD format can be used with many devices; the Binary mode must be used with others, and some require that you write your own "drivers."

Entering with BCD-Mode Standard Format

Using the Standard format of BCD Mode usually provides the most convenient means of entering data from one device. This format allows up to 8 BCD digits for mantissa and one BCD digit for exponent. The state of an overload-indicator signal and one optional BCD digit can also be entered, if desired.

SGN1	DI1	DI2	DI3	DI4	DI5	DI6	DI7	DI8		SGN2	DI9		OVLD	DI10	
+	0	.	3	4	5	6	0	0	E	+	4	.	8	4	LF

```
100   ENTER 11;Number,Function
110   PRINT "Number= ";Number
120   IF Function>=80 THEN
130      PRINT "Overload of function ";Function-80
140   ELSE
150      PRINT "Function= ";Function

160   END IF
```

The following results would be printed by the preceding program segment:

```
Number=  3456
Overload of function  4
```

The ENTER statement calls the Standard-format driver, which reads the BCD characters on the interface lines in the order shown and generates the approptiate ASCII characters. Characters are entered until the ",", is read, which terminates entry into the variable Number. The characters after the comma are used to build the value of Function. The ENTER statement is properly terminated when the line-feed (an ENTER-statement terminator) is encountered.

Notice that an "8" is generated by the driver when the OVLD line is true. The BASIC program must "separate" this from the "function" digit (DI10). The method shown in the example is only one of many methods available.

If a second variable would not have been included in the preceding ENTER statement, ENTER would have continued asking the driver for characters until it encountered the line-feed, which terminates the statement.

To contrast the preceding example, suppose that the following statement has been executed:

```
ENTER 11 USING "#,K";Number
```

In this case, the # specifier directs the ENTER statement to suppress its default requirement of looking for a line-feed character (or other statement-termination condition) to terminate the ENTER. Thus, the comma terminates both entry of data into Number and the ENTER statement. Consequently, a subsequent ENTER statement would begin entering characters beginning with the "8" character (OVLD), which may not be the desired action.

In such a case, several remedies are possible. The simplest is probably to go ahead and include a second variable so that the driver is left pointing to the first character after the ENTER is completed. The second variable is thus used for a "dummy" read operation. Another remedy is to write a non-zero value to BCD CONTROL register 1, which "resets" the driver pointer to the first character of the format (SGN1). Executing the following statement performs the driver reset.

```
CONTROL 11,1;1
```

This type of "problem" may also occur when the BCD device sends a line-feed as one of the BCD characters.

SGN1	DI1	DI2	DI3	DI4	DI5	DI6	DI7	DI8	SGN2	DI9	OVLD	DI10
—	1	2	LF	4	5	6	7	LF	E	+	0	,	0	0	LF

In such case, two numbers are sent separated by line-feeds. The following statements would read these two numeric values and then reset the driver pointer to the first character (the SGN1 character).

```
ENTER 11;Number_1,Number_2
CONTROL 11,1;1
```

If the CONTROL statement had not been executed, the driver would have been left pointing to the "E" character.

As another example, suppose the exponent is to be ignored but the overload and function digits are to be read. The following statement would be appropriate in such a situation.

```
ENTER 11;Number_1,Number_2,Dummy,Function
```

The variable Dummy is so named to show that it is included in the ENTER statement only to ensure that the overload and function digits are read and assigned to a variable (i.e., it is not used for any other purposes). Of course, the value could be used if desired.

If your application requires reading only certain characters or groups of characters, you may want to read Chapter 5 to see more examples of using images with ENTER statements.

Entering with Binary Mode

If your application represents data with eight-bit ASCII characters or has a data format that is not compatible with the Standard BCD format, the Binary Mode can be used. With the Binary Mode, data are entered in groups of eight bits each, rather than in groups of four-bit BCD digits. Five bytes are latched with each handshake; the driver reads the bytes sequentially until the fifth byte is read, after which it sends a line-feed character to terminate the ENTER. Another handshake operation is required if more data are to be entered.

As an example, let's assume that the following logic signals are present on the interface lines. Only 16 signals are shown here because that is all that we will be using for this example.

Signal Name	DI1-8	DI1-4	DI1-2	DI1-1	DI2-8	DI2-4	DI2-2	DI2-1	DI3-8	DI3-4	DI3-2	DI3-1	DI4-8	DI4-4	DI4-2	DI4-1
Logic Level	0	1	0	0	0	0	0	1	0	0	1	1	0	0	0	1
Decimal Value	65								49							
ASCII Character	A								1							

Assume also that the I/O path name "@Bcd" is assigned to the select code of a BCD Interface. The following ENTER statement enters these two bytes of data as numbers in the range 0 through 255.

```
ENTER @Bcd USING "B";Di1_di2,Di3_di4
```

The "B" specifier directs the computer to enter one byte of data from the interface and place it into the corresponding numeric variable, which happens two times in this case. The variables Di1_di2 and Di3_di4 receive values of 65 and 49, respectively. The ENTER statement continues to request characters from the Binary-Mode driver until a line-feed (generated by the driver) is returned, which terminates the ENTER statement. Even though only two bytes were used to fill variables in this example, all five bytes of data were read from the interface.

The following statement could be used to enter the two bytes as one 16-bit word.

```
ENTER @Bcd USING "W";Word1
```

The variable "Word1" receives a value of 16 689 (= 256*65 + 49).

As another example, suppose that the data on the lines are to be interpreted as ASCII characters. The following diagrams show the ASCII representations of data read from the interface; entering ten bytes of data in this mode requires two handhake cycles.

DI1&DI2	DI3&DI4	DI5&DI6	DI7&DI8	DI9&DI10	
1	2	3	4	5	LF

DI1&DI2	DI3&DI4	DI5&DI6	DI7&DI8	DI9&DI10	
6	7	8	E	3	LF

The following ENTER statement enters characters until an item terminator is found and then calls the "number builder" routine to construct the number; this sequence is performed for each numeric variable in the statement.

```
ENTER Bcd;Number_1,Number_2
```

In this case, Number_1 is assigned a value of 12345, and Number_2 is assigned 6.78E+5. With a Binary-Mode ENTER, the driver does not read SGN1, SGN2, or OVLD, and does not insert any E's, or commas; only a line-feed is generated as a sixth character to terminate the ENTER statement.

If your application has a data format that is not compatible with the Standard BCD format, the Binary Mode can be used in conjunction with a routine of your own design that is tailored for your application's requirements. Let's look at an example of how this might be accomplished.

Suppose that your peripheral requires five digits of mantissa but must have three exponent digits and two function digits. A program will be used to read the data using the desired format. If the peripheral's handshake method is compatible with one of the handshake types available on the BCD Interface, the Binary mode may be used to enter the data; if not, see the example of implementing a handshake in the next section.

For this example, suppose the following conditions exist: the mantissa is entered from DI1 through DI5, the exponent is entered from DI6 through DI8, and function is entered from DI9 and DI10. The mantissa and exponent signs and the overload indicator are still available as individual signal lines, but they must be read with the STATUS statement.

The subroutine shown in the following program reads the data on the lines with ENTER and STATUS statements and then formats the data as required for the application. The formatted information is then entered from a string variable into the desired numeric variables.

```
100    ! This program executes a subroutine which enters data from the
110    ! BCD Interface using Binary mode and formats it as follows:
120    !
130    ! SGN1   DI1   DI2   DI3   DI4   DI5   E   SGN2   DI6   DI7   DI8   ,
140    ! OVLD   DI9   DI10   LF
150    !
160    ! Define ordering of BCD characters.
170    Bcd_chars$="0123456789+,-E."
180    !
190    Bcd=11
200    CONTROL Bcd,3;1   ! Set Binary mode.
210    !
220    GOSUB New_format                  ! Execute subroutine.
230    ENTER Format$;Number,Function    ! Use results for ENTER.
240    PRINT "Number=";Number
250    PRINT "Function=";Function
260    !
270    STOP
280    !
290 New_format: ! ******* Beginning of  Subroutine. *******
300                !
310    ! Perform a Binary-mode ENTER.
320    ENTER Bcd USING "5A";Bytes$ ! 5 bytes read.
330    !
340    ! Use STATUS to read SGN1, SGN2, OVLD.
350    STATUS Bcd,4;Sgns_and_ovld
360    !
370    ! Generate two ASCII characters from each byte.
380    FOR Byte=1 TO 5
390        !
400        ! Get numeric value of single byte from Bytes$.
410        Char=NUM(Bytes$[Byte])
420        !
430        ! Upper 4 bits form first ASCII char.
440        Up_4_bits=Char DIV 16 ! Shift right 4 places.
450        ! Use numeric value as index into Bcd_chars$.
460        First_char$=Bcd_chars$[Up_4_bits+1;1]
470        !
480        ! Lower 4 bits form 2nd ASCII char.
490        Lo_4_bits=Char MOD 16 ! Mask upper 4 bits.
500        ! Use numeric value as index into Bcd_chars$.
510        Second_char$=Bcd_chars$[Lo_4_bits+1;1]
520        !
530        ! Now append characters onto format string.
540        Digits$[2*Byte-1]=First_char$&Second_char$
550        !
560    NEXT Byte
570    !
580    !
590    ! Calc. SGN1's and SGN2's ASCII representations.
600    Sgn1=BIT(Sgns_and_ovld,2)
610    Sgn1$=CHR$(43+2*Sgn1) ! "+" if Lo; "-" if Hi.
620    Sgn2=BIT(Sgns_and_ovld,1)
630    Sgn2$=CHR$(43+2*Sgn2) ! "+" if Lo; "-" if Hi.
640    !
650    ! Calc. Overload's ASCII representation.
660    Ovld=BIT(Sgns_and_ovld,0)
670    Ovld$=CHR$(48+8*Ovld) ! "0" if Lo; "8" if Hi.
680    !
690    Number$=Sgn1$&Digits$[1,5]&"E"&Sgn2$&Digits$[6,8]
700    Function$=Ovld$&Digits$[9,10]
710    Format$=Number$&","&Function$&""
720    !
730    RETURN ! ******* End of Subroutine. *******
740    !
750    END
```

Entering with STATUS Statements

The preceding examples assumed that the handshake options available with the BCD Interface
are compatible with your peripheral. This section shows examples of designing enter operations
using STATUS and CONTROL statements to implement your own handshakes.

```
100     Bcd=11
110     CALL Enter_bytes(Bcd,Bytes$)
120     PRINT Bytes$
130     STOP
140     !
150     END
160     !
170     SUB Enter_bytes(Isc,Return_string$)
180         !
190         CONTROL Isc,2;1  ! Initiate handshake.
200         !
210 Check: STATUS Isc,1;Intr_stat
220         Irq=BIT(Intr_stat,6)
230         IF NOT Irq THEN Check   ! Wait for response.
240         !
250         ! Now read bytes in registers 5 -> 9.
260         STATUS 11,5;R5,R6,R7,R8,R9
270         !
280         ! Return bytes as a string.
290         Return_string$=CHR$(R5)&CHR$(R6)&CHR$(R7)
300         Return_string$=Return_string$&CHR$(R8)&CHR$(R9)
310         !
320     SUBEND
```

Note that the program uses the Interrupt Request bit (bit 6 of register 1) to determine when the
handshake is completed by the peripheral. This bit is cleared (0) when a Request is performed
(i.e., when the handshake is initiated by writing a non-zero value to CONTROL register 2). The
bit is set when the peripheral acknowledges the Control ($\overline{\text{CTLA/B}}$) signal by responding with
Data Flag ($\overline{\text{DFLGA/B}}$). The acknowledgement occurs when the Control line is Cleared by the
leading edge of Data Flag (Type 2 timing) or by its trailing edge (Type 1 timing).

The transfer of data can also be implemented with interrupts, which is described in the BCD
Interrupts section.

Entering Data from Two Peripherals

Data can be entered from two devices by using either BCD-Mode Optional format or by using STATUS statements. Optional format allows up to 4 BCD digits from the first peripheral and up to 5 BCD digits from the second peripheral; overload from either device may also be detected. Data from each device is handshaked independently.

Optional Format

This section describes how to use the Optional format with BASIC programs. In order to use this format, the peripheral's handshake convention must be compatible with one of the handshake options available on the BCD Interface. Since the preceding section described how to implement handshake routines with STATUS and CONTROL statements, you should refer to that discussion if your application requires that type of solution.

With the BCD-Mode Optional format, the data, sign, and overload signals are read and formatted into ASCII characters in the following sequence:

SGN1	DI4	DI2	DI6	DI8		SGN2	DI10	DI1	DI5	DI3	DI7		OVLD	DI9	
+	4	2	6	8	,	−	0	1	5	3	7	E	0	0	LF

The following program segment shows an example of how these characters might be entered, stored in variables, and printed.

```
100    ENTER  11;Number_1,Number_2
110    PRINT  "Number  1=  ";Number_1
120    PRINT  "Number  2=  ";Number_2
```

The following results would be printed by the preceding program segment:

```
Number  1=   4265
Number  2=  -1537
```

The ENTER statement calls the Optional-format driver, which reads the signals on the interface lines in the order shown and generates the approptiate ASCII characters. Characters are entered and sent to the "number builder" until the "," is read, which terminates entry into the variable Number_1; the internal representation of the numeric value is then generated. The characters after the comma are used to build the value of Number_2. The ENTER statement is properly terminated when the line-feed (an ENTER-statement terminator) is encountered.

If a second variable would not have been included in the preceding ENTER statement, ENTER would have continued asking the driver for characters until it encountered the line-feed, which terminates the statement.

It is important to note that an "8" is generated by the driver when the OVLD line is true or when **any** of the bits of DI9 are true, making the possibilities of exponent values 0, 8, 80, or 88; consequently, the BASIC program must "separate" these overload indicators.

Separating overload information from the mantissa may be a problem when one number can be represented in two ways; for instance, ".0001E8" and "10000" both represent the value "1.0E + 4", but the two representations have entirely different meanings. The first representation indicates an overload on the second device, while the second value does not.

To solve this potential problem, the second number can be entered into a string variable, as shown in the following segment.

```
100    ENTER 11;Number_1,Number_2$
110    !
120    ! Separate 2nd mantissa and exponent.
130    Exponent$=Number_2$[8,9]
140    Number_2$=Number_2$[1,6]
150    !
160    ! Place 2nd mantissa in numeric variable.
170    ENTER Number_2$;Number_2
180    !
190    PRINT "Number 1=";Number_1
200    ! Check overload information.
210    IF Exponent$[1;1]="8" THEN
220        PRINT "Overload on device 1."
230        PRINT
240    END IF
250    !
260    PRINT "Number 2=";Number_2
270    ! Check overload information.
280    IF Exponent$[2]="8" THEN
290        PRINT "Overload on device 2."
300        PRINT
310    END IF
320    PRINT
330    !
340    END
```

The program checks the exponent digits separately and indicates an overflow on either device.

To contrast the preceding examples, suppose that the following statement has been executed:

```
ENTER 11 USING "#,K";Number
```

In this case, the # specifier directs the ENTER statement to suppress its default requirement of looking for a line-feed character (or other statement-termination condition) to terminate the ENTER. Thus, the comma terminates both entry of data into Number and the ENTER statement. Consequently, a subsequent ENTER statement would begin entering characters beginning with the character following the comma (i.e., the first character of the second number), which may not be the desired action.

In such a case, several remedies are possible. The simplest is probably to go ahead and include a second variable so that the driver is left pointing to the first character after the ENTER is completed. The second variable is thus used for a "dummy" read operation. Another remedy is to write a non-zero value to BCD CONTROL register 1, which "resets" the driver pointer to the first character of the format (SGN1). Executing the following statement performs the driver reset.

```
CONTROL 11,1;1
```

This type of situation may also occur when the BCD device sends a line-feed as one of the BCD characters.

SGN1	DI4	DI2	DI6	DI8		SGN2	DI10	DI1	DI5	DI3	DI7		DI9	OVLD	
+	1	2	3	LF	,	+	0	1	5	3	LF	E	0	0	LF

In such case, two numbers are sent separated by line-feeds. The following statements would read these two numeric values and then reset the driver pointer to the first character (the SGN1 character).

```
ENTER 11;Number_1,Number_2
CONTROL 11,1;1
```

If the CONTROL statement had not been executed, the driver would have been left pointing to the "E" character.

Outputting Data Through the BCD Interface

All data outputs through the BCD Interface are made through the eight output lines. There are two general methods of sending data to devices through the BCD Interface — by using CONTROL statements and by using OUTPUT statements. With CONTROL statements, the data are latched on the output lines, but the handshake (if desired) must be performed with STATUS and CONTROL statements. With the OUTPUT statement, each data byte is sent individually under handshake control. With both methods, neither the setting of the Optional Format switch nor the current Mode (BCD or Binary) has any effect on how data are output through the interface.

Output Routines Using CONTROL and STATUS

Many applications do not require that data be sent with a handshake operation. In such cases, the following example shows how one byte of data may be sent to the peripheral.

```
100    Byte=2^6+2^4        ! Set Bits 6 and 4.
110    CONTROL 12,4;Byte  ! Send data w/o handshake.
```

If your application requires a handshake which is not compatible with the handshake options available on the BCD Interface, you can program your own. The following program shows an example handshake. The transition of the Data Flag signal that Clears the Control signals is still determined by the setting of the CTLA-2 and CTLB-2 switches. See the configuration section for further details.

```
100    Bcd=11
110    Chars$="1A2B"
120    Eol$=CHR$(10) ! LF is EOL sequence.
130    CALL Output_bcd(Bcd,Chars$,Eol$)
140    !
150    END
160    !
170    SUB Output_bcd(Isc,Characters$,Eol$)
180         !
190         Output_data$=Characters$&Eol$
200         FOR I=1 TO LEN(Output_data$)
210             CONTROL Isc,2;1 ! Initiate handshake.
220             !
230             ! Now output byte(s) to registers 4.
240             CONTROL Isc,4;NUM(Output_data$[I;1])
250             !
260             ! See if Ready for next byte.
270 Check:      STATUS Isc,1;Intr_stat
280             Irq=BIT(Intr_stat,6)
290             IF NOT Irq THEN Check  ! Wait for response.
300             !
310         NEXT I
320         !
330    SUBEND
```

The data are output on the Data Output lines in byte-serial fashion. The program uses the Interrupt Request indicator (Bit 6 of STATUS Register 1) to indicate the interface's and peripheral's joint readiness for a subsequent handshake operation. Interrupts can also be used; for more details, see the discussion of BCD Interrupts.

Sending Data with OUTPUT

With the OUTPUT statement, data are output byte-serially, one byte per handshake cycle. The following program shows an example of outputting data through the BCD Interface.

```
100    Bcd=11
110    OUTPUT Bcd;123,456,"ABC","DEF"       .
120    OUTPUT Bcd;123,456;"ABC";"DEF"
130    OUTPUT Bcd;"123","456";
140    !
150    END
```

The following diagram shows the sequence of ASCII characters sent to the destination device with the preceding program. The notation indicates that each ASCII character is sent through the output lines DO-7 through DO-0.

1	2	3	,	4	5	6	,	A	B	C	CR	LF	D	E	F	EOL Characters

1	2	3	4	5	6	A	B	C	D	E	F	EOL Characters

1	2	3	CR	LF	4	5	6
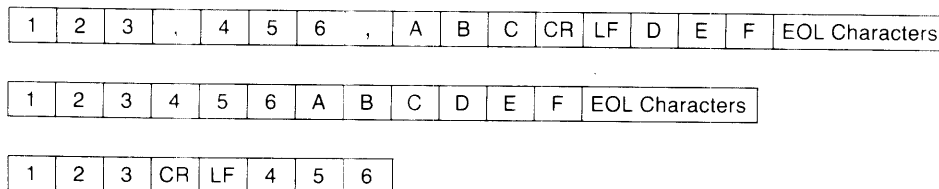
Notice that when a comma follows an output item in a free-field OUTPUT statement, a numeric item in the output data is terminated by a comma and a string item is terminated by a CR/LF sequence (one carriage-return and one line-feed character). If an item is followed by a semicolon, no item terminator is sent. If an item is the last one in the output list, an end-of-line (EOL) sequence is sent instead of the item terminator; the default EOL sequence is a CR/LF with no time delay. Changing the EOL sequence is described in Chapter 4.

In the preceding program, the FORMAT ON attribute was in effect so the ASCII representation of each data item was generated and sent to the peripheral device. It is also possible to OUTPUT with FORMAT OFF in effect by using I/O path names. See Chapter 10 for further details.

It is interesting to note that all handshake cycles latch **both** input and output data. In the following example, an OUTPUT statement is used to place one byte on the Data Out lines under handshake control. A STATUS statement is then used to read the Data In lines, since the handskake operation also latched the data on the input lines into STATUS Registers.

```
100    Byte=64+32                 ! Set bits 6 and 5.
110    OUTPUT 11 USING "#,B";Byte  ! Handshake byte 1 out.
120    ! Now read SGN1, SGN2, OVLD, and DI1 thru DI10.
130    STATUS 11,4;Reg4,Reg5,Reg6,Reg7,Reg8,Reg9
140    Sgn1=BIT(Reg4,2)
150    Sgn2=BIT(Reg4,1)
160    Ovld=BIT(Reg4,0)
170    Di1=Reg5 DIV 16
180    Di2=Reg5 MOD 16
190    Di3=Reg6 DIV 16
200    Di4=Reg6 MOD 16
          .
          .
          .
```

The program determines the states of the sign, overload, and data lines. The data may then be formatted as desired.

BCD Interface Timeouts

When a peripheral device does not respond to a handshake request from the computer, it is convenient to be able to sense this condition and respond accordingly. Using the ON TIMEOUT statement sets up and enables a branch which will be initiated when the computer determines that the interface has taken too much time to respond.

Timeout events were generally discussed in Chapter 7. However, specific details such as the effects of the TIMEOUT event's occurrence on each interface and how the time parameter is measured were not described. This section describes such topics.

Timeout Time Parameter
When an ON TIMEOUT is set up for an interface, the time required to complete each handshake is measured and compared to the time specified in the ON TIMEOUT statement. The interval measured is shown in the following diagram.
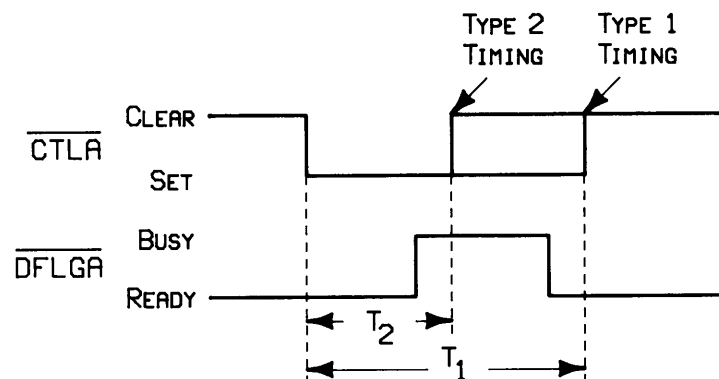
Measuring the BCD Interface's TIMEOUT Parameter

Timing begins when the $\overline{CTLA}$ and $\overline{CTLB}$ signals are placed in the Set state to initate a handshake cycle. The computer continues to check the time elapsed against the specified time (TIMEOUT time parameter). Timing ends when the peripheral has completed its response; with both Type 1 and Type 2 timing, this occurs **only** when the Control line is cleared **and** the Data Flag line is placed in the Ready state by the peripheral.

Timeout Service Routines
When a TIMEOUT occurs, the computer automatically executes an **Interface Reset**. The Peripheral Reset line to the peripheral (Preset) is pulsed low for at least 15 microseconds, and $\overline{CTLA}$ and $\overline{CTLB}$ are then Cleared. This action should "get the peripheral's attention", if it is functional. The service routine should then take the appropriate corrective action. See a previous section called "Interface Reset" for further effects of the reset.

Timeout service routines generally determine whether or not the peripheral is still functional. If so, the computer may take corrective action such as to re-initiate the preceding transfer. If not, perhaps the program may inform the operator of the condition and then proceed.

The following program shows an example of setting up a branch to a service routine upon detecting a TIMEOUT on the BCD Interface. When a TIMEOUT occurs while trying to send the first message, the service routine attempts to send an escape character to the peripheral, which here is a request for status of our fictitious peripheral. If the peripheral does not respond, the destination of data is changed to the CRT.

```
100    Bcd=11    ! Interface select code of BCD.
110    Dest=Bcd  ! Destination is device is BCD.
120    ON TIMEOUT Bcd,2 GOSUB Try_bcd_again
130    !
140    Message$="This sent to BCD."
150    OUTPUT Dest;Message$
160    ! If TIMEOUT, this line is executed upon RETURN.
170    !
180    ! All subsequent data sent to Dest=CRT (if TIMEOUT).
190    Message$="This sent to CRT."
200    OUTPUT Dest;Message$
210    !
220    STOP
230    !
240 Try_bcd_again: ON TIMEOUT Bcd,3 GOTO Forget_it
250                !
260                ! See if escape character is accepted.
270                OUTPUT Bcd USING "#,B";27
280                ! If accepted, then 2nd TIMEOUT didn't occur;
290                ! so this segment might contain a routine
300                ! that interrogates peripheral.
310                !
320                ON TIMEOUT Bcd,3 GOSUB Try_bcd_again
330                GOTO Exit_point
340                !
350 Forget_it:     PRINT "BCD Down; Data will be sent to CRT."
360                PRINT
370                Dest=1
380                BEEP
390                OFF TIMEOUT Bcd ! No longer need active TIMEOUT.
400                !
410 Exit_point:    RETURN ! to line following TIMEOUT's occurrence.
420                !
430    END
```

The timeout service routine may be programmed to attempt to continue the transfer where it timed out; however, this action may be difficult to implement for two reasons: the computer may not be keeping track of where in the transfer the TIMEOUT occurred, and the automatic Interface Reset performed when the TIMEOUT occured may have also reset the peripheral. How your program implements the transfer and how the peripheral respond to the reset will determine the feasibility of continuing the transfer.

BCD Interface Interrupts

The BCD Interface can detect one type of interrupt condition: an interrupt can be generated when the interface is Ready for a subsequent data transfer, which generally occurs after the program initiates a handshake and the peripheral completes it.

Setting Up and Enabling Interrupts

When an event occurs, the event is logged by the BASIC operating system. After the event is logged, any further interrupts from the interface are disabled until specifically re-enabled by a program. All further computer responses to the event depend entirely on the program.

The following segment shows a typical sequence of setting up and enabling a BCD interrupt to initiate its branch.

```
100    ON INTR 11 GOSUB Bcd_intr
110    Mask=1
120    ENABLE INTR 11;Mask
```

The value of the interrupt mask (Mask in the program) determines whether the interrupt is to be enabled or disabled. In this case, any non-zero value enables the Ready interrupt.

Interrupt Service Routines

Since there is only one type of interrupt possible with the BCD Interface, the service routine does not need to determine the interrupt cause. In general, all the service routine needs to do is to determine whether another data item is to be transferred or the transfer is to be terminated. The following program shows a typical interrupt service routine.

```
100    Bcd=11
110    ON INTR Bcd GOSUB Get_bytes
120    !
130    CONTROL Bcd,2;1    ! Initiate 1st handshake.
140    ENABLE INTR Bcd;1 ! Enable Ready Interrupts.
150    !
160    ! Execute background routine.
170    WHILE Iteration<1.E+6
180        Iteration=Iteration+1
190        DISP Iteration
200    END WHILE
210    !
220 Get_bytes:!
230            STATUS Bcd,5;Reg5,Reg6,Reg7,Reg8,Reg9
240            PRINT Reg5,Reg6,Reg7,Reg8,Reg9
250            CONTROL Bcd,2;1 ! Initiate next handshake.
260            ENABLE INTR Bcd ! Re-enable (use same Mask).
270        RETURN
280        !
290    END
```

The main program sets up the branch location, initiates the first data-transfer handshake, and then enables the interface to interrupt when it is Ready; the peripheral is Ready when it has cleared the Control line(s) and placed the Data Flag line(s) in the Ready state.

The service routine reads the data on lines DI1 through DI10, initiates the subsequent handshake, and then re-enables another Ready interrupt. since the mask parameter was not included, the last specified value (1) was used.

Obviously, this is a very simplistic example; however, the main topics of using interrupts have been shown. Your routine may need to format the data, keep track of how many bytes have been transferred, and check for terminator characters.

Summary of
BCD Status and Control Registers

Status Register 0   — Card Identification = 4.

Control Register 0   — Reset Interface (if non-zero value sent).

Status Register 1

Interrupt Status

Most Significant Bit

Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Interrupt Are Enabled	Interrupt Request	Hardware Priority (INT LVL Switches)		0	0	0	0
Value = 128	Value = 64	Value = 32	Value = 16	Value = 0	Value = 0	Value = 0	Value = 0

Control Register 1   — Reset driver pointer (if non-zero value sent).

Status Register 2

Busy Bit

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	Interface Is Busy
Value = 0	Value = 0	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Bit 0 is 1 when a handshake is currently in progress.

Control Register 2   — Request data by Setting CTLA and CTLB (if a non-zero value is sent); this operation also clears an Interrupt Request (clears bit 6 of Status Register 1).

Status Register 3   — Binary Mode: 1 if the interface is currently operating in Binary mode, and 0 if in BCD mode.

Control Register 3   — Set Binary Mode: set Binary Mode if non-zero value sent, and BCD Mode if zero sent.

Status Register 4

Most Significant Bit   Switch and Line States

Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OF Switch Is ON	DATA Switch Is ON	SGN1 Switch Is ON	SGN2 Switch Is ON	OVLD Switch Is ON	SGN1 Input Is True	SGN2 Input Is True	OVLD Input Is True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Control Register 4

Most Significant Bit   Data Out Lines

Least Significant Bit

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

Status Register 5

Most Significant Bit   BCD Digits D1 and D2

Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DI1-8 Is True	DI1-4 Is True	DI1-2 Is True	DI1-1 Is True	DI2-8 Is True	DI2-4 Is True	DI2-2 Is True	DI2-1 Is True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Status Register 6

Most Significant Bit   BCD Digits D3 and D4

Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DI3-8 Is True	DI3-4 Is True	DI3-2 Is True	DI3-1 Is True	DI4-8 Is True	DI4-4 Is True	DI4-2 Is True	DI4-1 Is True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Status Register 7   BCD Digits D5 and D6

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DI5-8 Is True	DI5-4 Is True	DI5-2 Is True	DI5-1 Is True	DI6-8 Is True	DI6-4 Is True	DI6-2 Is True	DI6-1 Is True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Status Register 8   BCD Digits D7 and D8

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DI7-8 Is True	DI7-4 Is True	DI7-2 Is True	DI7-1 Is True	DI8-8 Is True	DI8-4 Is True	DI8-2 Is True	DI8-1 Is True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Status Register 9   BCD Digits D9 and D10

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DI9-8 Is True	DI9-4 Is True	DI9-2 Is True	DI9-1 Is True	DI10-8 Is True	DI10-4 Is True	DI10-2 Is True	DI10-1 Is True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Summary of BCD READIO and WRITEIO Registers

This section describes the BCD Interface's READIO and WRITEIO registers. Keep in mind that these registers should be used **only** when an operation cannot be performed with a STATUS or CONTROL statement.

BCD READIO Registers

Register 1 — Card Identification

Register 3 — Interface Status

Register 17 — DI1 and DI2

Register 19 — DI3 and DI4

Register 21 — DI5 and DI6

Register 23 — DI7 and DI8

Register 25 — DI9 and DI10

Register 27 — Peripheral Status

READIO Register 1   **Card Identification**

The contents of this register are always 4.

READIO Register 3   **Interrupt Status**

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Interrupt Are Enabled	Interrupt Request	Hardware Priority (INT LVL Switches)		0	0	0	0
Value = 128	Value = 64	Value = 32	Value = 16	Value = 0	Value = 0	Value = 0	Value = 0

READIO Register 17   **DI1 and DI2**

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DI1-8 Is True	DI1-4 Is True	DI1-2 Is True	DI1-1 Is True	DI2-8 Is True	DI2-4 Is True	DI2-2 Is True	DI2-1 Is True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

READIO Register 19 DI3 and DI4

Most Significant Bit | | | | | | | Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DI3-8 Is True	DI3-4 Is True	DI3-2 Is True	DI3-1 Is True	DI4-8 Is True	DI4-4 Is True	DI4-2 Is True	DI4-1 Is True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

READIO Register 21 DI5 and DI6

Most Significant Bit | | | | | | | Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DI5-8 Is True	DI5-4 Is True	DI5-2 Is True	DI5-1 Is True	DI6-8 Is True	DI6-4 Is True	DI6-2 Is True	DI6-1 Is True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

READIO Register 23 DI7 and DI8

Most Significant Bit | | | | | | | Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DI7-8 Is True	DI7-4 Is True	DI7-2 Is True	DI7-1 Is True	DI8-8 Is True	DI8-4 Is True	DI8-2 Is True	DI8-1 Is True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

READIO Register 25 DI9 and DI10

Most Significant Bit | | | | | | | Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DI9-8 Is True	DI9-4 Is True	DI9-2 Is True	DI9-1 Is True	DI10-8 Is True	DI10-4 Is True	DI10-2 Is True	DI10-1 Is True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

READIO Register 27   **Switch and Line States**

Most Significant Bit   Least Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OF Switch Is ON	DATA Switch Is ON	SGN1 Switch Is ON	SGN2 Switch Is ON	OVLD Switch Is ON	SGN1 Input Is True	SGN2 Input Is True	OVLD Input Is True
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

BCD WRITEIO Registers

Register 1 — Reset Interface

Register 3 — Enable Interrupt

Register 5 — Output Data

Register 7 — Initiate Handshake

WRITEIO Register 1   — Reset interface (any value causes reset).

WRITEIO Register 3   — Enable interrupt if Bit 7 Set (1); disable if Bit 7 Clear (0).

WRITEIO Register 5   **Set Data Output Lines**

Most Significant Bit   Least Significant Bit

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

WRITEIO Register 7   — Initiate handshake: sending any value to this register initiates handshake cycle by setting CTLA and CTLB.

.

EPROM Programming	Chapter
	18

Introduction

With HP Series 200 BASIC, erasable programmable read-only memory (EPROM) devices are generally used like other mass storage devices. However, EPROM can also be accessed as individual bytes or words of data. This chapter describes both types of usage.

Accessories Required

In order to program and read EPROM memory devices with HP Series 200 computers, you will need the following HP accessories:

- HP 98253 EPROM Programmer card
- HP 98255 EPROM Memory card(s) and compatible EPROM devices

Hardware Installation

At this point, you should install the programmer and memory cards or verify that they have already been properly installed. The following manuals describe setting up your system to program EPROM devices.

- *HP 98253 EPROM Programmer Installation* — describes setting the select code switches on the programmer card and installing the card.
- *HP 98255 EPROM Memory Installation* — describes selecting compatible EPROM parts, loading the parts on the card, setting memory address switches, and installing EPROM memory cards.

The first example program in the chapter describes how to interrogate EPROM Programmer and Memory cards to determine their current configurations (and also to determine whether or not both are operational *before* installing EPROMs in the memory boards).

Brief Overview of Using EPROM Memory

EPROM memory is organized and accessed like other mass storage devices from BASIC. Briefly, programs and data can be stored in EPROM memory with the following procedure:

1. Determine the EPROM Programmer card's select code. Determine the address of the EPROM Memory card to be programmed, relative to other cards' addresses, which determines its mass storage unit number.

2. INITIALIZE the memory unit, which writes directory and system information in the EPROM (see "EPROM Directories" for further information).

3. Store the information using whichever one of the following statements is appropriate:

    CONTROL — store individual data words
    COPY — store any type of file
    SAVE — store the program as an ASCII file
    STORE — store the program as a PROG file
    STORE BIN — store a binary program in a BIN file
    STORE KEY — store typing-aid keys in a KEY file

4. Access the information with the corresponding one of the following statements:

    CAT — get a catalog listing of the files in the EPROM unit
    COPY — copy an EPROM file's contents into another file
    ENTER — enter data from a file into a program variable
    GET — load an ASCII program file into the computer
    LOAD — load a BIN, KEY, or PROG file into the computer
    LOADSUB — load SUB or FN subprograms from a PROG file
    TRANSFER — transfer data from data file to a memory BUFFER
    STATUS — read individual data words from EPROM memory

Initializing EPROM Memory

Like other mass storage media, EPROM media must be initialized before being used for storage. Since EPROM Memory cards are organized as mass storage *units*, each card being *one unit*, EPROM memory must be initialized by units. The EPROM Programmer card is used to initialize and store other information in EPROM. This section describes how to specify EPROM units and programmer cards while initializing and accessing EPROM.

EPROM Programmer Select Code

The EPROM Programmer card is accessed like other interface cards: you must specify its *select code* in BASIC statements. The factory default setting of the select code is 27, which is the select code assumed in the examples in this section. (Setting the select code is described in the *HP 98253 EPROM Programmer Installation* manual.) As further explained later, you don't usually need to specify the programmer card's select code when reading data from EPROM.

EPROM Addresses and Unit Numbers

With the BASIC system, EPROM Memory cards should be given memory addresses 300 000 through 3FF FFF (hexadecimal). The address switches on EPROM Memory cards can therefore be set in the range of 0011000 through 0011111, which result in hexadecimal base addresses of 300 000 through 3E0 000, respectively, with intervals of 20 000 bytes (hex) between base addresses. When using addresses in this range, SW2 must be set to the "AD" position. (Note that differences between base addresses of cards containing 27128 EPROMs must be at least 40 000 hexadecimal. See the *HP 98255 EPROM Memory Installation* manual for further explanation.)

At power-up, the system automatically gives unit numbers to all cards according to the initialized cards' *relative* memory addresses. The card with the lowest numbered address (which is initialized) is given unit number 0; the initialized card with the next higher address is given unit number 1, and so forth. (Note that un-initialized EPROM units are not given unit numbers by the system.)

As an example, suppose that two EPROM Memory cards are properly installed in the computer with hexadecimal base addresses of 300 000 and 380 000. Assume that they have already been initialized. At power-up, the former card will be given unit number 0 and the latter will be given unit number 1.

If an initialized card with base address 340 000 is then installed (with power off, of course), this card is given unit number 1 and the card at address 380 000 is given unit number 2 at power-up. (Note that, like disc media, the unit number is *not* written on the media. Unit numbers are a function of relative EPROM addresses only.)

It is a good idea to keep track of the addresses of all EPROM Memory cards in the system so that you will know the resultant unit number of each card.

Verifying Hardware Operation

In order to INITIALIZE an EPROM unit, you will need to connect a programmer card to it. Connect the cable from the desired programmer card to the EPROM unit to be programmed; the power need not be turned off to make this connection. *All* EPROM devices on the unit to be initialized *must be completely erased.* Also, the address of the EPROM card to be initialized must be higher than all other initialized EPROM cards in the system, which results in the card being given a unit number one greater than the largest unit number currently in the system.

If you have been keeping track of memory addresses, you should know the unit number of EPROM Memory card to be programmed. If not, you can use the following program to determine the address of each EPROM Memory card in the computer by plugging the connector into each memory card in succession.

```
100     ! This program interrogates interfaces at select codes
110     ! 8 thru 31 to find an EPROM Programmer card. If one IS found,
120     ! the program reads and displays its STATUS registers; if one
130     ! is NOT found, the program reports this negative result.
140     !
150     ! Clear screen.
160     PRINT CHR$(12)
170     !
180     Sel_code=8                      ! Start with select code 8.
190     Found_card=0
200     ON ERROR GOTO Next_sel_code   ! Goto next select code if
210                                     ! no interface at this one.
220     REPEAT
230        STATUS Sel_code;Id
240        IF Id=27 THEN
250           Found_card=1
260           PRINT "EPROM Programmer card found at Select Code";Sel_code
270           PRINT
280        END IF
290 Next_sel_code: IF NOT Found_card THEN Sel_code=Sel_code+1
300     UNTIL Found_card=1 OR Sel_code>=31
310     OFF ERROR
320     !
330     IF Found_card=0 THEN
340        PRINT "EPROM Programmer card not found."
350        PRINT "Program stopped."
360        STOP
370     END IF
380     !
390     ! Check to see if connected to memory card.
400     STATUS Sel_code,4;Capacity
410     IF Capacity=0 THEN
420        PRINT "EPROM Programmer is NOT connected ";
430        PRINT "to an EPROM Memory card"
440        STOP
450     END IF
460     !
470     ! Read STATUS Registers 0 thru 6.
480     STATUS Sel_code;Reg0,Reg1,Reg2,Reg3,Reg4,Reg5,Reg6
490     !
500     ! Show register contents.
510     PRINT "STATUS Register 0:"
520     PRINT "  Card ID of EPROM Programmer card=";Id
530     !
540     PRINT "STATUS Register 6:"
550     PRINT USING "#,K,8D";" Connected to EPROM card at address ";Reg6
560     Msb_hex$=IVAL$(Reg6/65536,16)                    ! Get MSB's in hex.
570     PRINT " (";Msb_hex$[3,4];"0 000 hexadecimal)" ! Trim leading 0's.
580     !
590     PRINT "STATUS Register 4:"
```

```
600    PRINT "  Memory card size=";Reg4;" bytes";
610    Msb_hex$=IVAL$(Reg4/65536,16)                    ! Get MSB's in hex.
620    PRINT "  (";Msb_hex$[3,4];"0 000 hex)"           ! Trim leading 0's.
630    !
640    PRINT "STATUS Register 5:"
650    PRINT "  Number of contiguous, erased bytes=";Reg5;
660    Msb_hex$=IVAL$(Reg4/65536,16)                    ! Get MSB's in hex.
670    PRINT "  (";Msb_hex$[3,4];"0 000 hex)"           ! Trim leading 0's.
680    !
690    PRINT "STATUS Register 2:"
700    PRINT "  Current target address=";Reg2
710    !
720    PRINT "STATUS Register 3:"
730    Word$=IVAL$(Reg3,16)
740    PRINT "  Word at current target address=";Reg3;" (";Word$;" hex)"
750    !
760    PRINT "STATUS Register 1:"
770    IF Reg1=0 THEN
780      PRINT "  Programming time = 52.5 ms"
790    ELSE
800      PRINT "  Programming time = 13.1 ms"
810    END IF
820    !
830    END
```

The following display is a typical result of running the program.

```
EPROM Programmer card found at Select Code 27


STATUS Register 0:
Card ID of EPROM Programmer card= 27
STATUS Register 6:
Connected to EPROM card at address  3145728 (300 000 hexadecimal)
STATUS Register 4:
Memory card size= 262144  bytes (040 000 hexadecimal)
STATUS Register 5:
Number of contiguous, erased bytes= 0
STATUS Register 2:
Current target address= 0
STATUS Register 3:
Word at current target address= -1   (FFFF hex)
STATUS Register 1:
Programming time = 52.5 ms
```

The program interrogates interfaces until it finds an EPROM Programmer card. It then prints the values of the Programmer card's STATUS registers. Register 6 shows the memory address of the EPROM Memory card to which the programmer card is connected. The program also shows that it can determine the type of EPROM devices being used on the card (2764's or 27128's).

The "target address" register points to the memory location (an offset address to the card's base address) at which the next word of data will be read (STATUS register 3) or written (CONTROL register 3). Target address 0 is the first word on the EPROM card. STATUS register 1 indicates which programming time will be used (for each word) during subsequent programming operations; 0 indicates a program time of 52.5 milliseconds, and 1 indicates 13.1 milliseconds.

Initializing Units

To INITIALIZE an EPROM unit, you must specify the select code of the EPROM Programmer card and the unit number of the EPROM Memory card. For instance, the following statement initializes the memory with unit number 0 through the programmer card at select code 27.

```
INITIALIZE ":EPROM,27,0"
```

Because the unit number defaults to 0 if not specified, an equivalent statement would be:

```
INITIALIZE ":EPROM,27"
```

An error is reported if the specified programmer card is not connected to the specified EPROM unit. Furthermore, if the specified EPROM memory unit is not completely erased, error 66 (INITIALIZE Failed) is reported. Note that the entire card need not be filled with EPROMs for it to appear as entirely erased, since empty sockets and erased EPROM memory read as "FF" data bytes. The following simple program determines whether or not the EPROM unit contains all erased EPROMs (or erased EPROMs and empty sockets).

```
10      CONTROL 27,2;0   ! Set target address to first byte,
20      STATUS 27,4;Total_capacity,Erased_bytes
30      PRINT "EPROM card is ";
40      IF Total_capacity=Erased_bytes THEN
50        PRINT "completely erased (or empty),"
60      ELSE
70        PRINT "NOT completely erased,"
80      END IF
90      END
```

EPROM Directories

The INITIALIZE operation writes a directory and system information in the EPROM unit. This information occupies the first "sectors" of EPROM memory (since the unit is treated like mass storage, it is logically divided into 256-byte records known as sectors). The following table shows how the BASIC system allocates EPROM sectors.

EPROM Type	Usable Sectors	Sectors for System Use	Sectors for User	Maximum No. of Files
2764	511	0 – 6	7 – 510	40
27128	1023	0 – 11	12 – 1022	80

Note that the figures given for Total Sectors and Sectors for User are for *fully loaded* memory cards. Note also that the Total Sectors is one less than you may have expected, which shows that one sector is required by the system for overhead.

EPROM Catalogs

Performing a CAT of the EPROM card reveals that it has been initialized. You can either specify the select code of the programmer card or use 0, since reading the EPROM card does not require the programmer card be connected to it. However, if you do specify a select code, then that programmer card must be connected to the specified EPROM unit, or error 72 will be reported. The following statements perform the same function (specifying select code 27 would change the first line of the catalog listing accordingly):

```
CAT ":EPROM,0"
        or
CAT ":EPROM,27"
```

```
:EPROM,0
VOLUME LABEL: B9836
FILE NAME PRO TYPE   REC/FILE BYTE/REC    ADDRESS
```

This directory has the same general format as internal-disc directories, which are described in Chapter 7, "Data Storage and Retrieval," of *BASIC Programming Techniques.* You can also perform all operations on EPROM directories that you you can with other mass storage directories (such as SKIP and COUNT files and CAT individual PROG files).

Programming EPROM

Once an EPROM unit is initialized, you can store data and programs in it. The following storage operations are supported for EPROM memory:

- CHECKREAD — direct the system whether to perform an additional verify operation after all operations that write to mass storage
- CONTROL — store individual data words in EPROM
- COPY — copy any type of file (that already exists on another mass storage device) into EPROM
- SAVE — store the current program in an ASCII file
- STORE — store the current program in a PROG file
- STORE BIN — store a binary program in a BIN file
- STORE KEY — store the current typing-aid keys in a KEY file

Using these statements is described in the following sections of this chapter. The topic of reading EPROM information is described in a subsequent section.

Storing Data

As a simple example of storing a data file in EPROM, suppose that you want to store the date that the EPROM was initialized and the current number of EPROM chips on the card in EPROM memory. The following program shows a simple example of how you might perform this operation.

```
100    ! This Program stores the Date that the
110    ! EPROM Memory unit was initialized.
120    ! (An EPROM file name shows the date.)
130    !
140    ! Select EPROM mass storage unit.
150    Progmr_sc=27
160    Unit_no=0
170    Eprom_msus$=":EPROM,"&VAL$(Progmr_sc)&","&VAL$(Unit_no)
180    !
190    ! Determine date to write in EPROM.
200    Correct_date=0
210    REPEAT
220      DISP "Enter date to be stored in EPROM ";
230      DISP "(Press ENTER for time shown)."
240      OUTPUT KBD;DATE$(TIMEDATE);
250      ENTER KBD;Date_$
260      SET TIMEDATE DATE(Date_$)! Set date.
270      DISP "Is this correct?  ";DATE$(TIMEDATE)
280      ENTER KBD;Ans$
290      IF UPC$(Ans$[1,1])="Y" THEN Correct_date=1
300    UNTIL Correct_date
310    DISP
320    !
330    ! Format Date_$ from "DD MMM YYYY"
340    ! to "MMM_DD_YY".
350    Month$=Date_$[4,6]
360    Day$=TRIM$(Date_$[1,2]) ! Strip leading space (if one).
370    Year$=Date_$[10,11] ! Remove "19" from year.
380    File_name$=Month$&"_"&Day$&"_"&Year$
390    !
400    ! Create a one-record ASCII file on the internal disc
410    ! (use an external disc with Model 16)
420    ! with the DATE as the file's name.
430    Disc_msus$=":INTERNAL"
440    CREATE ASCII File_name$&Disc_msus$,1   ! Error if file exists.
450    !
460    ! Write info into EPROM file.
470    COPY File_name$&Disc_msus$ TO File_name$&Eprom_msus$
480    PURGE File_name$&Disc_msus$ ! Remove disc file after use.
490    !
500    ! Now read date with catalog of file names.
510    CAT Eprom_msus$
520    !
530    END
```

The program first prompts for the EPROM unit number (the programmer card is assumed to be at select code 27). The program then prompts for the date by presenting the current clock date to the user on the keyboard input line. The user can either modify the date or accept it as it is shown.

Assuming that the program is run on a Model 226 or 236, the program stores this information in an ASCII file on a disc in the internal drive. (It would be much faster to store the file in a MEMORY volume or in Bubble memory.) This information is then stored in EPROM, and the internal ASCII file is purged.

Data Storage Rates

The information is stored in EPROM at a *approximately* the following rates (program time is set by writing to CONTROL register 1):

Program Time	Seconds per Sector	Bytes per Second
13.1 ms	2	150
52.5 ms	7	38

Note that these times are for COPY, SAVE, and STORE operations. The storage rate when using CONTROL is lower slightly than these figures.

Determining Unused EPROM Memory

A potential problem with the example program in the preceding section is that there are times when you are not sure whether or not there is enough erased EPROM memory to store your information. Unfortunately, the system generally cannot determine beforehand whether there is sufficient room left in an EPROM unit to store the information it has been directed to store. This consequence is due to the fact that both blank sockets and erased EPROM memory read as all 1's (hexadecimal FF bytes). The system can, however, determine when there is not enough room left on a *fully loaded* card (Error 64 is reported).

Thus, when the system is directed to store data in EPROM, it begins programming the EPROMs one word at a time at the "next available" location. After each word is programmed, the system reads the word and compares it to what was to be written; this operation is known as "verifying" the word. An error will be reported when the word is not verified (such as when a blank socket, a previously programmed word, or other hardware failure has been reached). So before you attempt to store any information in EPROM memory, you should determine whether or not there is enough erased memory to hold the data.

Then general method of determining whether or not an EPROM memory unit has enough erased space to store your information is as follows: Determine the total number of usable sectors on the EPROM card, and then subtract the number already used. The result is the number of sectors available for storing additional information. This procedure is broken down into steps as follows (an example follows the procedure):

1. Determine the number of usable sectors on the EPROM card.

   a. Determine the number of usable sectors of EPROM by using the following formulas:

   Total Sectors = (Chips on card)*(Bytes/Chip)*(1 Sector/256 Bytes)
   Usable Sectors = Total Sectors − 1 sector (used by the system)

   in which: Bytes/Chip = 16 384 (for 27128's)

   $$= 8\ 192\ (\text{for } 2764\text{'s})$$

   b. Use the CAT statement to determine how many EPROM sectors are already being used by files (already programmed).

2. Determine the number of sectors required to store your information.

   a. For ASCII data files, this number will simply be the number of records specified in the CREATE ASCII statement that created the file.

   b. For BDAT data files, multiply the number of records in the file by the record size (default = 256 bytes), divide this product by 256, and round any non-integer result to the next larger integer. Add one to this result to account for the sector used by the system (for EOF pointer and number of records).

   c. For programs, place the information in a mass storage file using STORE or SAVE (MEMORY volumes and BUBBLE memory are best suited for this purpose). Use the CAT statement to determine how many 256-byte sectors were required to store the information.

   d. For all other types of files, a quick look at the directory of the media on which the file is presently stored shows how many sectors are required to store the file.

3. Compare the amount of usable memory in EPROM to the amount of memory required for your information. If there is sufficient memory, perform the storage operation. Otherwise, use another EPROM unit or mass storage device.

As an example, suppose that you want to store a BDAT file that contains 20 records of 20 bytes each in EPROM. Since 20*20 = 400, and 400/256 = 1.5625 (which rounds up to the next larger integer of 2), two sectors of EPROM are required for the data. Add one sector for system use. Therefore, three sectors are required to store the file.

To determine how much EPROM memory is available, first calculate the total number of sectors on the card. For this example, suppose that only two 27128 chips are on the board. The total number of sectors on the card is calculated by applying the preceding formula:

Total sectors = 2 * 16 384 / 256 = 128

Usable sectors = Total sectors − 1 = 127

To see how many sectors have already been used, perform a CAT of the EPROM card; assume EPROM unit 0.

```
CAT ":EPROM,0"


:EPROM,0
VOLUME LABEL: B9836
FILE NAME PRO TYPE   REC/FILE BYTE/REC   ADDRESS


Mar_8_83        ASCII        1      256        12
EPROM_BITS      ASCII       17      256        13
EPROM_INIT      ASCII       11      256        30
```

The CAT reveals that the last file begins at sector 30 and is 11 sectors in length. Thus, the next unused sector begins at sector 41. Since sector addresses start at 0, 42 sectors have already been used. Assuming that you have not written any data in subsequent sectors (such as with the CONTROL statement), there are 85 (= 127 − 42) sectors of unused EPROM remaining. The BDAT file can be stored in the unit.

Storing Programs

Storing programs in EPROM is a simple operation. Like storing programs in other mass storage media, you can use either the STORE statement or the SAVE statement. The one you will use depends on whether you want the program to be stored in a PROG or an ASCII file; the STORE statement stores the program in a PROG file, while the SAVE statement stores it in an ASCII file. If the program is already stored on another device, use the COPY statement.

Compiled Pascal subprograms, or "CSUBs," can also be stored in EPROM with COPY. For instructions on how to write these compiled subprograms, see the *CSUB Utility* manual.

As with storing data files, you must ensure that there is enough memory on the card to hold the program. First execute a CAT operation on the EPROM unit to determine how many sectors are unused. Then determine how many sectors will be required to store the program by using STORE or SAVE to store the program on another mass storage device. If there is enough unused EPROM memory, execute STORE or SAVE with the destination as the desired EPROM unit. For instance, the following statements are typical ways to store programs in EPROM.

```
STORE "Prog_1:EPROM,27,0"
SAVE "Prog_1:EPROM,27"
```

Programming Individual Words and Bytes

You can also program individual words and bytes in EPROM with the BASIC system. However, you should *not* use these techniques to program EPROMs which are to be used as mass storage "units" in this manner. In other words, if you are going to access the EPROM with mass storage statements, use only operations such as SAVE, STORE, and COPY to program the EPROM unit. If the EPROM is to be for another purpose, such as to store machine-language code in another system, you can use these techniques to program the EPROMs.

To program individual words, use CONTROL to set the target address and then write the desired word at that address. Repeat this process for as many words as you need to write. Note that you need to set the target address before every write operation. If you don't, an error will be reported.

The automatic verify operation is still performed for each word written into EPROM memory. The following example program shows how you might perform this type of operation; the first 16 384 bytes of the EPROMs in sockets marked "0U" and "0L" are programmed. (The program takes approximately eight minutes to run.)

```
100   ! Assume data source is a BDAT file that contains exactly
110   ! 8192 INTEGER elements (written with FORMAT OFF).
120   !
130   ASSIGN @Source TO "EPROMWORDS:INTERNAL"
140   INTEGER Int_array(0:8191)
150   ENTER @Source;Int_array(*)
160   !
170   ! Write 8K words (16K bytes).
180   FOR Addr=0 TO 16382 STEP 2
190     CONTROL 27,2;Addr,Int_array(Addr/2)   ! Write to EVEN addresses.
200   NEXT Addr
210   !
220   END
```

Notice that the target address (CONTROL register 2) begins at an even address (0) and is incremented by two for each subsequent word. Attempting to program a word at an odd address will generate an error.

To program individual bytes, you will need to mask the byte that is not to be programmed. For instance, suppose that you want to insert one EPROM chip in the board and program it with data bytes. Inserting the chip in one of the sockets marked with an "L" gives the memory odd addresses. The program will need to be modified so that it writes only the least significant eight bits of each word (since you can only program words, which begin at even addresses).

To program only the least significant byte, you would make the most significant byte all 1's so that the program operation will verify (remember that empty sockets and erased bits read as all 1's). The following program shows an example of programming single bytes of data in the EPROM located in the socket marked "0L." Note that the only difference between this program and the previous one is the manipulation of the upper eight bits of each integer.

```
100    ! Assume data source is a BDAT file that contains exactly
110    ! 8192 INTEGER elements (written with FORMAT OFF).
120    !
130    ASSIGN @Source TO "EPROMBYTES:INTERNAL"
140    INTEGER Int_array(0:8191)
150    ENTER @Source;Int_array(*)
160    !
161    ! Define mask for upper 8 bits.
162    Ff00=IVAL("FF00",16)
163    !
170    ! Write 8K bytes.
180    FOR Addr=0 TO 16382 STEP 2   ! Must still write to EVEN addresses.
181       Low_byte=BINIOR(Ff00,Int_array(Addr/2))  ! MSB=FF (LSB=unchanged).
190       CONTROL 27,2;Addr,Low_byte
200    NEXT Addr
210    !
220    END
```

To program bytes of an EPROM located in a socket marked "U", you would left-shift the 8-bit value by eight places (which shifts the least significant byte to the most significant byte). For instance, the following statement shifts the least significant byte to the most significant byte and then makes the least significant byte all 1's:

```
High_byte=BINIOR(SHIFT(Low_byte,-8),255)
```

Programming EPROM with such eight-bit values writes the eight bits into EPROM devices at even addresses (i.e., in sockets marked with "U").

Operations Not Allowed

Once data is written in EPROM, it cannot be selectively erased (without erasing the entire EPROM device). Consequently, the following mass storage operations are not allowed for EPROM mass storage:

    CONTROL (cannot be used to write to registers 7 and 8 of an I/O path name assigned to a
            BDAT file)
    CREATE ASCII
    CREATE BDAT
    COPY (of an entire mass storage unit *into* EPROM)
    OUTPUT
    PROTECT
    PURGE
    RENAME
    RE-SAVE
    RE-STORE
    RE-STORE KEY
    TRANSFER (*to* an EPROM file)

Reading EPROM Memory

After an EPROM unit has been programmed, you can perform the following operations to read the information:

- CAT — get a catalog listing of the files in an EPROM unit
- COPY — copy an EPROM file's (or unit's) contents into another file (or unit)
- ENTER — enter data from an ASCII or BDAT data file into program variables
- GET — load an ASCII program file into the computer
- LOAD — load a BIN, KEY, or PROG file into the computer
- LOADSUB — load SUB or FN subprogram(s) from a PROG file
- TRANSFER — transfer data from a BDAT data file into a memory BUFFER
- STATUS — read individual data words from EPROM memory

Retrieving Data and Programs

Reading data files stored in EPROM is similar to reading data files stored in other mass storage devices; the only difference is the mass storage unit specifier (msus), which will be of the form :EPROM,Select_code,Unit_number. Remember that both Select_code and Unit_number parameters can be any type of numeric expression. Also keep in mind that when reading EPROM units you do not need to specify the select code of the EPROM programmer card; you can specify a select code of 0. However, if you do specify the programmer card's select code, it must be connected to the specified EPROM unit. If the unit number parameter is omitted, a default value of 0 is used.

The broad subject of using ENTER to read data files is discussed in the *BASIC Programming Techniques* manual. The *BASIC Interfacing Techniques* manual discusses the ENTER statement in detail. The interfacing manual also describes using the TRANSFER statement to transfer the contents of data files into memory BUFFERs.

Like reading data files, retrieving programs from EPROM is identical to performing these operations from other mass storage devices. Refer to the *BASIC Programming Techniques* manual.

Summary of EPROM Programmer STATUS and CONTROL Registers

STATUS Register 0   **ID Register**

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	1	1	0	1	1
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

This register contains a value of 27 (decimal) which is the ID of an EPROM Programmer card.

CONTROL Register 0 — Interface Reset
Writing any non-zero value into this register resets the card; writing a value of zero causes no action.

STATUS Register 1 — Read Program Time
A value of 0 indicates that the program time is 52.5 milliseconds for each 16-bit word (default); a non-zero value indicates that the program time is 13.1 milliseconds.

CONTROL Register 1 — Set Program Time
Writing a value of 0 into this register sets the program time to 52.5 milliseconds for each 16-bit word; any non-zero value sets program time to 13.1 milliseconds.

STATUS Register 2 — Read Target Address
This register contains the offset address (relative to the card's base address) at which the next word of data will be read (via STATUS Register 3) or written (via CONTROL Register 3). The default address is 0, which is the address of the first byte on the card.

CONTROL Register 2 — Set Target Address
Writing to this register sets the offset address at which the next word of data will be read (via STATUS Register 3) or written (via CONTROL Register 3). The target address must always be an *even* number.

Summary of EPROM Programmer
STATUS and CONTROL Registers (cont.)

STATUS Register 3 — Read Word at Target Address

This register contains the 16-bit word at the current target address.

CONTROL Register 3 — Write Word at Target Address

Writing a data word to this register programs a 16-bit word at the current target address. The target address must be set (via CONTROL register 2) before every word is written. Automatic verification is also performed after the word is programmed.

STATUS Register 4 — Current Memory Card Capacity (in bytes)

This register contains the current capacity of a *fully loaded* card in bytes; it also indirectly indicates which type of EPROM devices are being used on the card. If 262 144 is returned, then 27128 EPROMs are being used; if 131 072 is returned, then 2764 devices are being used. A 0 is returned if the programmer card is not currently connected to any EPROM memory card.

CONTROL Register 4 — Undefined.

STATUS Register 5 — Number of Contiguous, Erased Bytes

Reading this register causes the system to begin counting the number of subsequent bytes, beginning at the current target address, that are erased (or are empty sockets). The counting is stopped when a programmed byte (i.e., one containing at least one logical 0) is found or when the end of the card is reached. If the byte at the current target address is not FF, then a count of 0 is returned. Error 84 is reported if the programmer card is not currently connected to any EPROM card.

CONTROL Register 5 — Undefined.

STATUS Register 6 — Base Address of EPROM Memory Card

This register contains the (absolute) base address of the EPROM memory card to which the programmer card is currently connected; this base address is also the absolute address of the first word on the card. Error 84 is reported if the programmer card is not currently connected to any EPROM memory card.

CONTROL Register 6 — Undefined.

	Appendix
# Useful Tables	**A**

Option Numbers

1	BASIC Main	19	ERR
2	GRAPH	20	DISC
3	GRAPHX	21	CS80
4	IO	22	BUBBLE
5	BASIC Main	23	EPROM
6	TRANS	24	HP 9885
7	MAT	25	HPIB
8	PDEV	9	XREF
9	XREF	27	RS232
10	KBD	28	GPIO
11	CLOCK	29	BCD
12	LEX	30	DCOMM
13	BASIC Main	31-39	Reserved
14	MS	40	"PHYREC"
15	SRM	42	CRTB
16-18	Reserved	43	CRTA

Interface Select Codes

Internal Select Codes

1 Display (alpha)
2 Keyboard
3 Display (graphics)
4 Internal floppy-disc drive
5 Optional powerfail protection interface
6 Display (Graphics for bit mapped)
7 HP-IB interface (built-in)

Factory Presets for External Interfaces

8 HP-IB
9 RS-232
10 (not used)
11 BCD
12 GPIO
14 HP-IB Disc Interface
20 Data Communications
21 Shared Resource Management
27 EPROM Programmer
28 Color Output
30 Bubble Memory

Display-Enhancement Characters

Alpha Displays:

Character Code	Action Resulting from Displaying the Character
128	All enhancements off
129	Inverse mode on
130	Blinking mode on
131	Inverse and Blinking modes on
132	Underline mode on
133	Underline and Inverse modes on
134	Underline and Blinking modes on
135	Underline, Inverse, and Blinking modes on
136	White
137	Red
138	Yellow
139	Green
140	Cyan
141	Blue
142	Magenta
143	Black

Bit-Mapped Displays:

Character Code	Action Resulting from Displaying the Character
128	All enhancements off
129	Inverse mode on
130	No action
131	Inverse mode on
132	Underline mode on
133	Underline and Inverse modes on
134	Underline mode on
135	Underline and Inverse modes on
135	White ⎫
137	Red ⎪
138	Yellow ⎪
139	Green ⎬ Default color map with at least three color planes.
140	Cyan ⎪
141	Blue ⎪
142	Magenta ⎪
143	Black ⎭

US ASCII Character Codes

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

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

US ASCII Character Codes

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

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

European Display Characters

ASCII Char.	Dec	Binary
	0	00000000
	1	00000001
	2	00000010
	3	00000011
	4	00000100
	5	00000101
	6	00000110
	7	00000111
	8	00001000
	9	00001001
	10	00001010
	11	00001011
	12	00001100
	13	00001101
	14	00001110
	15	00001111
	16	00010000
	17	00010001
	18	00010010
	19	00010011
	20	00010100
	21	00010101
	22	00010110
	23	00010111
	24	00011000
	25	00011001
	26	00011010
	27	00011011
	28	00011100
	29	00011101
	30	00011110
	31	00011111

ASCII Char.	Dec	Binary
	32	00100000
!	33	00100001
"	34	00100010
#	35	00100011
$	36	00100100
%	37	00100101
&	38	00100110
'	39	00100111
(	40	00101000
)	41	00101001
*	42	00101010
+	43	00101011
,	44	00101100
-	45	00101101
.	46	00101110
/	47	00101111
0	48	00110000
1	49	00110001
2	50	00110010
3	51	00110011
4	52	00110100
5	53	00110101
6	54	00110110
7	55	00110111
8	56	00111000
9	57	00111001
:	58	00111010
;	59	00111011
<	60	00111100
=	61	00111101
>	62	00111110
?	63	00111111

ASCII Char.	Dec	Binary
@	64	01000000
A	65	01000001
B	66	01000010
C	67	01000011
D	68	01000100
E	69	01000101
F	70	01000110
G	71	01000111
H	72	01001000
I	73	01001001
J	74	01001010
K	75	01001011
L	76	01001100
M	77	01001101
N	78	01001110
O	79	01001111
P	80	01010000
Q	81	01010001
R	82	01010010
S	83	01010011
T	84	01010100
U	85	01010101
V	86	01010110
W	87	01010111
X	88	01011000
Y	89	01011001
Z	90	01011010
[	91	01011011
\	92	01011100
]	93	01011101
^	94	01011110
_	95	01011111

ASCII Char.	Dec	Binary
`	96	01100000
a	97	01100001
b	98	01100010
c	99	01100011
d	100	01100100
e	101	01100101
f	102	01100110
g	103	01100111
h	104	01101000
i	105	01101001
j	106	01101010
k	107	01101011
l	108	01101100
m	109	01101101
n	110	01101110
o	111	01101111
p	112	01110000
q	113	01110001
r	114	01110010
s	115	01110011
t	116	01110100
u	117	01110101
v	118	01110110
w	119	01110111
x	120	01111000
y	121	01111001
z	122	01111010
{	123	01111011
|	124	01111100
}	125	01111101
~	126	01111110
	127	01111111

European Display Characters

ASCII Char.	Dec	Binary	ASCII Char.	Dec	Binary	ASCII Char.	Dec	Binary	ASCII Char.	Dec	Binary
NOTE	128	10000000		160	10100000	á	192	11000000		224	11100000
NOTE	129	10000001		161	10100001	é	193	11000001		225	11100001
NOTE	130	10000010		162	10100010	ó	194	11000010		226	11100010
NOTE	131	10000011		163	10100011	ú	195	11000011		227	11100011
NOTE	132	10000100		164	10100100	à	196	11000100		228	11100100
NOTE	133	10000101		165	10100101	è	197	11000101		229	11100101
NOTE	134	10000110		166	10100110	ò	198	11000110		230	11100110
NOTE	135	10000111		167	10100111	ù	199	11000111		231	11100111
NOTE	136	10001000		168	10101000	ä	200	11001000		232	11101000
NOTE	137	10001001		169	10101001	ë	201	11001001		233	11101001
NOTE	138	10001010		170	10101010	ö	202	11001010		234	11101010
NOTE	139	10001011		171	10101011	ü	203	11001011		235	11101011
NOTE	140	10001100		172	10101100	å	204	11001100		236	11101100
NOTE	141	10001101		173	10101101	é	205	11001101		237	11101101
NOTE	142	10001110		174	10101110	ô	206	11001110		238	11101110
NOTE	143	10001111	£	175	10101111	ü	207	11001111		239	11101111
	144	10010000		176	10110000	Ä	208	11010000		240	11110000
	145	10010001		177	10110001	î	209	11010001		241	11110001
	146	10010010		178	10110010	Ö	210	11010010		242	11110010
	147	10010011	°	179	10110011	Æ	211	11010011		243	11110011
	148	10010100		180	10110100	â	212	11010100		244	11110100
	149	10010101	ç	181	10110101	ì	213	11010101		245	11110101
	150	10010110	Ñ	182	10110110	Ø	214	11010110		246	11110110
	151	10010111	ñ	183	10110111	æ	215	11010111		247	11110111
	152	10011000	¡	184	10111000	Å	216	11011000		248	11111000
	153	10011001	¿	185	10111001	í	217	11011001		249	11111001
	154	10011010	¤	186	10111010	Ü	218	11011010		250	11111010
	155	10011011	£	187	10111011	ü	219	11011011		251	11111011
	156	10011100		188	10111100	É	220	11011100		252	11111100
	157	10011101	¢	189	10111101	ï	221	11011101		253	11111101
	158	10011110		190	10111110	ß	222	11011110		254	11111110
	159	10011111		191	10111111		223	11011111	▓	255	11111111

Note 1: All characters in the first column appear as an "ⁿ" character in edit mode and on the keyboard input line and system message line.
Note 2: Characters 128 thru 135 produce highlights on machines with monochrome highlights when used in PRINT and DISP statements.
Note 3: Characters 136 thru 143 change the color of text printed or displayed on machines capable of displaying text in color.
Note 4: Characters 144 thru 159 are ignored by PRINT and DISP statements.

Katakana Display Characters

ASCII Char.	Dec	Binary	ASCII Char.	Dec	Binary	ASCII Char.	Dec	Binary	ASCII Char.	Dec	Binary
	0	00000000		32	00100000	@	64	01000000	`	96	01100000
	1	00000001	!	33	00100001	A	65	01000001	a	97	01100001
	2	00000010	"	34	00100010	B	66	01000010	b	98	01100010
	3	00000011	#	35	00100011	C	67	01000011	c	99	01100011
	4	00000100	$	36	00100100	D	68	01000100	d	100	01100100
	5	00000101	%	37	00100101	E	69	01000101	e	101	01100101
	6	00000110	&	38	00100110	F	70	01000110	f	102	01100110
	7	00000111	'	39	00100111	G	71	01000111	g	103	01100111
	8	00001000	(	40	00101000	H	72	01001000	h	104	01101000
	9	00001001	)	41	00101001	I	73	01001001	i	105	01101001
	10	00001010	*	42	00101010	J	74	01001010	j	106	01101010
	11	00001011	+	43	00101011	K	75	01001011	k	107	01101011
	12	00001100	,	44	00101100	L	76	01001100	l	108	01101100
	13	00001101	-	45	00101101	M	77	01001101	m	109	01101101
	14	00001110	.	46	00101110	N	78	01001110	n	110	01101110
	15	00001111	/	47	00101111	O	79	01001111	o	111	01101111
	16	00010000	0	48	00110000	P	80	01010000	p	112	01110000
	17	00010001	1	49	00110001	Q	81	01010001	q	113	01110001
	18	00010010	2	50	00110010	R	82	01010010	r	114	01110010
	19	00010011	3	51	00110011	S	83	01010011	s	115	01110011
	20	00010100	4	52	00110100	T	84	01010100	t	116	01110100
	21	00010101	5	53	00110101	U	85	01010101	u	117	01110101
	22	00010110	6	54	00110110	V	86	01010110	v	118	01110110
	23	00010111	7	55	00110111	W	87	01010111	w	119	01110111
	24	00011000	8	56	00111000	X	88	01011000	x	120	01111000
	25	00011001	9	57	00111001	Y	89	01011001	y	121	01111001
	26	00011010	:	58	00111010	Z	90	01011010	z	122	01111010
	27	00011011	;	59	00111011	[	91	01011011	{	123	01111011
	28	00011100	<	60	00111100	¥	92	01011100			124	01111100
	29	00011101	=	61	00111101	]	93	01011101	}	125	01111101
	30	00011110	>	62	00111110	^	94	01011110	~	126	01111110
	31	00011111	?	63	00111111	_	95	01011111		127	01111111

Katakana Display Characters

ASCII Char.	Dec	Binary	ASCII Char.	Dec	Binary	ASCII Char.	Dec	Binary	ASCII Char.	Dec	Binary
NOTE	128	10000000		160	10100000		192	11000000		224	11100000
NOTE	129	10000001		161	10100001		193	11000001		225	11100001
NOTE	130	10000010		162	10100010		194	11000010		226	11100010
NOTE	131	10000011		163	10100011		195	11000011		227	11100011
NOTE	132	10000100		164	10100100		196	11000100		228	11100100
NOTE	133	10000101		165	10100101		197	11000101		229	11100101
NOTE	134	10000110		166	10100110		198	11000110		230	11100110
NOTE	135	10000111		167	10100111		199	11000111		231	11100111
NOTE	136	10001000		168	10101000		200	11001000		232	11101000
NOTE	137	10001001		169	10101001		201	11001001		233	11101001
NOTE	138	10001010		170	10101010		202	11001010		234	11101010
NOTE	139	10001011		171	10101011		203	11001011		235	11101011
NOTE	140	10001100		172	10101100		204	11001100		236	11101100
NOTE	141	10001101		173	10101101		205	11001101		237	11101101
NOTE	142	10001110		174	10101110		206	11001110		238	11101110
NOTE	143	10001111		175	10101111		207	11001111		239	11101111
	144	10010000		176	10110000		208	11010000		240	11110000
	145	10010001		177	10110001		209	11010001		241	11110001
	146	10010010		178	10110010		210	11010010		242	11110010
	147	10010011		179	10110011		211	11010011		243	11110011
	148	10010100		180	10110100		212	11010100		244	11110100
	149	10010101		181	10110101		213	11010101		245	11110101
	150	10010110		182	10110110		214	11010110		246	11110110
	151	10010111		183	10110111		215	11010111		247	11110111
	152	10011000		184	10111000		216	11011000		248	11111000
	153	10011001		185	10111001		217	11011001		249	11111001
	154	10011010		186	10111010		218	11011010		250	11111010
	155	10011011		187	10111011		219	11011011		251	11111011
	156	10011100		188	10111100		220	11011100		252	11111100
	157	10011101		189	10111101		221	11011101		253	11111101
	158	10011110		190	10111110		222	11011110		254	11111110
	159	10011111		191	10111111		223	11011111		255	11111111

Note 1: All characters in the first column appear as an � character in edit mode and on the keyboard input line and system message line.
Note 2: Characters 128 thru 135 produce highlights on machines with monochrome highlights when used in PRINT and DISP statements.
Note 3: Characters 136 thru 143 change the color of text printed or displayed on machines capable of displaying text in color.
Note 4: Characters 144 thru 159 are ignored by PRINT and DISP statements.

Master Reset Table

	Power On	SCRATCH A	SCRATCH	SCRATCH C	RESET	Note 2 END/ STOP	LOAD	LOAD &Go	GET	GET &Go	LOADSUB	Main Prerun	SUB Entry	SUB Exit
CRT														
CRT DISP Line	Clear	Clear	—	—	Clear	—	—	·—	—	—	—	—	—	—
CRT Display Functions	Off	Off	—	—	—	—	—	—	—	—	—	—	—	—
CRT Message Line	Ready	Clear	Clear	Clear	Reset	—	—	—	—	—	—	Clear	—	—
CRT Input Line (Note 6)	Clear	Clear	Clear	—	Clear	—	—	—	—	—	—	—	—	—
CRT Printout Area	Clear	Clear	—	—	—	—	—	—	—	—	—	—	—	—
CRT Print Position (TABXY)	1,1	1,1	—	—	Note 15	—	—	—	—	—	—	—	—	—
ALPHA ON/OFF (Note 3)	On	On	On	On	On	On	—	—	—	—	—	—	—	—
KEYBOARD														
Keyboard Recall Buffer	Clear	—	—	—	—	—	—	—	—	—	—	—	—	—
Keyboard Result Buffer	Empty	Empty	—	—	—	—	—	—	—	—	—	—	—	—
Keyboard Knob Mode	↕	↕	↕	↕	↕	—	—	—	—	—	—	—	—	—
Tabs On Input Line	None	None	—	—	—	—	—	—	—	—	—	—	—	—
Typing Aid Labels	Note 16	Note 16	—	—	—	—	—	—	—	—	—	—	—	—
Keyboard Katakana Mode	Off	Off	Off	—	Off	—	—	—	—	—	—	—	—	—
SUSPEND INTERACTIVE	Off	Off	Off	Off	Off	Off	Off	Off	Off	Off	—	Off	—	—
PRINTING														
Print column	1	1	—	—	1	—	—	—	—	—	—	—	—	—
PRINTALL	Off	Off	—	—	Off	—	—	—	—	—	—	—	—	—
PRINTALL IS	1	1	—	—	—	—	—	—	—	—	—	—	—	—
PRINTER IS	1	1	—	—	—	—	—	—	—	—	—	—	—	.
ENVIRONMENTS & VARIABLES														
Allocated Variables	None	None	None	None	Note 1	Note 1	None	None	None	None	—	None	None	Pre-ent
Normal Variables	None	None	None	None	—	—	None	None	None	None	—	Note 11	Note 11	Pre-ent
COM Variables	None	None	—	None	—	—	—	Note 9	—	Note 9	—	—	—	—
OPTION BASE	0	0	0	—	—	—	—	Note 9	—	Note 9	—	Note 9	Note 9	Pre-ent
I/O Path Names	None	Closed	Closed	Closed	None	Closed	Closed	Closed	Closed	Closed	—	Closed	—	sub clsd
I/O Path Names in COM	None	Closed	—	Closed	None	—	Note 10	Note 10	Note 10	Note 10	—	—	—	—
Keyboard Variable Access	No	No	No	No	Main	Main	No	In cnt.	No	In cnt.	In cnt.	Main	SUB	Pre-ent
BASIC Program Lines	None	None	None	—	—	—	Note 4	Note 4	Note 4	Note 4	Note 4	—	—	—
BASIC Program Environment	Main	Main	Main	Main	Main	Main	Main	Main	Main	Main	—	Main	SUB	Pre-ent
Normal Binary Programs	None	None	—	—	—	—	Note 5	Note 5	—	—	—	—	—	—
SUB Stack	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	Clear	Push	Pop
NPAR	0	0	0	0	0	0	0	0	0	0	—	0	Actual	Pre-ent
CONTINUE Allowed	No	No	No	No	No	No	No	Yes	No	Yes	Yes	Yes	Yes	Yes
ON <event> ACTIONS														
ON <event> Log	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty	—	Empty	Note 8	Note 8
System Priority	0	0	0	0	0	0	0	0	0	0	—	0	Note 7	Pre-ent
ON KEY Labels	None	None	None	None	None	None	None	None	None	None	—	None	—	Pre-ent
ENABLE/DISABLE	Enable	Enable	Enable	Enable	Enable	Enable	Enable	Enable	Enable	Enable	—	Enable	—	—
KNOBX	0	0	0	0	0	0	0	0	0	0	—	0	—	—

	Power On	SCRATCH A	SCRATCH	SCRATCH C	RESET	Note 2 END/STOP	LOAD	LOAD &Go	GET	GET &Go	LOADSUB	Main Prerun	SUB Entry	SUB Exit
MISC.														
GOSUB Stack	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	Clear	Local	Pre-ent
TIMEDATE	Note 14	—	—	—	—	—	—	—	—	—	—	—	—	—
ERRL, ERRN, and ERRDS	0	0	—	—	—	—	—	0	—	0	—	0	—	—
ERRM$	Null	Null	—	—	—	—	—	Null	—	Null	—	Null	—	—
DATA Pointer	None	None	None	None	None	None	None	1st main	None	1st main	—	1st main	1st sub	Pre-ent
LEXICAL ORDER IS	Stand.	Stand.	—	—	—	—	—	—	—	—	—	—	—	—
MASS STORAGE IS	Note 12	Note 12	—	—	—	—	—	—	—	—	—	—	—	—
CHECKREAD ON/OFF	Off	Off	—	—	—	—	—	—	—	—	—	—	—	—
Angle Mode	RAD	RAD	RAD	RAD	—	—	RAD	RAD	RAD	RAD	—	RAD	—	Pre-ent
Random Number Seed	Note 13	Note 13	Note 13	—	—	—	—	Note 13	—	Note 13	—	Note 13	—	—
DET	0	0	0	—	—	—	—	—	—	—	—	0	—	—
TRANSFER	None	Aborts	Note 17	Waits	Aborts	Waits	None	Note 18	None	Waits	—	None	—	Note 19
TRACE ALL	Off	Off	Off	—	—	—	—	—	—	—	—	—	—	—

— = Unchanged

Pre-ent = As existed previous to entry into the subprogram.

In cnt. = Access to variables in current context only.

1st main = Pointer set to first DATA statement in main program.

1st sub = Pointer set to first DATA statement in subprogram.

sub clsd = All local I O path names are closed at subexit.

Waits = Operation waits until TRANSFER completes.

Note 1: Only those allocated in the main program are available.

Note 2: Pressing the STOP key is identical in function to executing STOP. Editing or altering a paused program causes the program to go into the stopped state.

Note 3: Alpha is turned on automatically by typing on the input line, by writing to the display line, or by an output to the message line.

Note 4: Modified according to the statement or command parameters and file contents.

Note 5: Any new binary programs in the file are loaded.

Note 6: Includes cursor position, INS CHR mode, ANY CHAR sequence state, but not tabs, auto-repeat rate, and auto-repeat delay. (These last three are defaulted only at SCRATCH A and Power On.)

Note 7: The system priority changes at SUB entry if the subroutine was invoked by an ON <event> CALL.

Note 8: See the appropriate keyword.

Note 9: As specified by the new environment or program.

Note 10: A COM mismatch between programs will close I/O path names. If I/O path names exist in a labeled COM, and a LOAD or GET brings in a program which does not contain that labeled COM, those I/O path names are closed.

Note 11: Numeric variables are set to 0. and string lengths are set to 0.

Note 12: The default mass storage device is INTERNAL (the right-hand drive) on the 9826 and 9836. See the 9816 Installation Manual for information on its default mass storage device.

Note 13: The default random number seed is INT(PI × $(2^{31} - 2)/180$). This is equal to 37 480 660.

Note 14: The default TIMEDATE is 2.086 629 12 E + 11 (midnight March 1, 1900, Julian time).

Note 15: After a RESET, the CRT print position is in column one of the next line below the print position before the RESET.

Note 16: Typing aid labels are displayed unless a program is in the RUN state.

Note 17: Operation waits until TRANSFER completes unless both I/O path names are in COM.

Note 18: Operation waits until TRANSFER completes unless both I/O path names are in a COM area preserved during the LOAD.

Note 19: Operation waits until TRANSFER completes if the TRANSFER uses a local I/O path name.

Further Comments

The PAUSE key, the programmed PAUSE statement, and executing PAUSE from the keyboard all have identical effects. The only permanent effects of the sequence "PAUSE...CONTINUE" on a running program are:
1. Delay in execution.
2. Second and subsequent interrupt events of a given type are ignored.
3. INPUT, LINPUT, and ENTER 2 statements will be restarted.
4. ON KEY and ON KNOB are temporarily deactivated (i.e. not logged or executed) during the pause.
5. A TRANSFER may complete during the pause, causing ON EOT to be serviced at the next end-of-line.

Fatal program errors (i.e. those not trapped by ON ERROR) have the following effects:
— a PAUSE
— a beep
— an error message in the message line
— setting the values of the ERRL, the ERRN, and possibly the ERRDS functions
— setting the default EDIT line number to the number of the line in which the error occurred.

Autostart is equivalent to: Power On, LOAD "AUTOST", RUN.

CLR IO terminates ENTER and OUTPUT on all interfaces, handshake setup operations, HP-IB control operations, DISP, ENTER from CRT or keyboard, CAT, LIST, external plotter output, and output to the PRINTER IS, PRINTALL IS, and DUMP DEVICE IS devices when they are external. CLR IO does not terminate CONTROL, STATUS, READIO, WRITEIO, TRANSFER, real-time clock operations, mass storage operations (other than CAT), OUTPUT 2 (keyboard), or message line output.

CLR IO clears any pending closure key action.

If CLR IO is used to abort a DUMP GRAPHICS to an external device, the external device may be in the middle of an escape-code sequence. Thus, it might be counting characters to determine when to return to normal mode (from graphics mode). This means that a subsequent I/O operation to the same device may yield "strange" results. Handling this situation is the responsibility of the user and is beyond the scope of the firmware provided with the product. Sending 75 ASCII nulls is one way to "clear" the 9876 Graphics Printer.

Graphic Reset Table

	Power On	SCRATCH A	SCRATCH	SCRATCH C	RESET	Note 2 END STOP	GINIT	Main Prerun
PLOTTER IS	CRT	CRT	--	--	CRT	--	CRT	
Graphics Memory	Clear	Clear	--	--	Note 1	--	Note 1	
VIEWPORT	hrd clip	hrd clip	--	--	hrd clip	--	hrd clip	
X and Y Scaling (unit of measure)	GDU	GDU	--	--	GDU	--	GDU	
Set Clip	hrd clip	hrd clip	--	--	hrd clip	--	hrd clip	
Current Clip	hrd clip	hrd clip	--	--	hrd clip	--	hrd clip	
CLIP ON OFF	Off	Off	--	--	Off	--	Off	
PIVOT	0	0	--	--	0		0	
AREA PEN	1	1	--	--	1		1	
PEN	1	1	--	--	1		1	
LINE TYPE	1.5	1.5	--	--	1.5	--	1.5	
Pen Position	0,0	0,0	--	--	0,0	--	0,0	
LORG	1	1	--	--	1		1	
CSIZE	5.6	5.6	--	--	5.6		5.6	
LDIR	0	0	--	--	0		0	
PDIR	0	0	--	--	0		0	
GRAPHICS ON OFF	Off	Off	--	--	--			
ALPHA ON OFF (Note 3)	On	On	On	On	On	On	--	
DUMP DEVICE IS	701	701	--		--			
GRAPHICS INPUT IS	None	None	--	--	None		None	
TRACK ON OFF	Off	Off	--	--	Off		Off	
Color Map (Note 4)	Off	Off	--	--	Note 5	--	Note 5	
Drawing Mode	Norm	Norm	--		Norm	--	Norm	

— · Unchanged

hrd clip = The default hard clip boundaries of the CRT.

Note 1: Although RESET leaves the graphics memory unchanged, it will be cleared upon execution of the next graphics statement that sets a default plotter following the RESET.

Note 2: Pressing the STOP key is identical to executing STOP. Altering a paused program causes the program to go into the stopped state.

Note 3: Alpha is turned on automatically by typing on the input line, by writing to the display line, or by an output to the message line.

Note 4: With color map off, 8 standard colors are available. With color map on, 16 user-defined colors are available. See PLOTTER IS.

Note 5: Although the color map remains unchanged, it is changed if a graphics statement selects the device as a default plotter.

Interface Reset Table

	Power On	SCRATCH A	SCRATCH	BASIC RESET	Note 5 END/ STOP	LOAD	GET	Note 6 Reset Cmd	Main Prerun	SUB Entry	SUB Exit	CLR I/O
GPIO Card												
Interrupt Enable Bit	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	—
Active Timeout Counter	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	Clear	—	—	—
Enable Interrupt Mask	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	—
Hardware Reset of Card (PRESET)	Reset	Note 1	Note 1	Reset	Note 1	Note 1	Note 1	Reset	Note 1	—	—	Note 1
PSTS Error Flag	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	—
RS-232 Card												
Interrupt Enable Bit	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	—
Active Timeout Counter	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	Clear	—	—	—
Enable Interrupt Mask	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	—
Hardware Reset of Card	Reset	Reset	—	Reset	—	—	—	Reset	—	—	—	—
Data Rate/Character Format	Swtch	Swtch	—	—	—	—	—	—	—	—	—	—
RTS-DTR Latch	Clear	Clear	—	—	—	—	—	Clear	—	—	—	—
Request to Send Line	Clear	Clear	—	Clear	—	—	—	Clear	—	—	—	Note 2
Data Terminal Ready Line	Clear	Clear	—	Clear	—	—	—	Clear	—	—	—	Note 2
Line Status Register	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	Clear
Modem Status Register	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	Clear
Data-In Buffer	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty	—	—	Empty
Error-Pend. Flag	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	Clear
HP-IB												
Interrupt Enable Bit	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	—
Active Timeout Counter	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	Clear	—	—	—
Interrupt Enable Mask	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	—
User Interrupt Status	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	—
Serial Poll Register	Clear	Clear	—	Clear	—	—	—	Clear	—	—	—	—
Parallel Poll Register	Clear	Clear	—	Clear	—	—	—	Clear	—	—	—	—
My Address Register	Note 4	Note 4	—	—	—	—	—	—	—	—	—	—
IFC Sent	Note 3	Note 3	—	Note 3	—	—	—	Note 3	—	—	—	—
REN Set True	Note 3	Note 3	—	Note 3	—	—	—	Note 3	—	—	—	—
Data Communications												
Interrupt Enable Bit	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	—
Active Timeout Counter	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	Clear	—	—	—
Interrupt Enable Mask	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	—
Hardware Reset of Card	Reset	Note 7	—	Reset	—	—	—	Note 7	—	—	—	—
Line State	Dscon	Dscor	—	Dscon	—	—	—	Dscon	—	—	—	—
Data Buffers	Empty	Empty	—	Empty	—	—	—	Empty	—	—	—	—
Protocol Selection (Async or Data Link)	Swtch	Note 8	—	Swtch	—	—	—	Note 8	—	—	—	—
Protocol Options	Swtch	Swtch	—	Swtch	—	—	—	Swtch	—	—	—	—
BCD Card												
Interrupt Enable Bit	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	—
Active Timeout Counter	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	Clear	—	—	—
Interrupt Enable Mask	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	Clear	—	—	—
Hardware Reset of Card	Reset	Note 1	Note 1	Note 1	Note 1	Note 1	Note 1	Reset	Note 1	—	—	Note 1
Rewind Driver	Rwd	Rwd	Rwd	Rwd	Rwd	Rwd	Rwd	Rwd	Rwd	—	—	Rwd
BCD/Binary Mode	Swtch	Swtch	—	—	—	—	—	—	—	—	—	—

	Power On	SCRATCH A	SCRATCH	BASIC RESET	Note 5 END STOP	LOAD	GET	Note 6 Reset Cmd	Main Prerun	SUB Entry	SUB Exit	CLR I O
EPROM Programmer												
Hardware Reset of Card	Reset	Reset	—	Reset	—	—	—	Reset	-	—	-	—
Programming Time Register	Clear	Clear	—	—	—	—	—	Clear	-	—	-	—
Target Address Register	Clear	Clear	—	—	—	—	—	Clear	-	—	—	—

— = Unchanged

Swtch = Set according to the switches on the interface card

Dscon = A disconnect is performed

Note 1: Reset only if card is not ready.

Note 2: Cleared only if corresponding modem control line is not set.

Note 3: Sent only if System Controller.

Note 4: If System Controller and Active Controller, address is set to 21. Otherwise, it is set to 20.

Note 5: Pressing the STOP key is identical in function to executing STOP or END. Editing or altering a paused program causes the program to go into the stopped state.

Note 6: Caused by sending a non-zero value to CONTROL register 0.

Note 7: This is a "soft reset," which does not include an interface self-test or a reconfiguration of protocol.

Note 8: Set according to the value used in the most recent CONTROL statement directed to Register 3. If there has been no CONTROL 3 statement, the switch settings are used.

Second Byte of Non-ASCII Key Sequences (String)

Holding the CTRL key and pressing a non-ASCII key generates a two-character sequence on the CRT. The first character is an "inverse-video" K. This table can be used to look up the key that corresponds to the second character of the sequence. (On the small keyboard of the Model 216, some non-ASCII keys generate ASCII characters when they are pressed while holding the CTRL key.)

Character	Value	Key
space		1
!	33	STOP
"		1
#	35	CLR LN
$	36	ANY CHAR
%	37	CLR→END
&	38	Select
'	39	Prev
(	40	SHIFT -TAB
)	41	TAB
*	42	INS LN
+	43	INS CHR
,	44	Next
-	45	DEL CHR
.	46	Ignored
/	47	DEL LN
0	48	k0
1	49	k1
2	50	k2
3	51	k3
4	52	k4
5	53	k5
6	54	k6
7	55	k7
8	56	k8
9	57	k9
:	58	SHIFT -system f6 [2]
;	59	SHIFT -system f7 [2]
<	60	←
=	61	RESULT
>	62	→
?	63	RECALL
@	64	SHIFT - RECALL
A	65	PRT ALL
B	66	BACK SPACE
C	67	CONTINUE
D	68	EDIT
E	69	ENTER
F	70	DISPLAY FCTNS
G	71	SHIFT - →
H	72	SHIFT - ←
I	73	CLR I/O
J	74	Katakana Mode
K	75	CLR SCR
L	76	GRAPHICS
M	77	ALPHA
N	78	DUMP GRAPHICS
O	79	DUMP ALPHA

Character	Value	Key
P	80	PAUSE
Q		1
R	82	RUN
S	83	STEP
T	84	SHIFT -↓
U	85	CAPS LOCK
V	86	↓
W	87	SHIFT -↑
X	88	EXECUTE
Y	89	Roman Mode
Z		1
[	91	CLR TAB
\	92	▼
]	93	SET TAB
^	94	↑
_	95	SHIFT -▼
`		1
a	97	k10
b	98	k11
c	99	k12
d	100	k13
e	101	k14
f	102	k15
g	103	k16
h	104	k17
i	105	k18
j	106	k19
k	107	k20
l	108	k21
m	109	k22
n	110	k23
o	111	SHIFT -system f1 [2]
p	112	SHIFT -system f2 [2]
q	113	SHIFT -system f3 [2]
r	114	SHIFT -system f4 [2]
s	115	SHIFT -user f1 [2]
t	116	SHIFT -user f2 [2]
u	117	SHIFT -user f3 [2]
v	118	SHIFT -user f4 [2]
w	119	SHIFT -user f5 [2]
x	120	SHIFT -user f6 [2]
y	121	SHIFT -user f7 [2]
z	122	SHIFT -user f8 [2]
}	123	System
|	124	Menu
{	125	User
~	126	SHIFT - Menu
▒		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 System and user refer to the softkey menu which is currently active.

Selected High-Precision Metric Conversion Factors

English Units	Metric Units	To convert from English to Metric, multiply by:	To convert from Metric to English, multiply by:
Length			
mil	micrometre (micron)	2.54×10^{1} ☆	$3.937\ 007\ 874 \times 10^{-2}$
inch	millimetre	2.54×10^{1} ☆	$3.937\ 007\ 874 \times 10^{-2}$
foot	metre †	3.048×10^{-1} ☆	$3.280\ 839\ 895$
mile (intl.)	kilometre	$1.609\ 344$ ☆	$6.213\ 711\ 922 \times 10^{-1}$
Area			
inch^2	millimetre^2	$6.451\ 6 \times 10^{2}$ ☆	$1.550\ 003\ 100 \times 10^{-3}$
foot^2	metre^2	$9.290\ 304 \times 10^{-2}$ ☆	$1.076\ 391\ 042 \times 10^{1}$
mile^2	kilometre^2	$2.589\ 988\ 110$	$3.861\ 021\ 585 \times 10^{-1}$
acre (U.S survey)	hectare	$4.046\ 873 \times 10^{-1}$	$2.471\ 044$
Volume			
inches^3	millimetres^3	$1.638\ 706\ 4 \times 10^{4}$ ☆	$6.102\ 374\ 409 \times 10^{-5}$
feet^3	metres^3	$2.831\ 684\ 659 \times 10^{-2}$	$3.531\ 466\ 672 \times 10^{1}$
ounces (U.S. fluid)	centimetres^3	$2.957\ 353 \times 10^{1}$	$3.381\ 402 \times 10^{-2}$
gallon (U.S. fluid)	litre ‡	$3.785\ 412$	$2.641\ 721 \times 10^{-1}$
Mass			
pound (avdp.)	kilogram	$4.535\ 923\ 7 \times 10^{-1}$ ☆	$2.204\ 622\ 622$
ton (short)	ton (metric)	$9.071\ 847\ 4 \times 10^{-1}$ ☆	$1.102\ 311\ 311$
Force			
ounce (force)	dyne	$2.780\ 138\ 510 \times 10^{4}$	$3.596\ 943\ 090 \times 10^{-5}$
pound (force)	newton	$4.448\ 221\ 615$	$2.248\ 089\ 431 \times 10^{-1}$
Pressure			
psi	pascal	$6.894\ 757\ 293 \times 10^{3}$	$1.450\ 377\ 377 \times 10^{-4}$
inches of Hg (at 32°F)	millibar	$3.386\ 4 \times 10^{1}$	$2.952\ 9 \times 10^{-2}$
Energy			
BTU (IST)	Calorie (kg, thermochem.)	$2.521\ 644\ 007 \times 10^{-1}$	$3.965\ 666\ 831$
BTU (IST)	watt-hour	$2.930\ 710\ 702 \times 10^{-1}$	$3.412\ 141\ 633$
BTU (IST)	joule §	$1.055\ 055\ 853 \times 10^{3}$	$9.478\ 171\ 203 \times 10^{-4}$
ft•lb	joule	$1.355\ 817\ 948$	$7.375\ 621\ 493 \times 10^{-1}$
Power			
BTU (IST)/hr	watt	$2.930\ 710\ 702 \times 10^{-1}$	$3.412\ 141\ 633$
horsepower (mechanical)	watt	$7.456\ 998\ 716 \times 10^{2}$	$1.341\ 022\ 090 \times 10^{-3}$
horsepower (electric)	watt	7.46×10^{2} ☆	$1.340\ 482\ 574 \times 10^{-3}$
ft•lb/s	watt	$1.355\ 817\ 948$	$7.375\ 621\ 493 \times 10^{-1}$
Temperature			
°Rankine	kelvin	1.8 ☆	$5.555\ 555\ 556 \times 10^{-1}$
°Fahrenheit	°Celsius	$°C = (°F - 32) / 1.8$ ☆	$°F = (°C \times 1.8) + 32$ ☆

☆ Exact conversion
† Conversion redefined in 1959
‡ Conversion redefined in 1964
§ Conversion redefined in 1956

Note: The preferred metric unit for force is the newton; for pressure, the pascal; and for energy, the joule.

Prefix	Symbol	Multiplier
exa	E	10^{18}
peta	P	10^{15}
tera	T	10^{12}
giga	G	10^{9}
mega	M	10^{6}
kilo	k	10^{3}
hecto	h	10^{2}
deka	da	10^{1}

Prefix	Symbol	Multiplier
deci	d	10^{-1}
centi	c	10^{-2}
milli	m	10^{-3}
micro	μ	10^{-6}
nano	n	10^{-9}
pico	p	10^{-12}
femto	f	10^{-15}
atto	a	10^{-18}

Sources

American Society for Testing and Materials (ASTM). "Standard for Metric Practice" Reprinted from Annual Book of ASTM Standards.

U.S. Department of Commerce, National Bureau of Standards. "NBS Guidelines for the Use of the Metric System". Reprinted from Dimensions/NBS. (October 1977).

Subject Index

d

e

f

g

h

i

k

l

m

n

o

p

r

s

t

u

w

HEWLETT PACKARD