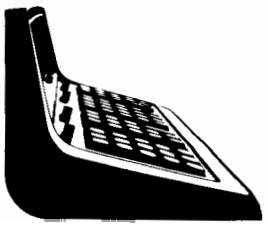
BASIC Language Interfacing Concepts

Part No. 09835-90600
Microfiche No. 09835-99600



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

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Table of Contents

Chapter 3: HP Interface Cards

Appendix

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.

September 1980. . . First Edition

December 1980. . . Second Edition pages: 13, 65, 66, 74, 85, 86, 87, 88, 92, 121, 133, 160, 161, 162, 177

| June 1981...Third Edition. Updated pages: 79, 132

Reader's Guide

This guide for interfacing peripheral devices to Hewlett-Packard desktop computers is designed to provide additional information which may be helpful to the user who needs to interface his peripheral equipment to HP desktop computers and/or program the resultant system for interfacing applications.

It is not intended to be a replacement for either the operating and programming manuals for the desktop computer, or the installation and service manuals for the individual interface cards. The maximum benefit can be obtained from this guide if these individual manuals are studied first. They provide the user with a detailed description of the individual operations available from the computer, and of the various functions provided by each interface card. At the same time they assume that the user has a certain level of knowledge about the programming techniques (software) and electronics (hardware) involved in interfacing applications.

The purpose of this interfacing guide is twofold. First, it is intended to provide some introductory hardware and software concepts which are assumed by the manuals, but with which the user may not have previous experience. The second purpose of the guide is to present an alternative approach to explaining the operations discussed in these manuals. For example, while the computer operating manual discusses the use and the detailed syntax of those programming statements associated with interrupt operations, the guide expands this information by discussing how interrupts are implemented, and when they should and should not be used. The guide also presents introductory information on such topics as the Hewlett-Packard Interface Bus (HP-IB) and serial I/O which is not available in the manuals for these interface cards. In addition, since this guide is not intended to describe a single computer, interface card, or peripheral device as a stand-alone piece of equipment, it can discuss the use of all three elements as an integrated system.

The guide is primarily oriented around the HP 9835/45B Desktop Computers and the five associated interface cards: the 98032A, 98033A, 98034A, 98035A, and 98036A. Example programs are presented in BASIC, the high-level programming language of the 9835/45B. However, a majority of the concepts that are discussed apply to interfacing in general and the user should find a reading of this guide helpful in understanding the operations of other HP desktop computers and interface cards. For example, the HP System 45B Desktop Computer uses the same set of interface cards, and operates in a manner similar to the System 35, with the System 35 program statements replaced by their 9845B equivalents.

There are some statements and functions that the System 35/45B computers have that are unavailable on a System 45A. The differences are minor, for the most part, and they can be minimized with alternative programming techniques. For example:

```
      9835A/B,9845B                          9845A

10   WRITE BIN 6;X              =  10   OUTPUT 6 USING "#,B";X
20   WAIT WRITE 6,4;X           =  20   IF NOT IOFLAG(6) THEN 20
                                &  30   WRITE IO 6,4;X
```

Chapter 1 of this guide presents general background information useful for interfacing applications. For the engineer not experienced in software concepts, information is given on computer data representations and I/O (Input/Output) programming. For the programmer not experienced in hardware concepts, topics such as logic levels, TTL gates, latches, and flip-flops are discussed. The reader with a background in hardware and software can proceed directly to Chapter 2.

In Chapter 2, the discussion is centered around programming for interfacing applications. It is not the purpose of this section to teach the BASIC programming language or to present the detailed syntax and restrictions of those programming statements related to I/O operations. This is the purpose of the operating manuals. Instead, the guide tries to give the user an appreciation for what takes place on the low level when the high level programming statements are executed. It is the philosophy of this guide that if the user understands these low level operations, many of the observations that appear to be strange from the high level will lose much of their mystery. Also, such an understanding should allow the reader to make more intelligent use of the power available in desktop computer systems.

Chapter 3 concentrates on the individual interface cards themselves. Here again, an alternative approach to the installation and service manuals for these cards is taken, and a register-operational model of these interfaces is developed. All of the functions provided by these cards are described in terms of sequences of register operations.

The appendices contain a collection of useful tables, diagrams, and timing information, along with a selected bibliography of references for additional reading.

Chapter 1
General Background Concepts

The Task of an Interface (an Overview)

In discussing interfacing peripheral devices to a desktop computer, the first question that naturally arises is "What does an interface do, and why is it necessary?" In order to answer this question, it is helpful to understand some of the characteristics of the computer and of the peripheral devices which it is to control.

A computer by itself is not a very useful device. Its power comes from its ability to accept inputs from an outside source, modify these inputs according to a given set of rules (as expressed by the program in the computer), and output the results of these computations to some external device. Some typical input devices are punched card readers, tape readers, digitizers, and digital voltmeters. Output devices would include printers, tape punches, plotters, and graphic displays. In addition, there is a seemingly endless list of special-purpose sensing devices (input) and control equipment (output) designed to perform particular tasks.

Ideally, every such device that was built would conform to some standard that specified all the characteristics of its I/O (Input/Output) connection, thus making all such devices "plug-to-plug" compatible. Unfortunately, no such standard exists. As a result, four major areas of incompatibility arise when one attempts to connect a peripheral device to a computing controller. It is the task of the interface to provide the necessary compatibility in these areas.

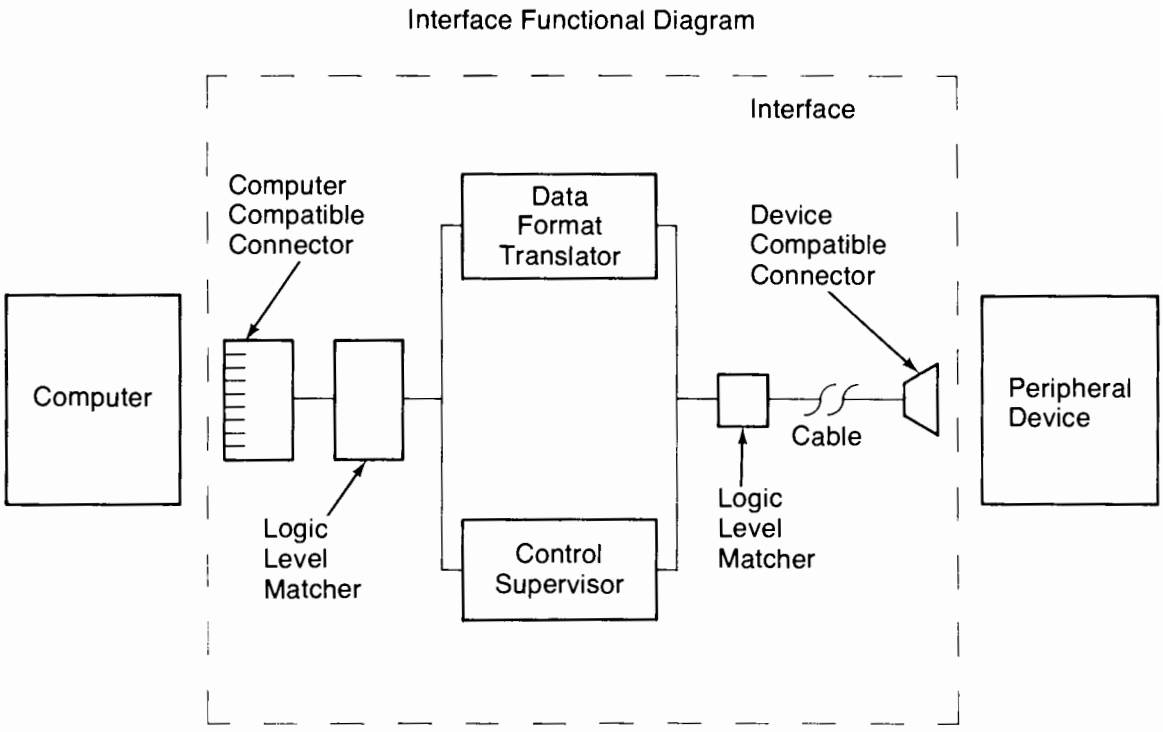


Figure 1

Mechanical Compatibility

The simplest requirement for the interface to meet is that of providing mechanical compatibility. This consists of merely supplying the appropriate connector at each end of the interface, and wiring the connectors in such a way that each input line at one end of the interface is connected to its corresponding output line at the other end (see Figure 1). If there were no other incompatibilities to overcome, this pair of cross-wired connectors would constitute the entire interface. In practice, things are rarely this simple.

Electrical Compatibility

A second function of an interface is to match the electrical characteristics (i.e., current and voltage levels, sometimes called logic levels) of the computer to those of its peripheral. Since HP desktop computers and their associated interfaces are designed using compatible electronic logic levels (called TTL), the logic-level-matcher functional block at the computer end of the generalized interface shown in Figure 1 is not necessary. Fortunately, many peripheral devices also use TTL levels in their circuitry. A discussion of TTL levels is contained in the Logic Levels and TTL implementations section, along with information on interfacing to devices that use other voltage levels.

Data Compatibility

Once an interface has made the computer and its peripheral device mechanically and electrically compatible, they are capable of exchanging messages as electrical signals over wires called data lines. But just as two humans who do not speak the same language need a translator, data messages between a computer and its peripheral may also require some sort of format translation. The computer with its versatile programming capability, will usually perform this function. But in some cases, this task is given to the interface for reasons of speed. The 98033A BCD and the 98036A Bit Serial interfaces are examples of cases where the task of data reformatting is assigned to the interface. More discussions of this data translation process are contained in the sections describing these interfaces.

Timing Compatibility

Humans have the remarkable ability to talk and listen at the same time (or at least in rapid succession) without losing too much of the content of the conversation, as our speaking and listening rates are well matched. Computers and their peripheral devices, on the other hand, have such a wide range of operating speeds that a much more orderly mechanism is required for successful transfer of data messages. Providing timing compatibility (sometimes called the handshake function), along with other miscellaneous control operations, is the fourth major task of the interface.

This overview of the various functions of an interface has been very general. The sections that follow give more detailed information about the ways that HP interfaces implement each of these functions, along with other background information on HP desktop computer architecture, data formats, and other topics related to interfacing.

Software

Data Representations in the Computer

Since the primary purpose of interfacing is to exchange data between a computing controller and its peripheral devices, or between two computers, it would be helpful to first look at how this data is represented within the computer.

The memory of any digital computing device is made up of a large number of storage locations called bits. The number of bits that make up the memory can vary from a few hundred in a small hand-held calculator to several million in large computers. Each of these bits ("bit" is an abbreviation for **bi**nary digi**t**) can be set to and will maintain one of two states. Depending on the meaning assigned to it, the bit may represent yes or no, on or off, one or zero, true or false, etc. A single bit by itself, however, is only capable of representing simple two-state information.

To store more complex information, it is necessary to group several bits together into a logical package. For example, if we wish to represent the decimal digits 0 through 9 in the computer memory, we could collect bits into groups of four, and use the following encoding scheme.

0	1	2	3	4	5	6	7	8	9
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Figure 2

Since each bit can take on two states (represented here by the symbols 0 and 1), a group of N bits can take on 2^N states. In this example, the groups of four bits are capable of representing $2^4 = 16$ states. Since there are only 10 decimal digits to be represented, we do not use 6 of the possible 16 states. To represent alphabetical information, we would need to have a representation for each of the 26 letters of the English alphabet. This would require groups of 5 bits each, since $2^5 = 32$. To represent both decimal digits and English letters (36 characters total) would require 6 bits.

In the example above, we could just as easily have assigned the following encoding scheme: 0 = 0110, 1 = 1011, 2 = 0000, 3 = 0011, 4 = 0101, etc. And indeed, many computers use an internal representation of letters, numbers, and symbols that make the task of performing the desired operations on these items as simple as possible. This will vary from computer to computer depending on how it will manipulate this data. This variety of internal representations causes no problem until two computers or a computer and its peripheral need to exchange data. Then it becomes necessary for both devices to use the same data representation, or for one of the devices to be capable of translating between the two representations.

A third alternative is that each device may use whichever internal representation is most convenient, but that all data will be input or output in some standard representation. There are several of these standard representations that are becoming popular and widely accepted, depending on the particular job. For example, if only numeric data is to be represented, the encoding scheme first given in our example is widely used. This scheme is called BCD or Binary Coded Decimal representation. One of the most general and widely-used encoding schemes for data exchange is known as ASCII (pronounced as'ki), which is an acronym for **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. The ASCII code commonly uses 8-bit packages and has representations for numerical digits, upper-case, and lower-case letters, common typewriter symbols (#,$,%,&,=,?, etc.), and special control characters (carriage-return, line-feed, etc.). A complete table of the ASCII encoding scheme is found in the Appendix. A large number of peripheral devices made by HP and other manufacturers use ASCII code for sending and receiving alpha-numeric data.

HP desktop computers also use 8-bit ASCII code for the internal representation of alpha-numeric data (called strings). These 8-bit packages are so convenient for data representations that they have been given the name "byte." Indeed, it is now quite common to measure memory sizes in terms of these 8-bit bytes.

Although 8-bit bytes are ideal for storage and transfer of alpha-numeric strings of characters, they are not very well suited for internal representation of numeric values. It is difficult to perform arithmetic operations on numbers that are expressed as strings of ASCII symbols.

The simplest method for storing and manipulating numeric values uses the so-called binary representation. In this method, a group of N bits is used to represent a number, and each position in the group has a value which is a power of two. For example, to represent the number 98 as an 8-bit binary number, we note that when broken into powers of two, 98 = 64 + 32 + 2 as shown in Figure 3.

bit #	7	6	5	4	3	2	1	0
value	128	64	32	16	8	4	2	1
Eg., 98 =	0	1	1	0	0	0	1	0

Figure 3

Since any number can be expressed as a sum of powers of two in only one way, this binary representation yields a unique pattern for each number. In numbering the bits, we have called the least significant bit "bit zero." It is also common to find the bits numbered starting from one. Most of the manuals and documentation for HP desktop computers number the bits starting with zero, since this makes the value of the n-th bit position equal to 2^n. But being aware that two conventions for numbering the bits are in common usage could help to avoid possible confusion.

In the example given above, the largest number that can be represented by the 8-bits is 255. Thus, we say that the range of an 8-bit binary representation is zero to 255 (often written as [0,255]). If we need to represent wider ranges of values, we can use larger groupings of bits. Indeed, all HP desktop computers (except the 9815) use groups of 16 bits (called words) to represent binary data inside the machine. These binary values are used for such things as counters, limit values (as in saving the size of an array), and pointers to locations within the memory.

One limitation of the binary system just described, however, is that only positive integers are represented. Negative values can be easily incorporated into the system if we pick one bit (usually the highest one) to represent the sign of the number. For example, if we use an 8-bit byte and let bit 7 be the sign bit, using the convention that 0 is plus and 1 is minus, then 00000101 would represent a + 5 while 10000101 would represent − 5. This convention is called the "sign/magnitude" binary representation. It is simple to understand, but unfortunately it causes difficulties in computation. This is because the hardware processor that does arithmetic on these numbers must have a subtractor as well as an adder. This makes the processor more costly and less efficient, since it must first decide (from the sign bits) whether an add or subtract must be done.

An alternative representation for both positive and negative binary values is called the "two's complement form." In this form, positive values have the same form as in the sign/magnitude representation. Negative numbers, however, are formed by the following rule: complement the number (i.e., replace all ones with zeros and zeros with ones) and add one (ignoring any carry out of the highest bit). For example, + 5 is still represented by 00000101. Minus 5 is obtained by complementing (11111010) and adding one (11111011). Thus, in an 8-bit, 2's-complement representation, − 5 = 11111011. Notice that if we apply the complement-add-one rule to the representation for − 5, we get back the representation for + 5, so that the rule is symmetric. The advantage of 2's-complement notation is that only an adder is required. For example, to calculate the value of 7 − 5, we rewrite it as

$$7 + (-5) = 00000111 + 11111011 = 00000010 = 2.$$

Thus we subtracted 5 from 7 using only a binary adder.

The table below gives an example of all values that can be represented by a 3-bit binary number in 2's-complement form.

−4	−3	−2	−1	0	1	2	3
100	101	110	111	000	001	010	011

Figure 4

In general, an N-bit, 2's-complement form can represent all integers in the range $[-2^{N-1}, +2^{N-1}-1]$. For the 16-bit binary values that are used internally for counters and pointers, this range is [−32768, 32767]. If larger ranges of integers need to be represented, packages of larger number of bits could be used. Notice that the representations of values is not independent of the number of bits used in the representation. For example, in the table above, 101 represents a − 3 when using a 3-bit, 2's-complement format; but 101 represents a + 5 in a system using 4 or more bits.

Examples

1. Show that in 16-bit, 2's-complement form, the two decimal values + 5000 and − 5000 are represented by 0001001110001000 and 1110110001111000 respectively.

   0001001110001000 = 8 + 128 + 256 + 512 + 4096 = 5000.

   To find the decimal equivalent of 1110110001111000, first convert to its positive equivalent by complementing the bits (yielding 0001001110000111) and add one to get 0001001110001000. Since this is the binary form of + 5000 as found above, the original pattern represents − 5000.

2. Show that the binary numbers below have the equivalent decimal representations given.

   0101011011001110 = 22222
   0011000000111001 = 12345
   1111111111111111 = − 1
   1000000000000001 = − 32767

Notice that when the "complement and add one" operation is performed on the binary equivalent of − 32768, the same binary pattern is re-generated. This is because there is no 16-bit, 2's-complement representation for a + 32768. Thus, when using the rule of converting between positive and negative binary values, a one in the sign bit and all other bits being zeros must be treated as a special case.

We still have not solved the problem of representing non-integer values. In the decimal system we handle this by the use of a decimal point. For example,

12.75 = 1x(10) + 2x(1) + 7x(1/10) + 5x(1/100).

We could represent this same number in binary by the use of a "binary point" as

12.75 = 1100.11 = 1x(8) + 1x(4) + 0x(2) + 0x(1) + 1x(1/2) + 1x(1/4).

In each system, there are some numbers that cannot be exactly represented in a finite number of places. For example, the decimal representation of 1/3 = 0.33333... requires an infinite number of threes to represent exactly. Similarly, the binary representation of 1/10 = 0.0001100110011... cannot be represented exactly. Since most data presented to a computer from the real world is in decimal form (e.g., $235.17), conversion to binary form for internal storage and computation often results in inaccuracy due to the lack of an exact representation. This inaccuracy is in addition to any roundoff errors introduced by the subsequent calculations performed on that value.

To get around this deficiency, real numbers are stored within HP desktop computers in a decimal format. The structure of this format is shown below.

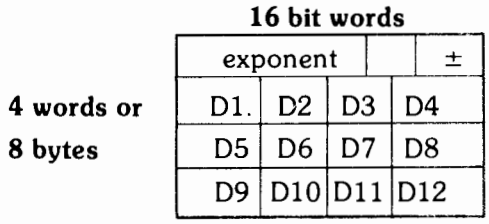


Figure 5

Each value occupies four 16-bit words (8 bytes). Each digit uses four bits and is in BCD format, with four digits packed into one 16-bit word. The sign and exponent of the number are encoded into the first word of the representation. Bit 0 is the sign of the value (0 = plus, 1 = minus), while bits 15-6 represent the exponent using a 10-bit, 2's-complement form. (Bits 5-1 are not used.) All calculations are done in this so called "floating point" format, and the task of converting between this representation and a string of ASCII characters for I/O purposes is relatively straightforward.

Examples

1. Calculate the four 16-bit words that are the internal representation of the following decimal values.

   a. 2.71828182846?

Answer:	0000000000000000	exponent = 0, sign = +
	0010011100011000	2 7 1 8
	0010100000011000	2 8 1 8
	0010100001000110	2 8 4 6

   b. − 1234.56789?

Answer:	0000000011000001	exponent = 3, sign = −
	0001001000110100	1 2 3 4
	0101011001111000	5 6 7 8
	1001000000000000	9 0 0 0

c. – 0.00123456789?

Answer: 1111111101000001 exponent = – 3, sign = –
0001001000110100 1 2 3 4
0101011001111000 5 6 7 8
1001000000000000 9 0 0 0

Input/Output Data Representations

We just looked at data representations within HP desktop computers. The table below summarizes these representations.

Data Type	Bits Used	Representation
Strings	8-bit bytes	ASCII
Integer	16-bit words	2's-complement binary (user program variables)
Numeric	64-bit registers	Decimal floating-point (user program variables)
Short	32-bit registers	Decimal floating-point Short precision (user program variables)

Figure 6

For I/O purposes, these internal representations must be converted into a format that can be understood by — and is dependent upon — the particular peripheral with which the computer is to communicate. Each peripheral can be categorized by two characteristics for purposes of data transfer: the number of bits required for each item of data transferred, and the format of those data bits (ASCII, binary, BCD, etc.). A small number of data types are sufficient to handle most peripheral devices, and HP desktop computers provide interfaces for each of these major categories. A detailed description of each of these interfaces is contained in later sections of this guide; and here we will merely look at the types of data formats that each interface card supports.

The Five Types of HP Interfaces

98032A Bit Parallel Interface

Because of its great versatility, this card is the general-purpose interface used with most standard HP peripherals and many special-purpose devices supplied by the user. It can accommodate data items of up to 16-bits in parallel. Assume, as an example, that this interface card is being used to connect the computer to a printer which uses the ASCII character set. Each character to be sent to the printer would be encoded using the 8-bit ASCII representation shown in the Appendix. To send an entire message such as "The value of pi is 3.14159." to the printer, each character would be sent all 8-bits at once in parallel. That is, all eight bits would be presented to the printer at once, one on each of eight separate data lines. When all eight data lines are set to the proper pattern of ones and zeros to represent the character being sent, the printer is told that the data on the lines is now valid, the printer senses the pattern on those lines, and prints the ASCII character assigned to that particular pattern. When the printer indicates that it has printed the character just given to it, the computer then changes the data lines to represent the next character in the message and the cycle repeats. This method of data transfer is sometimes called "bit-parallel, character-serial transmission."

Notice that when using the ASCII code, only 8 of the 16 data lines are used. Other peripheral devices which use codes other than ASCII might use only a few or all 16 of the data lines to represent their data. It is also important to note that HP desktop computers only provide ASCII representations of I/O data automatically. That is, when high-level I/O statements (such as ENTER and OUTPUT) are used in a program, they generate and expect to receive data coded in the ASCII representation. If any other encoding scheme is used, it is up to the user's program to know the representation being used and to convert the bit patterns received into a form that can be used within the computer.

Sometimes this is a simple task. For example, if a peripheral device supplied data in the form of 16-bit, 2's-complement numbers, the program would read a 16-bit value, convert it to internal floating point representation (see the Data Representations in Computer Section), and return the decimal equivalent of that value which would be a number in the range -32768 to $+32767$. Another device, however, might send only positive values using 16-bit binary representation. That is, it does not use the 2's-complement form, but rather all bits represent positive powers of two giving the 16-bit number a range of 0 to 65535. Since the READBIN function only reads numbers in 16-bit, 2's-complement form, the following program segment would be required to do the necessary conversion.

```
50          A=READBIN(4)
60          IF A<0 THEN A=A+65536
```

98033A BCD Interface

Data representations from input devices fall into three major categories. These are ASCII (directly supported by the enter statement), binary values (obtained with the READ-BIN function), and all other codes, which must be interpreted by the user's program. Of these other codes, one is in such common use that a special interface card has been developed to take the burden of data translation from the user's program. This code is called the BCD (Binary Coded Decimal) representation. It is typically used in measurement instruments such as a digital voltmeter (DVM). For example, assume that we have a DVM that is measuring a voltage level of 12.735 millivolts. The output connector of the DVM would supply four data lines for each digit in the reading (see Figure 7). Each of these digits would be encoded using the 4-bit BCD representation shown in Figure 2. In addition, a few data bits (typically 3 or 4) would be used to represent the range that the DVM is set to (i.e., volts, kilovolts, millivolts, etc.).

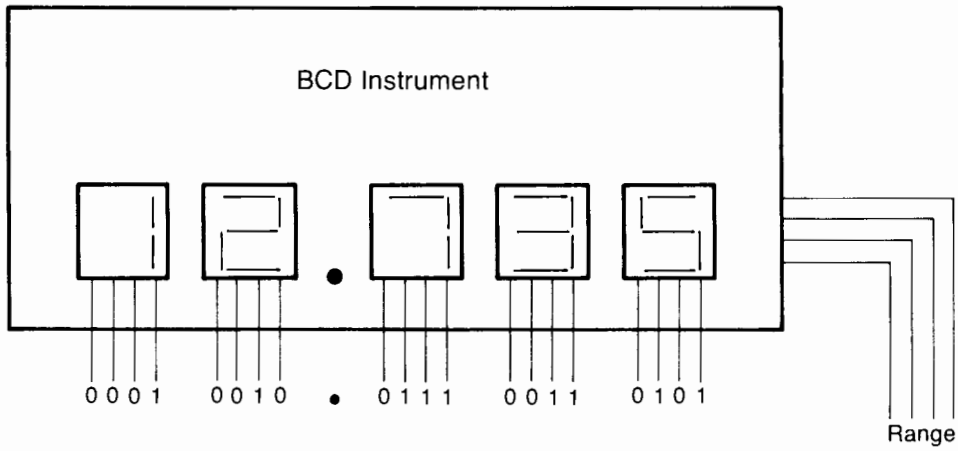


Figure 7

In using the 98032A Bit Parallel Interface to take a reading from this instrument, we would encounter two major problems. Since a 5 digit reading is represented by more than 16 bits, the DVM would need to deliver a reading in the form of two 16-bit packets. Our program would have to break up these two 16-bit patterns four bits at a time and convert them to digits and a range multiplier for the value being read, then combine these digits and the multiplier to form a number that represented the reading that was taken. This would be a complex and time consuming task for the program.

Instead, this task is performed by the 98033A BCD Interface. This card accepts from the device up to eight 4-bit BCD digits and a 4-bit multiplier in parallel, then converts this reading into a sequence of ASCII characters (in our example, "12.735E−3") that can be directly read by the computer's ASCII read statement. More information about the capabilities of the 98033A interface is contained in Chapter 3.

98034A HP-IB Interface

The task of interfacing a peripheral device to the computer would be greatly simplified if the four areas of interfacing incompatibility discussed in "The Task of an Interface" section could be overcome. That is, if a standard were developed that completely specified the mechanical, electrical, data, and timing characteristics of an I/O bus, then all computers and peripheral devices that followed this standard would be "plug-to-plug" compatible.

Such a standard has been adopted by the Institute of Electrical and Electronic Engineers (IEEE 488-1978). This standard has become so popular that dozens of manufacturers are providing hundreds of devices which conform to its specifications and can be interfaced to one another by simply plugging them together. There is no special representation which must be used for data messages on this bus, although the vast majority of IEEE-488 devices have implemented ASCII as their encoding scheme.

The 98034A HP-IB (Hewlett-Packard Interface Bus) card interfaces HP desktop computers to the IEEE-488 bus. A more detailed description of the HP-IB is given in Chapter 3.

98036A Serial I/O Interface

A new data representation problem arises in the area of data communications. Strictly speaking, any exchange of data between a computer and its peripheral devices could be called data communications; but this term is usually reserved to mean the exchange of data between two computers (or between a computer and a terminal) that are located at some distance from one another. If both machines are in the same building, they are usually connected by long cables. If they are in different buildings (or different cities) telephone lines might be used to make the connection. In either case, the cost of the connection rises rapidly with the number of bits that are sent in parallel. Therefore, a scheme has been devised that allows the exchanged information to be sent over a single data line.

Using this method, not only are the characters of the message sent in a serial fashion, but the bit patterns for each character are also sent serially, one bit after another along the single data line. This requires some rather sophisticated timing considerations which are handled by the interface itself. This allows the program to treat the interface as a simple 8-bit parallel device. That is, the user writes his message to the interface as a sequence of 8-bit (usually ASCII) bytes, just as he would to the 98032A interface. The Serial I/O interface then performs the task of converting each character to a bit-serial stream and sending it over the data communications line. For input, the interface receives a sequence of bits for each character, assembles them into a parallel 8-bit byte, and delivers this byte to the computer all at once.

More information about the particular capabilities of the 98036A interface is contained in Chapter 3.

98035A Real Time Clock

It is sometimes necessary to time operations or to have an interrupt generated at a certain point in time. The 98035A card was developed to handle these real time operations.

The real time clock contains its own rechargable battery so that the real time is maintained even when the computer is off. The battery is recharged each time the computer is turned on and maintains the correct time for at least two months on a full charge.

The real time clock has four timing/counting units all running off of the same 1MHz crystal. There are also four input and four output ports which can be used in combination with the four timing/counting units. These input and output ports act as buffers to external drivers and external receivers, respectively.

THIS PAGE
LEFT BLANK
INTENTIONALLY

The Data Transfer Process

Up to now, we have been concerned with how the various bit patterns on the data lines are to be interpreted. We have talked about sending and receiving sequences of characters, but have not mentioned how this process is accomplished.

The main difficulty involved is one of timing. If the speed of the computer and its peripheral are not exactly matched, the faster device will somehow have to slow down the pace of its I/O operations so that it will not get ahead of the slower device. This is accomplished through a mechanism known as "handshake." The detailed description of the handshake process is discussed in the sections on the interface cards, and here only the concept of the handshake will be considered.

Handshake for the output process (Figure 8) proceeds as follows. The first of a sequence of characters to be transmitted is placed on the data lines. When this operation is complete, the interface indicates that the data is valid by setting a special control line. When the peripheral detects that this control line is set, it raises another line called flag to indicate to the computer that it is momentarily going to go busy in order to process that character. It then takes the information from the data lines and processes it. This processing may involve printing a character, plotting a point, or whatever other function the peripheral device is designed to perform. Some devices do not operate from single characters, but wait until an entire sequence of characters is received to perform their actions. For example, the 9876A/B Thermal Line Printer contains a block of read/write memory called a buffer, into which characters to be printed are placed. For this device, the processing of most characters consists of merely placing that character in the buffer. Then when it receives a line-feed character (ASCII 10), it prints the entire line contained in its buffer. In any case, when the processing of that character is complete, the peripheral lowers the flag to indicate that it is again ready. The computer then places the next character on the data lines and the entire handshake process repeats again.

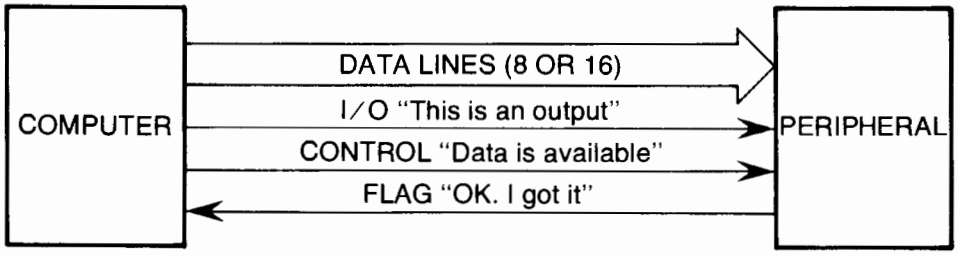


Figure 8. Data Output Handshake

The sequence of events for input (Figure 9) is similar to that for output. On a separate I/O indicator line, the computer specifies that an input operation is to be performed, and then sets the control line. This time when the peripheral sees control go set, since the I/O line is indicating input[1], it knows that it is to supply the data. The peripheral first sets the flag line to logic "zero" to indicate that it is busy, and goes to gather the requested data. This may involve taking a sample for a DVM, advancing a paper tape, digitizing an X,Y coordinate, or doing whatever the device was designed to do in order to gather data. This data is then placed on the data lines by the peripheral and the flag line then set to logic "one" to indicate that the data is now ready. The computer will then read the data and the handshake on one input character is complete. If a complete reading consists of several characters, the computer will again set the control line when it is ready for the next character and the process repeats.

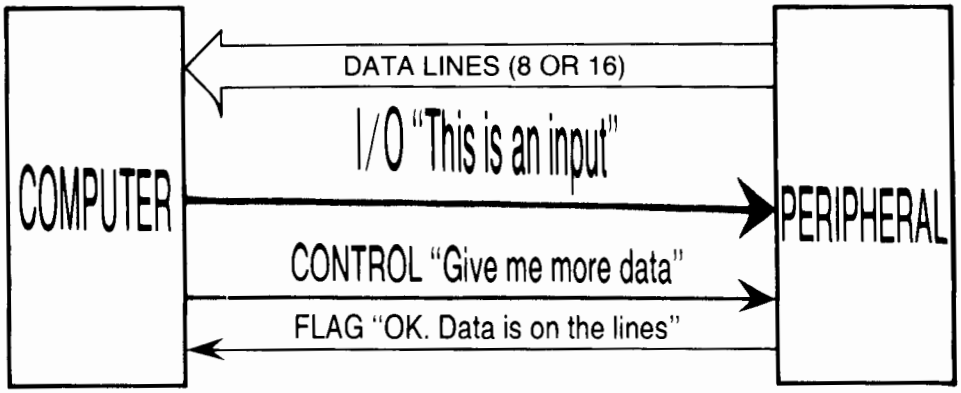


Figure 9. Data Input Handshake

It is important to note that in both the input and the output processes, the computer initiates the handshake procedure by setting the appropriate state of the I/O line and then setting control. Under no circumstances does the peripheral ever initiate a data transfer operation. This concept will be especially important when we discuss the interrupt process. Interrupts are always generated by the peripheral in response to a request from the computer, and not at the discretion of the peripheral device.

Finally, it should be mentioned that the concept of a handshake is a very general one and not limited to the description given here. Other schemes are possible and commonly used. This particular version of a three-wire handshake (I/O, control, flag) is adopted by the HP 98032A interface card and is the one that should be understood when connecting this interface to peripheral devices.

Most data and control lines on the HP interface cards use negative-true logic. It is easy to tell whether negative-true or positive-true logic is being used for a particular line from the schematic diagram of the interface card, which is found at the back of the installation and service

1 Throughout this guide, the terms input and output are always used with the computer as the point of reference. Thus, input means from the peripheral to the computer.

manual for that interface. If the name of the line (e.g., PFLG) appears with a bar drawn over the top of it, that line is using negative-true logic; otherwise it is using positive-true logic. For example, the 98032A interface has two general-purpose control lines called CTL0 and CTL1. The state of CTL0 is set from the program by use of the CONTROL MASK and WAIT WRITE statements as shown below:

```
CONTROL MASK <select code>;<control byte>
WAIT WRITE<select code>,5;<control byte>
```

The least significant bit of the control byte is used to set CTL0. Since on the schematic diagram for this card, that line is labeled with a bar over it, it is using negative-true logic. Thus, the statements

```
CONTROL MASK 6;0
WAIT WRITE 6,5;0
```

will set the CTL0 line high, and

```
CONTROL MASK 6;1
WAIT WRITE 6,5;1
```

will set it low.

The WAIT WRITE statement is sufficient to actually store a value into the control register (R5) of the interface. The CONTROL MASK statement is included above to assure that our control byte is left unchanged by an ENTER or OUTPUT or some other I/O statement directed to the same interface. These other I/O statements manipulate the control (R5) register to accomplish their own task, however, they do include the user's control byte value as a mask (as set by CONTROL MASK) to avoid altering external lines and such. If you do register I/O to the control register (R5), its considered good programming practice to establish the same control byte for the CONTROL MASK (as a protective measure) as for the WAIT WRITE (to write into R5). See "The Use of the Control Register" for an in-depth explanation of this process.

Hardware

Logic Levels and TTL Implementations

In previous sections we have used such phrases as "putting data on the lines," "setting the control line," and "making the flag line go busy," without really saying how these things are electrically implemented by the interface. But when it comes time to wire the interface to a non-standard peripheral device, it is helpful to understand how the electronic circuitry of the interface card relates to the operational concepts we have been discussing. In this section we will discuss some of the electronic concepts necessary to understand that circuitry.

The two main electrical concepts involved are those of voltage and current. For our purposes here, it may be helpful to explain these concepts in terms of an analogy with a forced air heating system found in many houses. In this system, after the air has been heated, a blower is used to create a pressure that is higher than the surrounding atmospheric pressure in the rest of the house. This blower is connected through a series of air ducts to the outlet registers placed throughout the various rooms. Because the pressure at the blower is higher, the heated air is forced to flow through the ductwork and out the registers. The pressure at any point in the system is always at a level somewhere between the maximum pressure produced by the blower and the atmospheric pressure at the outlets, and is determined by how much resistance the air has encountered from the ductwork along its path from the blower to the point which we are measuring. More air will travel along those paths in the heating system that present a lower resistance to the air flow. Indeed, the homeowner can vary the resistance in various branches by opening and closing louvers at the registers, resulting in redistributing the airflow throughout the house.

In an electrical system, the voltage at any point in the system can be thought of an analogous to the pressure in our heating network, and the current as analogous to the air flow. Just as the blower created air pressure above the normal atmospheric level, a battery or an active power supply is used to obtain voltage levels above some background reference point usually referred to as ground level or simply a ground. By allowing current to flow from the power supply to ground through appropriately chosen electrical resistors, we can obtain any desired voltage levels in this range to be used for whatever purposes we require.

An example of this is shown in Figure 10. At the top of the circuit we connect a 5 volt power supply, and the triangle at the bottom is a common symbol used to represent a ground point (e.g., a voltage level of zero).

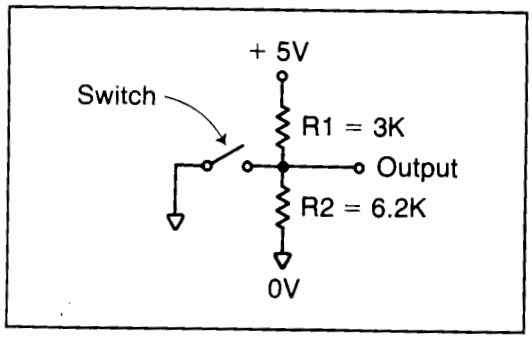


Figure 10

Current flows through the two resistors (R1 and R2) establishing some intermediate voltage level at the output. The formula for calculating this output voltage is given by

$$V_{out} = V_{in} * \left(\frac{R2}{R1 + R2}\right)$$

Using the values of V_{in} = 5 volts, R1 = 3K (resistances are measured in units called ohms, and K is an abbreviation for kilo-ohms = 1000 ohms), and R2 = 6.2K, we obtain a value for the output voltage of approximately 3.4 volts. If we now connect the output point to ground through a switch, by opening and closing this switch we can change the output voltage from 3.4V to 0V. That is, when the switch is closed, the resistance in this path is almost zero (only the small resistance of the wire itself) and practically no current flows through the R2 path. Thus the entire 5 volts is dropped by R1 leaving the voltage at the output point zero.

If we run a wire from the output point to someone who has a voltage measuring device like a voltmeter, as we open and close the switch he will see his voltmeter register 3.4V and 0V alternately. And if we now agree on some meaning to be assigned to the high and low voltage levels, we can use this electrical circuit to transmit information.

For example, the flag line of the interface card uses the high voltage level to indicate busy, and the low level to indicate ready. And rather than the mechanical switch, the interfaces employ electronic devices called gates to switch between high and low levels at electronic speeds. These gates will be discussed later in this section.

The signaling scheme described would work just as well using other values for the power supply voltage, resistors, and output voltage. The example values given were chosen because they correspond to the TTL (**T**ransistor-**T**ransistor **L**ogic) levels that are in common usage in computer hardware. Prepackaged integrated circuits are readily available which are used in generating and detecting high and low voltage levels, and in performing certain "logic" operations on these signals as will be discussed later. These chips or IC's as they are called, are made up of a large number of transistors and other electronic elements reduced to a very small size and sealed in convenient packages. It is the electrical properties of these transistors that dictate the high and low voltage levels that must be used. In general, it is very difficult and expensive to provide circuits that will provide and detect exact voltage levels. Therefore, TTL devices allow a range of voltages given by the table in Figure 11.

TTL High Level	= 3 volts to 5 volts
Indeterminate	= 0.7 volts to 3 volts
TTL Low Level	= 0 volts to 0.7 volts

Figure 11

The exact values of the crossover voltages vary with the type of IC used and with the manufacturer, but are typically within a few tenths of a volt of the levels given. Output voltages in the indeterminate range may result in the detecting IC sensing a high or a low, and should be avoided when designing TTL circuits.

Because the interface is implemented in terms of high and low voltage levels and the computer deals with bits (ones and zeros), there are two ways of assigning a correspondence between them. That is, we can assign either high = 1 and low = 0, or high = 0 and low = 1. Both methods are in common use, and the choice of one or the other is usually determined by other design considerations within the computer. Further confusion can arise since these two states are also referred to as true or false. This is the reason that when concepts such as handshake were discussed in the previous sections, we simply referred to the states of the control and flag lines by their logical meanings of set or clear, and ready or busy, without worrying about whether these states were implemented as high or low voltage levels on the interface itself. Indeed, some interface cards allow the user to define whether the ready state of the flag line, for example, will correspond to a high or low level. This places fewer constraints on the design of the peripheral being interfaced and is discussed further in the section on jumpers.

Because these two conventions are in common use, they have been given the names positive-true logic and negative-true logic. The table in Figure 12 shows the meanings of these conventions.

Positive-True Logic:	High = True = 1
	Low = False= 0
Negative-True Logic:	High = False= 0
	Low = True = 1

Figure 12

Thus if the computer placed a bit that was set to a one on a particular data line, this line would be set high in a positive-true system and low in a negative-true system. For example, an ASCII 'E" character (binary value 01000101) placed on the data lines would appear as LHLLLHLH if positive-true logic were being used, and as HLHHHLHL if negative-true logic were being used.

Certain interface processes such as the handshake discussed in the previous section involve several lines changing their states in a definite time sequence. The exact relationship of these lines during the sequence of events is often shown in a graphical representation called a timing diagram. An example of a timing diagram for some of the lines involved in the handshake process for the 98032A interface is shown in Figure 13.

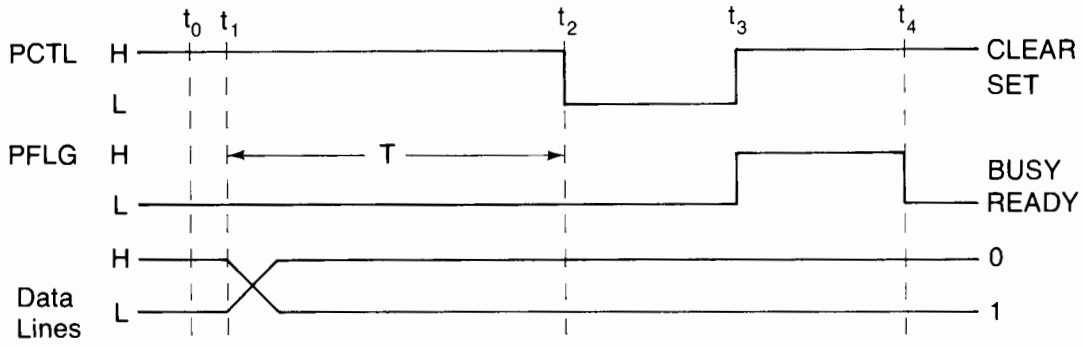


Figure 13. An Example Timing Diagram

In this diagram, time proceeds along the horizontal axis from left to right, and the states (high/low) of the various lines of interest are shown one above the other. A vertical line drawn through the diagram represents the same instant in time for all of the lines. These time points may be indefinite, such as t_0 in the example which shows the state of the lines at some time before the handshake has begun; or they may be definite times such as t_1 which shows the point at which the data on the lines begins to change. Sometimes, the interval between two time points (t_1 and t_2 in the example) is fixed by some requirement of the system, and given a name such as T. In other cases, such as the interval between t_2 and t_3, there is no restriction placed on the time that may elapse between these two events.

The state of the PCTL and PFLG lines in the example are definite (high or low) within each time interval. The handshake timing diagram cannot, however, show the data lines as being either high or low during a given interval, since the state of these lines depends on the data that is being exchanged. In this case, the two parallel (high and low) lines in the diagram simply represent a stable state on the data lines that may be either high or low, while the crossover represents the time during which data on these lines is in a state of transition.

The example timing diagram for an output handshake process would be read as follows. At some time t_0 before a data transfer has begun, the PCTL line is in its normal clear state (high), the PFLG line is ready (low), and the data lines are stable, still containing the last character sent. At time t_1 the interface places the new data on the lines. After allowing a time interval T for the data to become stable, the interface sets PCTL low at t_2 to inform the peripheral that the data on the lines is valid. Longer cables require longer time periods (T) for the data lines to stabilize at their new levels. After an unrestricted time interval, the peripheral acknowledges that it has seen control go set by raising its PFLG line to the busy state at t_3. Upon receiving this acknowledgment, the interface allows its PCTL line to return to the clear state. Finally, when the peripheral has completed processing the information on the data lines, it indicates this fact to the interface by returning its PFLG line to the ready state at t_4. At this time, the PCTL and PFLG lines are back to the same state as they were at t_0 and ready to repeat the entire handshake cycle for the next data transfer.

The complete handshake process also involves the FLG and I/O lines as discussed in the section of Chapter 3 about the 98032A interface. This simplified example is intended merely to present the essential features of timing diagram representations.

Gates, Latches, and Flip-Flops

It would be extremely difficult if not impossible to write useful computer programs without the availability of conditionals such as the "if" statement which allow the program to test some condition and perform one action if the condition is true, and another action if it is false. The operations performed by an interface card also require the use of logic, or the ability to make decisions. The functions of the interface card could be fully described by a flowchart or a computer program. Indeed, if it were not for speed requirements, most of the functions of the interface card could actually be replaced by a computer program. For example, the handshake process described in the last section could be performed by a program which implemented the flowchart shown in Figure 14.

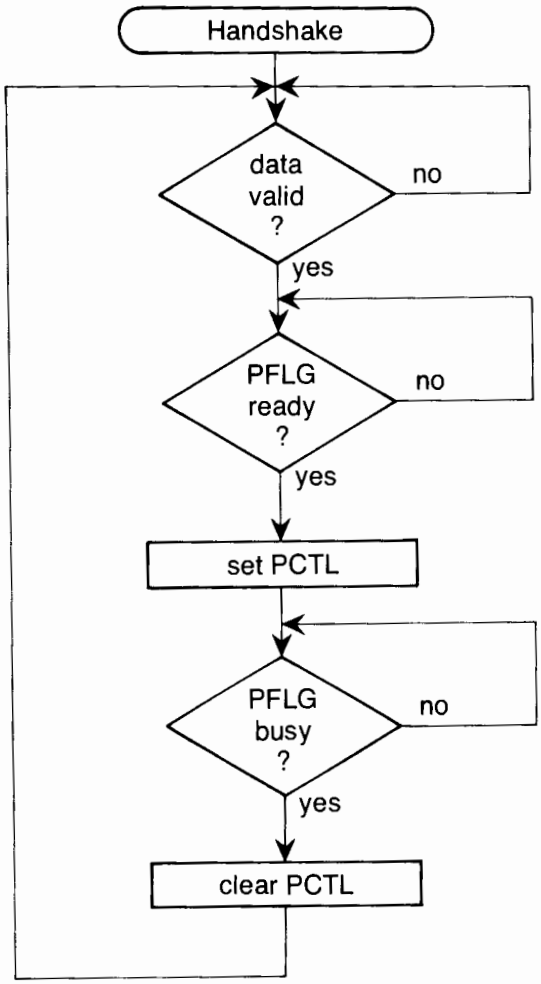


Figure 14

Gates

Since the interface cards are implemented through hardware (electronic circuits) rather than software (computer programs), logic elements called gates are used to perform the required decisions. In reality these gates are made up of very complex electronic networks composed mainly of resistors and transistors. Fortunately, it is not necessary to understand the detailed workings of these circuits in order to present the operational characteristics of these logic elements. Before looking at how these gates are used in the construction of an interface, we will first describe the various types of gates that are available. Figure 15 shows four of the basic logic elements that are used as building blocks for constructing more complex logic elements and implementing conditional operations.

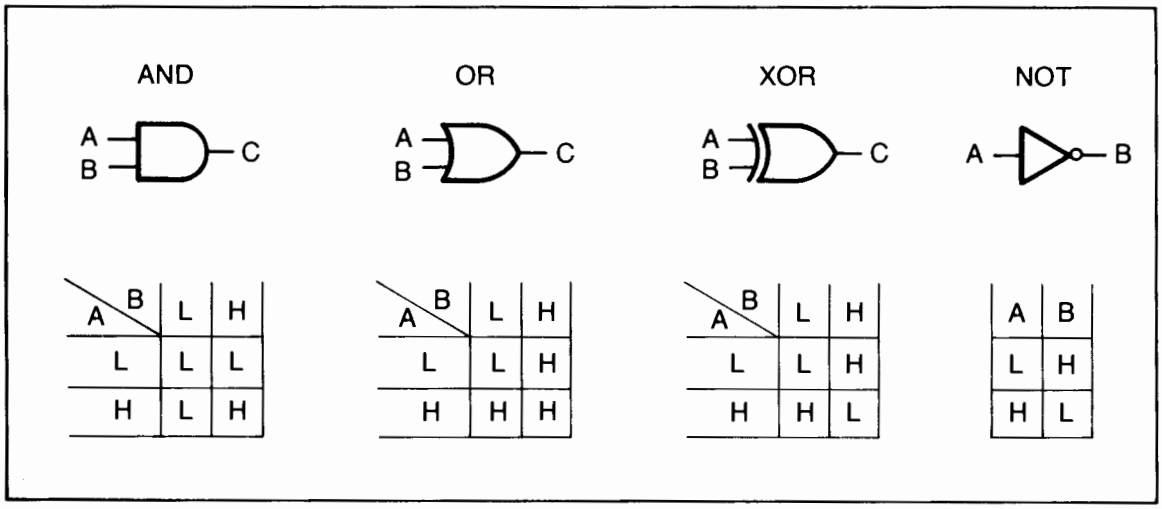


Figure 15

For the AND gate, A and B are the inputs and C is the output. It performs a logic AND function since C is high if and only if both A and B are high, otherwise C is low. This information is presented below the symbol for the AND gate in a form called a truth table. It simply shows the state of the output line for any combination of states of the input lines. In the OR gate (sometimes called an inclusive OR gate), C is high if either A or B is high. In the exclusive-OR gate, C is high if either A or B is high, but not both. And finally the NOT gate, often called an inverter, outputs a high if the input is low, or a low if the input is high.

Several gates of the same type can be obtained in a single integrated circuit package which makes the construction of logic circuits such as an interface card more compact and less costly than if individual components were used. Also available are packages which combine AND, OR, and XOR gates with inverters on their input lines, their output line, or both, leading to a wide variety of combinations. For example, AND gates with inverted outputs are available and are called NOT-AND or simply NAND gates. Figure 16 shows the symbol and truth table for this type of gate.

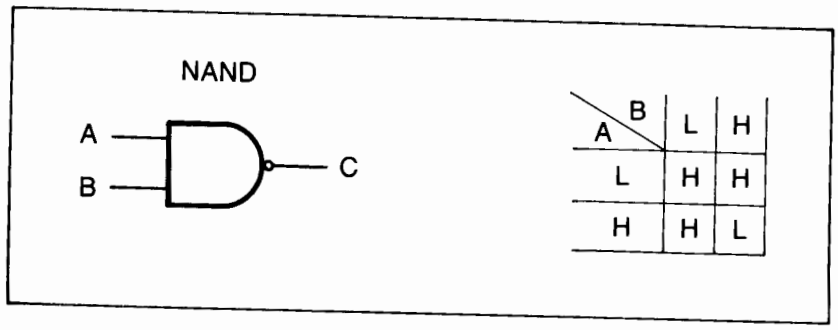


Figure 16

Notice that the truth table is the one that would be obtained by inverting the output of an AND gate. In general, the truth table for any logic element whose input or output lines have circles drawn on them can be obtained from the corresponding table for the element without the circles, and replacing the circles with inverters.

An example will serve to illustrate how these logic elements are used as building blocks in constructing circuits that are capable of making decisions. Let's assume that we have a peripheral that delivers data to the computer at some time after it sees the control line (PCTL) go set. Normally, this operation is automatic so that as soon as control is set, the device responds by issuing a pulse (low to high and back to low transition) on a line from the peripheral called READY. This READY line would usually be connected directly to the PFLG line on the interface to complete the handshake. But we would like to have an alternate mode of operation, established by the computer program, that would allow an operator at the device to signal the ready response by pressing a button. The circuit shown in Figure 17 could accomplish this task, making use of the logic elements that we have been discussing.

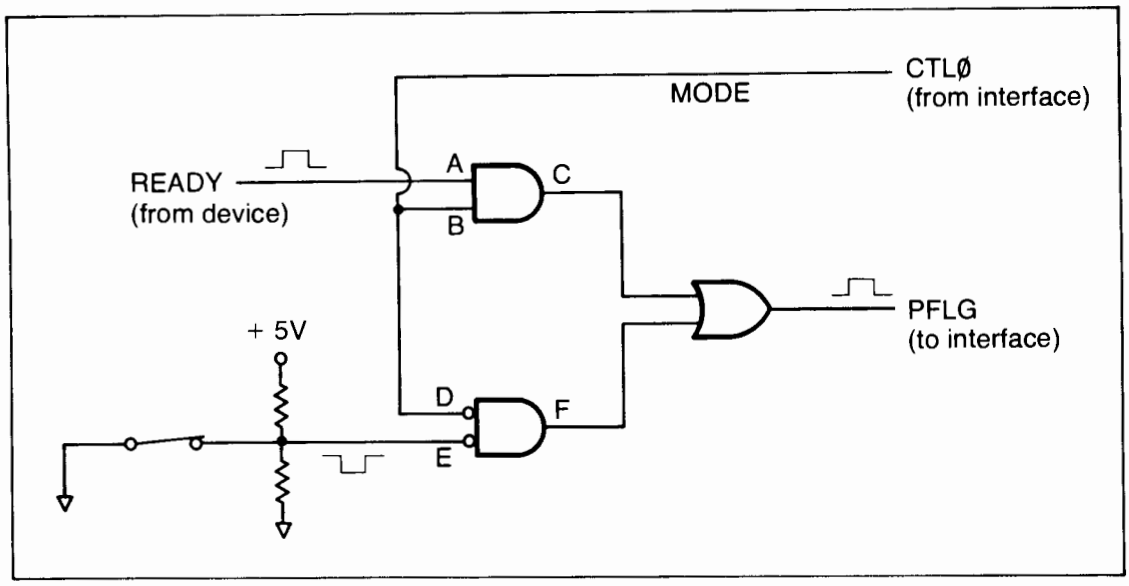


Figure 17. An Example Use of Logic Elements

We first define a MODE line which determines the mode of operation: automatic when it is high, and operator-controlled when it is low. This line is connected to one of the general purpose control bits (CTL0 on the 98032A card) so that it may be set to the desired mode of operation by the program. When MODE is low, the input to the AND gate at pin B is low, so that no matter whether A is high or low, the output at C is low. That is, when MODE is low (operator control) the READY pulse is blocked by the AND gate and C remains low. When MODE is high, C is high only when A is also high, so that the positive pulse is now passed by the AND gate and appears at the output C.

The second AND gate controls the signal from the operator's push button. When the switch is open, the pull-up resistors hold the input E high (see Figure 10). Pressing the button grounds the input E and causes it to go low, returning again to high when the button is released. (In actual use, a debounced switch should be used to prevent multiple pulses.) Since the input E is an inverted input, this switch presents a positive pulse to the AND gate when it is activated. Input D operates in the same way as input B did for the READY pulse, to either block or pass the signal from input E. But since it is an inverted input, it passes the signal when MODE is low and blocks it when MODE is high. Thus, either the READY pulse or the one provided by the push button will appear at C or F, while the other line remains low. If these two lines are connected to the PFLG line through an OR gate, one and only one of the pulses will drive PFLG, depending on the state of the MODE line.

Thus, this configuration of logic gates implements the function stated by: if mode is automatic, pass the READY pulse to the PFLG line and block the pulse generated by the operator; if mode is manual, pass the operator's pulse and block the READY signal. Again we see that gates are used to provide a hardware implementation of a function that could be expressed by a logical flow diagram.

Latches

If the data output lines from the computer were directly connected to the data input lines to the peripheral, then during the handshake process it would be the responsibility of the computer to maintain the data on the lines until the peripheral had acknowledged that the data had been accepted. Normally this would cause no problem since the computer is merely waiting anyway for that acknowledgment so that it can put the next data item on the lines. But if only one character is being sent, the computer could go on with the program if it did not have to stay in the output driver to hold the data on the lines. This becomes more important, even essential, when operating under interrupt. In the interrupt mode, the computer places the first character of the output message on the data lines, tells the interface to generate an interrupt when the peripheral has taken that character and the next one can be sent, and then goes on with program execution instead of waiting for the handshake process to complete. This would not be possible if the computer had to maintain the data on the lines.

Therefore, one of the functions of the interface is to be able to hold or latch the information on the data lines until the peripheral has had a chance to take it, and thus relieve the computer of this responsibility. In the hardware of the interface, this is accomplished through an electronic device called a latch (Figure 18). Typically a latch has a number of input lines and a corresponding number of output lines, plus an additional line usually called the clock line. When the clock line is activated, whatever data currently is being presented on the input lines is held by the latch and presented on the output lines. Then when the clock line is deactivated, this same data is maintained on the output lines. The way in which the clock line is activated (i.e., positive pulse, negative pulse, low to high transition, etc.) depends on the particular type of latch being used, and need not concern us here.

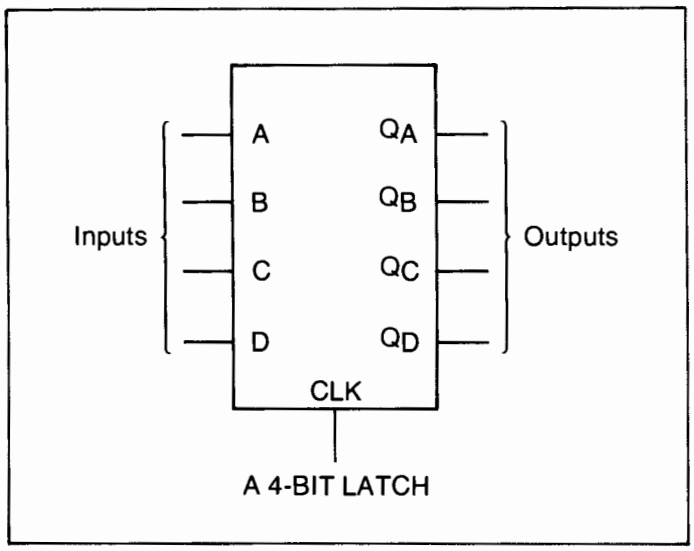


Figure 18

Chips are available which provide latching for four bits of data on a single integrated circuit package. Thus, to provide latching for the 16-bit output data bus, the 98032A interface uses four of these 4-bit latches. Four more are used for the 16 data input lines. These input latches, in a manner similar to the output operation, relieve the peripheral of the responsibility of maintaining the data on the interface input lines until the computer has had a chance to take it.

These latches are sometimes referred to as one-character buffers, and should not be confused with the buffers described in later sections dealing with the transfer of interrupt buffers. These interrupt buffers are multi-character holding locations that are located in the read/write memory of the computer itself.

Flip-flops

A flip-flop is a device that is similar to a one-bit latch, but with more extensive control properties. There are many different types of flip-flops each designed to satisfy a different set of requirements. Figure 19 shows a schematic representation for one common type called a D-type flip-flop.

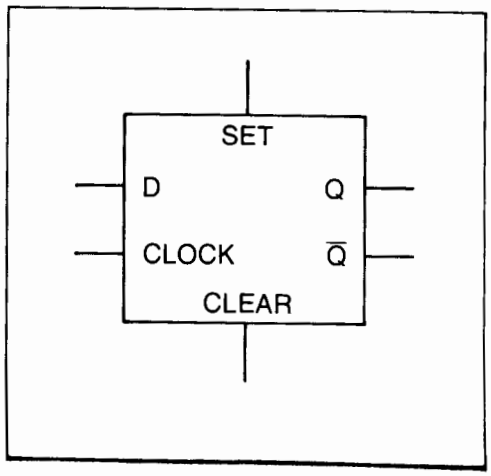


Figure 19

When the clock line is activated, the current state of the D input is latched and presented at the Q output line. On deactivating the clock signal, Q will hold its state independently of what happens on the D input. For convenience in designing logic networks using these flip-flops, an inverted output, $\overline{Q}$, is also provided. That is, Q is always in the opposite state from that of $\overline{Q}$.

Two additional lines are provided to set Q to the high state or to clear Q to the low state, independently from the clock and D lines. These set and clear lines are often used to initialize the flip-flop to the desired "wake up" state.

Just as the latches were used to maintain information on the data lines, flip-flops are used to allow the interface to "remember" information about what state it is currently operating in. For example, we will see later that the computer will send a particular message to the interface card to tell it that it is enabled to operate in the interrupt mode. The card remains in this mode until it is disabled by another message from the computer. In the meantime, it remembers which mode it is in (enabled or disabled) by storing that information in one of these flip-flops.

The Use of Jumpers

In the last section we saw that flip-flops could be used to change various modes of operation on the interface by programmable signals from the computer. For example, the interrupt-enable flip-flop could be turned on and off by the computer at will. Other modes of operation are a property of the system itself and do not change during the running of a program. It would be more convenient if these modes could be set one time on the card itself, and then the program would not have to be concerned with them.

As an example of this, consider the handshake diagram of Figure 13. The meaning assigned to the PFLG line from the peripheral is high = busy and low = ready. We might want to interface a device, however, whose handshake line used the opposite sense; that is, high = ready and low = busy. The 98032A card provides for such an inverted sense by installing a jumper (i.e., a wire connecting two terminal points in the circuit) on the interface itself. Figure 20 shows how the use of this jumper accomplishes the desired inversion of the PFLG signal.

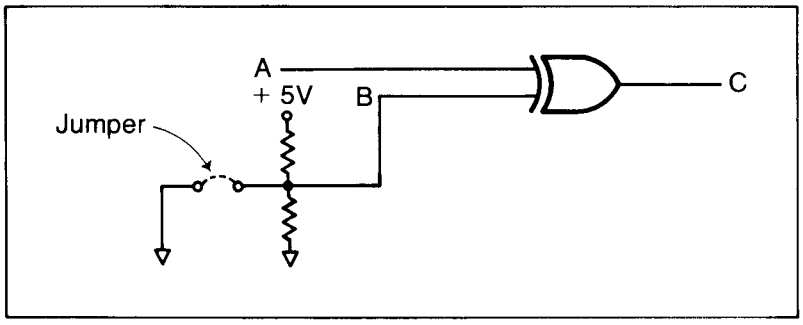


Figure 20

The properties of the exclusive-OR gate used were given in Section IC2. Its output, C, is high if either A or B is high but not both. If the jumper is not installed, the resistive divider holds the B input high. Looking at the truth table for an exclusive-OR gate, we see that in this case if A is high, C is low; and if A is low, C is high. Thus, the signal presented at the A input appears inverted at the C output. If we now put in the jumper, the B input is connected to ground (low) and the state of C is always the same as the state of A. As a result, the signal seen at C is either the same as A or the complement of A depending on whether the jumper is in or out.

It should be noted that in the particular example used, (i.e., the PFLG line on the 98032A card), this line has already gone through a separate inverter gate before arriving at the A input line in our diagram. As a result, the PFLG line itself is inverted when the jumper is installed, and not inverted when the jumper is out.

Other jumpers may be used in an entirely different way from the example just given. For instance, we will see later that the data latches on the 98032A interface are divided into two groups of eight. In the so-called bytes mode these two groups of latches can be clocked separately, whereas in the words (16-bit) mode they are clocked together at one time. The jumper which selects the word mode simply connects the two clock signals for these latches together.

Finally, we should mention that the use of jumpers provides a means of making these connections in a manner that is most economical of space on the interface card. On other cards where room is available, miniature slide switches may be used to achieve the same result. Also, switches are used instead of jumpers where the user might want to change the mode of operation based on the particular application. In any case, these switches and jumpers are used to select modes that will not be required to change during the running of a particular program.

The installation and service manuals for each interface card go into more detail on the switches and jumpers provided by each card, and their intended uses. The purpose of this section is merely to give the reader some idea of how a jumper or switch can be used to perform the functions described in those manuals.

Chapter 2
Programming for Interface Operations

Standard I/O Programming

A Register Operational Model of an Interface

An interface is a complex electronic circuit that provides mechanical, electrical, data format, and timing compatibility between a computer and the peripheral device to which it is connected. From a programmer's point of view, however, the primary task of interfacing is to provide a means of exchanging data between the computer and the peripheral. Thus, a well-designed interface should isolate the programmer from the details of the electronics and timing, and appear as a simple "black box" whose I/O characteristics can be presented in a simple model and described by a set of operational rules.

In Chapter 3, we will look at the various interfaces provided for HP desktop computers from a hardware point of view and cover some of the special characteristics of each of them. In this section, it will be sufficient to model the interface as a set of four registers through which all the capabilities of the card can be accessed. These four registers are given in Figure 21.

	IN	OUT
R4	PRIMARY DATA IN	PRIMARY DATA OUT
R5	STATUS	CONTROL
R6	SECONDARY DATA IN	SECONDARY DATA OUT
R7	SECONDARY STATUS	SECONDARY CONTROL

Figure 21

The names of the four registers (R4, R5, R6, and R7) are simply names given to four address locations in the computer memory map. These registers should be thought of as residing on the interface card itself.

The computer sees these interface registers as 16-bit, binary registers, and always sends and receives 16-bit binary words when addressing them. If a particular interface utilizes less than the full 16-bits (for example, when exchanging 8-bit ASCII data bytes) then on input the upper (more significant) bits are received as zeros. On output to these registers, if fewer than 16 bits are utilized by the interface, it ignores the upper bits. Thus, these bits may be ones or zeros and are sometimes called "don't care" bits.

All of the interface cards use the R4 register for data I/O operations, and the R5 register for status and control information. The names given in the table above for the R6 and R7 registers are only general indicators of the functions of these registers. Their exact interpretation varies with each card and is described in more detail in the sections on the individual interfaces.

In order to give specific examples of the use of these I/O registers, we will use the meanings given to them by the 98032A Bit Parallel Interface, sometimes called the GPIO (General Purpose Input Output) Interface. It defines these registers as follows.

	IN	OUT
R4	DATA IN	DATA OUT
R5	STATUS	CONTROL
R6	HIGH BYTE DATA	HIGH BYTE DATA
R7	(not used)	TRIGGER

Figure 22

The GPIO uses the R6 register in a special way when operating in the optional "byte mode" as described in the Chapter 3. For our purposes here, the R4 register is the one through which all data is transmitted and received. We will give examples below of how these registers are used to do simple input/output operations.

Select Codes

As mentioned earlier, a set of I/O registers R4-R7 exist on each interface card. When more than one card is connected to the computer and, for example, an R4-in operation is performed, we need a mechanism for determining which interface should respond. This is accomplished by means of a 4-bit register in the computer called the Peripheral Address (or simply PA) register. This PA register holds a binary number in the range of 0 to 15, thus allowing for up to 16 interfaces to be addressed. Each interface has an externally-settable select code switch which can also be set to any value between zero and 15. (You should note, however, that several select codes are for internal interfaces and should not be set as a select code for an interface card.) Whenever an operation to one of the interface registers is performed, the computer presents the current contents of the PA register to all of the interface cards simultaneously. Only that card whose select code matches the PA register will respond to the operation.

When a BASIC I/O statement such as OUTPUT 6; A, B, C or STATUS 6; A is executed, the I/O ROM automatically puts the binary value of the select code parameter (in this case, 0110 = 6) in the PA register before addressing the required interface registers.

Direct Register Access

All interface card operations are carried out by sequences of operations to and from the interface registers. The more common tasks (reading and writing data, checking status and setting control bits, etc.) have been provided at the BASIC programming level by simple statements and functions such as OUTPUT, ENTER, STATUS, etc. These high-level statements isolate the programmer from the details of the register sequences required to perform each task.

In the event that the programmer should wish to perform some sequence of operations other than those provided by the BASIC language, additional statements have been provided that give the BASIC program direct access to the interface registers.

These are the write-interface-register statement and the read-interface-register statement, whose syntaxes are given below.

WRITE IO <select code>,<register number>;<output>
READ IO<select code>,<register number>;<variable>

The write-interface statement outputs a 16-bit, 2's complement representation of the value specified by the <output> parameter to the register specified. The read-interface statement inputs a 16-bit 2's-complement binary value from the specified register and returns its decimal equivalent in the statement variable.

The register number given should be in the range of 4 to 7.

In the following section, we will see how this direct register addressing works by reducing familiar operations such as writing data and reading status to their equivalent register sequences.

Before doing this, there are two additional lines to the interface required to complete the functional description of the card. These may be considered as 1-bit, read-only registers called status (STS) and flag (FLG).

The status bit (not to be confused with the status register, R5 to be discussed later) is a single bit indicator that the interface and the peripheral connected to it is operational. For example, if a peripheral device has a line coming from it that indicates power on, it could be connected to the STS line. Then the program could quickly determine whether the device is turned on or off. Or as another example, a printer might have the STS line connected to its out-of-paper indicator (if it has one) to indicate to the program that it is no longer operational when the paper runs out.

The flag line is a momentary ready/busy indicator used to keep the computer from getting ahead of the peripheral. The use of this line is covered in more detail in "The Handshake Process" section of Chapter 3. For our purposes here, it is sufficient to know that on the flag line, a one indicates ready and a zero indicates busy. For example, if the computer had a sequence of ASCII characters to send to a slow printer, it would send one character (which makes the flag line go busy) and then wait for the flag line to go ready again before sending the next character.

These FLG and STS lines may be tested from the BASIC program by using the following functions.

```
Variable = IOFLAG(<select code>)
Variable = IOSTATUS(<select code>)
```

These functions return a one or a zero indicating the current state of the FLG or the STS line.

The Read and Write-interface statements and FLG-test function are usually used together to prevent accessing an interface which is not ready, as shown below:

```
Self: IF NOT IOFLAG (<select code>) THEN Self
      WRITE IO <select code>,<register number>;value
```

The above two program lines can be combined by using the WAIT WRITE statement, and its companion WAIT READ statement can be used for input. The syntax for each is

```
WAIT WRITE <select code>,<register number>;<value>
WAIT READ <select code>,<register number>;<variable>
```

The Status and Control Registers

The primary purpose of the interface is to allow data to be exchanged between the computer and the peripheral device to which it is connected. HP's 98000 series interface cards are extremely versatile, however, and most of them are programmable. This means that they have various optional capabilities that can be set and changed by control instructions from the computer. Referring to the register model of the interface, this programming is done by outputting specific bit patterns to the R5 register. Some of the interfaces use other registers for extended control bits and these are described in the sections covering the specific interface cards.

The R5-out control register is usually addressed from the BASIC programming language using the CONTROL MASK and WAIT WRITE statements combined.[1] For example, the statements

```
CONTROL MASK 6,9
WAIT WRITE 6,5,9
```

would output the binary equivalent (00001001) of a decimal 9 to the R5 register of the interface set to select code 6, causing bits 0 and 3 of that register to be set. The effect of setting those bits is determined by which type of interface is involved and the meaning of each of the control bits is decribed in the section on the individual interfaces.

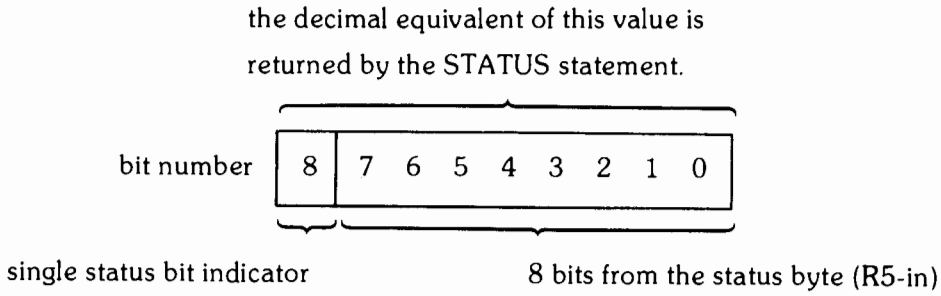
1 See "The Use of the Control Register" section for an explanation of the CONTROL MASK-WAIT WRITE statements combination.

The interface cards can also return information to the computer telling which optional programming features are currently selected. This information, called the status byte, is obtained from the interface through an R5-in operation. This status byte consists of 8 bits whose meanings are determined by the particular interface card that is being addressed. These bit assignments are explained in detail in the sections on the individual cards.

At the BASIC level, this status byte is obtained by using the read-status (STATUS) statement. For example, the statement STATUS 6;A would perform an R5-in operation on the interface set to select code 6 and return the decimal equivalent of the binary bit pattern that it received.

This status byte should not be confused with the single status bit described earlier. That status bit is merely a 1-bit, quickly testable indicator of whether or not the card is functionally operational whereas the status byte contains up to eight bits of information about the current programming configuration of the card.

Whenever a STATUS statement is performed, the 8 bits of status are returned plus an artificial 9th bit that represents the single status bit (as returned by the IOSTATUS function).



Thus a functionally operational interface should return a value for the STATUS statement greater than 256 ($= 2^8$ = status bit set). If the value of the status byte is read using the WAIT READ statement directly from the R5-in register, only the 8 bits of that register are returned.

Binary I/O Operations

We are now in a position to look at the sequence of events that takes place between the computer and the interface card when simple I/O operations are carried out. In particular, we will simulate the actions of the WRITE BIN and the READ BIN statements through the use of the direct interface access statements and functions explained in the last section.

When the I/O ROM performs these operations, there is a considerable amount of checking and internal "bookkeeping" that goes on to insure that systems conflicts are avoided. Here we will only be concerned with the basic communication between the computer and the interface. We will also look at this communication from the point of view of the register operational model of the interface as described earlier. In the sections on the interface cards themselves we will look at more detail about what actually takes place on the card.

The following BASIC program simulates the actions that take place when the statement WRITE BIN 6;27 is executed to send the binary value 27 to a device on select code 6.

```
10   IF NOT IOSTATUS(6) THEN Card_down
20    IF NOT IOFLAG(6) THEN 20          !  Can be replaced with
30    WRITE IO 6,4;27                   !  "WAIT WRITE 6,4;27"
40   WRITE IO 6,7;0
```

First, the status bit is tested to make sure that the device is operational. If it is not, we branch to the "card-down" routine, which in the I/O ROM would issue an error 167. If the device is operational, we then loop until the flag line indicates ready. The data to be sent is then placed in the R4 output register. This merely places the data in the output latches on the interface but no output operation to the device has taken place yet. In the last line, the output to the R7 register actually triggers the data transfer to the device. (Note that the actual value sent to the R7 register, a zero in this example, does not matter. Only the R7 out operation itself is sensed by the interface as the trigger command.) If more data were to be sent to the same device, we would repeat lines 10, 20, 30, and 40 for each data item. It is important that each time through this loop we wait for the flag to indicate ready. If the flag is indicating busy, the last data item is still in the output latches and has not yet been taken by the device. If we were to execute line 30 in this state, the new data would overwrite the old data in the latches and the old data item would be lost.

we can also use the direct register operations to simulate the input process. The following BASIC program performs the same operations as when the function A = READBIN(6) is executed to read a binary value from the device on select code 6 and assign it to the variable A.

```
10   IF NOT IOSTATUS(6) THEN Card_down
20    IF NOT IOFLAG(6) THEN 20          !  Can be replaced with
30    READ IO 6,4;A                     !  "WAIT READ 6,4;A"
40   WRITE IO 6,7;0
50    IF NOT IOFLAG(6) THEN 50          !  Can be replaced with
60    READ IO 6,4;A                     !  "WAIT READ 6,4;A"
```

The first two lines of this routine are the same as in the previous simulation of the WRITEBIN statement, and serve the same purpose. The R4 in operation in line 30 merely tells the interface that an input operation is to be performed. When the trigger (R7 out) is done in line 40, the card goes busy and demands a data item from the peripheral device. Line 50 waits for the interface to latch the data from the peripheral, and line 60 takes the data from the interface and places its decimal representation in the variable A. If more data items were to be input, lines 40 through 60 would be repeated. Notice that the interface has only one trigger (R7 out) register, which is used for triggering both input and output operations. The function to be performed is determined by whether the last data operation between the computer and the interface was an input or an output. This is why the "dummy" input operation in line 30 is required.

Normally, the user would not get involved with the specific sequence of events that take place when a simple WRITEBIN or READBIN operation is performed. These sequences were presented here merely as an example of the use of the interface registers. Other examples will be presented later that require the use of the WRITEIO and READIO instructions to accomplish certain tasks. Also, understanding this register model of the interface will be helpful in describing the events that take place during interrupt operations in the section on Data Transfers with Slow Devices.

Formatted I/O Operations

Strictly speaking, all simple data input/output operations could be performed with only the use of the write binary (WRITEBIN) and the read binary (READBIN) instructions. For example, if we wanted to output the value of pi (3.1415926536) to a printer, we could calculate each of the digits and send its ASCII code to the printer one at a time with the WRITEBIN statement, taking care to output the ASCII code for a decimal point at the proper place in the sequence. In practice, however, it is much easier to simply specify the value that we wish printed and let the I/O ROM perform the task of breaking it up into the proper sequence of ASCII characters. As a result, most simple data I/O is done using the ENTER and OUTPUT statements.

These statements are even more powerful since they work in conjunction with the IMAGE statement. This formatting capability allows the program to specify the exact form into which data should be put for output operations, and the sequence of characters that is expected from an input operation. The use of the ENTER, OUTPUT, and IMAGE capabilities is discussed with several examples in the programming manual for the I/O ROM.

Since most simple data I/O is done using the ENTER and OUTPUT statements, the question arises of when the READBIN/WRITEBIN instructions should be used in writing a program. In the examples that follow, it is assumed that the reader is familiar with the material presented in Chapter 2 of the I/O ROM Programming manual. If not, it would be helpful to read that section before continuing here.

Example: Non-ASCII Characters

The OUTPUT statement accepts three kinds of parameters: strings, numerics, and arrays. A string is a sequence of ASCII characters enclosed in quotes, such as "The value of pi is". A numeric parameter simply specifies a constant or a program variable whose value is to be converted to a sequence of ASCII characters to be sent. An array parameter specifies all or part of an array variable. The table of ASCII codes (see Appendix A), however, also provides what are called control characters. Some of these were designed specifically for controlling printing devices, such as carriage-return, line-feed, vertical tab, form-feed, etc. Others are used in special applications such as block transfers (start of text, end of text, etc.) and data communications (enquire, acknowledge, synchronize, etc.). Because of their effect on program listings, we normally use the "B" image specifier or the WRITEBIN statement to send them. For example, we might have a printer that uses paper that is perforated into pages. After sending some lines of output, we want the printer to skip to the top of the next page. In the ASCII character set, this instruction, called form-feed, has a value of 12. Thus, the statement WRITEBIN 6;12 would send an ASCII form-feed character to the printer on select code 6. It is important to note that this operation sends only a single ASCII character whose binary value is equivalent to a decimal 12, and we would see the printer skip to the top of the next page (assuming that the printer has this capability). Whereas if we had issued the statement OUTPUT 6;12, the ASCII characters for a printing "1" (decimal code 49) and a printing "2" (decimal code 50) would have been sent, along with some number of spaces, a carriage-return, and a line-feed, and we would have seen the actual value of "12" printed.

Example: Suppression of CR/LF

Another distinction between the OUTPUT and WRITEBIN statements is that the WRITEBIN statement outputs only the values specified, while the OUTPUT statement generates character sequences specified by the IMAGE statement, often giving unexpected results. For example, the HP 2640A is a teletype-like terminal that has a CRT (video display), and can be put into inverse video (dark characters on a light background). To put the 2640 into this inverse video mode, an escape character (decimal 27 ASCII code) followed by the ASCII characters "&dB" must be sent. Using the WRITEBIN statement, this could be done by executing the statement WRITEBIN 6;27,38,100,66. If instead we had executed

```
WRITEBIN 6;27
OUTPUT 6;"&dB"
```

we would successfully put the 2640 into inverse video, but the automatic CR/LF generated by the OUTPUT statement would have moved the cursor on the CRT to the start of the next line.

Example: Debugging Tools

Finally, the READBIN/WRITEBIN instructions are often valuable in debugging programs. For example, suppose we have a device that sends us a numeric value 12.345, and we execute the statement ENTER 3;A. When we execute this statement, we notice that the run light on the computer stays on and reading does not complete. We don't know whether we are not getting any data at all or if something else is wrong, perhaps a hardware failure. So we execute a series of READBIN(3) instructions and note the results.

READBIN number:	1	2	3	4	5	6	7	8
value returned:	49	50	46	51	52	53	13	busy
ASCII:	1	2	.	3	4	5	CR	

This tells us that the device is indeed sending what we expected to see and there is no hardware problem. However, we notice that no line-feed has been received. Since this is required for the ENTER statement to complete, we must use an IMAGE specifier to disable the line-feed requirement. We change the ENTER statement to read:

ENTER 3 USING "#,F";A

Now the ENTER completes, and we get the expected value. Many times this same technique of using READBIN instructions to look at the incoming data stream one character at a time will reveal that the input sequence is something other than what is expected, and the ENTER and IMAGE statements must be adjusted accordingly.

There can arise some confusion from the broad range of ENTER image specifiers, and their interactions. There are two primary categories of image specifiers, those that specify ENTER **statement terminating conditions**, and those that specify **variable type** (and these also determine the item terminating condition). Terminating condition specifiers determine when the ENTER statement itself can terminate, but it is important to remember that termination cannot occur before all variables in the ENTER list have been satisfied. When the terminating character for the ENTER statement (L/F for example) is the same as the required terminating character for the last item in the ENTER list, the termination of the statement and the variable can occur simultaneously. For instance, satisfying a string input with an L/F will also satisfy the ENTER statement's L/F requirement, (if the string is the last item in the list), i.e., ENTER 6;A$. Variable type specifiers determine how the incoming stream of data is to be interpreted and placed in the ENTER list variables.

terminated because a L/F was not necessary to complete the ENTER statement requirements. Satisfying the input to variable "A" was enough. Incidentally, the non-numeric character, C/R, completed the input operation for the variable "A," and thus the entire ENTER statement as well.

The "+" and "%" specifiers are intended for use with HP-IB transfers, where the EOI message (see "The 98034A Interface") is used by certain devices to signal the end of a data transfer. The "+" specifier causes EOI to be **ignored** as a terminator. The "%" specifier causes EOI to be ignored, **and** removes the normal L/F requirement for terminating the ENTER statement.

The variable type specifiers are described in the I/O ROM programming manual, and will not be covered in detail here. Rather, we will go through some examples of determining the image specifiers needed from a description of the incoming data.

- A terminal is connected to a 98036A interface, and we want to enter the data into a string variable. Each line of data is terminated by a C/R:

Incoming line of data — "This is any string of characters" C/R

Appropriate statement — ENTER 11 USING "T";TRL(13),A$

The "T" specifier allows a varying number of characters (up to the size of A$) to be entered into the string, until a **delimiter** is received; the TRL function allows our string input to terminate with a C/R as well as a L/F delimiter.

- The same terminal is connected to our 98036A interface, but the operator occasionally hits the Line-feed key instead of Return. We don't want our input to terminate when he hits the L/F — only the C/R is considered an end-of-record.

Incoming data line — "This string has" L/F "characters" C/R

Appropriate statement — ENTER 11 NOFORMAT;TRL(13),A$

This is a sneaky solution, but the NOFORMAT mode normally requires that the string be filled or that a specified number of characters be input. The TRL function allows us to terminate on a specified character value, in this case C/R(= decimal 13). Now our string input is **not** terminated by a L/F, but will enter everything coming in until a C/R is received, or the string is filled.

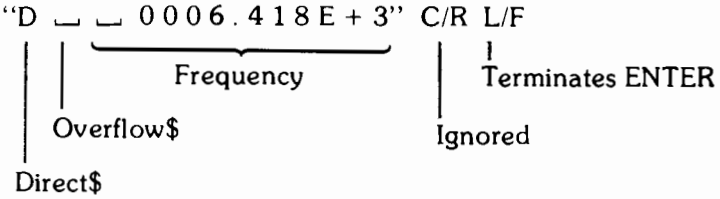
- A 5340A HP-IB Frequency Counter sends its data in a combined form of non-numeric and numeric characters. A set number of characters is sent each reading, and a typical reading might look like this:

"D ␣ ␣ 0006.418E+3" C/R L/F

Appropriate statement —

ENTER 716 USING "2(A),12N";Direct$,Overflow$,Frequency

The "2(A)" specifier directs the I/O ROM to take one character each for two string variables. The parentheses around the "A" distinguish this from the two-character input "2A". The "12N" specifier is used here because the number of data characters is fixed and known, although the "F" specifier could also have been used successfully. The data is put into the variables as shown below:



The novice programmer, and even experienced programmers, can expect some amount of trial-and-error when debugging formatted input statements. The wide range of devices available, and the near-infinite variety of data formats makes it difficult to correctly read incoming data streams on the first try; especially those with complex combinations of numeric and string data.

The following table is organized so the programmer can identify the correct image specifiers, given a knowledge of the characteristics of the incoming data.

Type of Data	Charac-teristics	Image Specifier	Condition Required to Terminate	Notes
String	Unknown or varying number of characters	"T"	L/F, C/R L/F. TRL, EOI, or string full.	Use TRL function to change L/F terminator to some other character, such as C/R. The "+" and "%" specifiers prevent EOI from terminating string input on HP-IB.
	Fixed number of characters (= n)	"{n}A"	Receipt of {n} characters.	Enters any and all characters until {n} characters have been input.
Numeric	Unknown or varying number of digits per number	"F" (American) or "H" (European)	TRL, EOI, or any non-numeric character.	The "F" specifier recognizes "." radix, while "H" recognizes "," radix. The "+" and "%" specifiers prevent EOI from terminating numeric input on HP-IB.
	Fixed number of digits per number (= n)	"{n}N" (American) or "{n}G" (European)	Receipt of {n} characters.	The "N" specifier recognizes "." radix, while "G" recognizes "," radix. All characters, even non-numeric, are counted as part of {n}.

Continued

Type of Data	Characteristics	Image Specifier	Condition Required to Terminate	Notes
Binary	Byte (8 bits)	"B"	Receipt of one data byte (8 bits).	If a 16 bit interface is used, only the lower 8 bits are taken to satisfy the "B" specifier.
	Word (16 bits)	"W" or "Y"	Receipt of one data word (16 bits).	On an 8 bit interface, first byte received becomes most significant 8 bits of word.
All	Skip over {n} characters	"{n}X"	Receipt of {n} characters.	
	Skip over {n} records, delimited by L/F	"{n}/"	Receipt of {n} L/Fs.	
ENTER statement	Requires all <enter list> variables to be satisfied, then a terminating character which can be modified as shown.	none	L/F,EOI, or TRL	At least one is required, and it may be same as last item terminator.
		"#"	ENTER list satisfied	Statement terminates when ENTER list input is complete.
		"+"	L/F or TRL	EOI is ignored, but L/F or TRL is still required (may be same as last item terminator).
		"%"	ENTER list satisfied	EOI is ignored, and statement terminates when ENTER list input is complete.

Interrupt I/O Programming

The Uses of Interrupt

A computer which has the ability to operate under interrupt provides the user with additional programming features that fall into two main categories. One of these is the optimization of data transfer operations in which the speed of the computer can be more closely matched to the speed of the peripheral device. The other is the ability to have a particular segment of the program in the computer executed at a time determined by the external device. The first of these abilities will be discussed here while the second will be covered in the section on User Programmed Service Routines.

In Figure 23, peripheral devices have been classified as slow, medium, and fast depending on the rate at which they are capable of transferring data.

Speed:	Slow Devices	Medium-speed Devices	Fast Devices
Examples:	Paper Tape Readers Card Readers Teletypes Plotters Digitizers	Thermal Printers Medium-speed DVM's	High-speed DVM's Magnetic Tapes A/D Converters Discs
Transfer Rates:	Below 1000 characters per second	1000 to 10000 characters per second	Above 10000 characters per second
Without Interrupt:	wait	read/write	– – –
With Interrupt:	interrupt	read/write	fast read/write DMA

Figure 23

Although some devices clearly fall into one category or another, this division should not be considered rigid, and the transfer rates in the table are intended to provide rough boundaries. In general, the way in which the device is being used in a particular application, rather than its maximum transfer rate, will determine the category to which it belongs in that application. For example, the 9876B Thermal Line Printer can accept characters at a rate of about 40,000 per second until a line-feed is received. It then requires 125ms (⅛ of a second) to print that line and be ready to accept further characters. Other devices, like digitizers, are totally time random. That is, the rate at which data is available may depend on an operator pressing the sample button on the digitizer.

In a computer without interrupt capability, data transfer is done via the normal ENTER and OUTPUT operations. These operations will have their own "natural" (i.e., computer imposed) speed limitations depending on such things as how much data is to be transferred, the type of data (numeric, strings, binary), and how much formatting has to be done on the data. Depending on these factors, the natural ENTER/OUTPUT speed of the 9835/45 can be anywhere between 1000 characters per second to well over 10000 characters per second.

If the speed of the peripheral device is slower than this natural read/write speed of the computer, the I/O ROM will simply wait after it has transferred one character until the device indicates that it is again ready before it sends or receives the next character. If, on the other hand, the peripheral's speed is faster than the natural read/write speed of the computer, the peripheral device itself will have to wait between each character until the computer is ready for the next transfer operation. If the peripheral cannot wait (i.e., slow down to the computer's natural read/write speed) then the data simply cannot be transferred between the computer and the device.

This situation can be greatly improved if the computer can support transfer mechanisms other than the normal ENTER/OUTPUT operations. For slow devices, we would like to have the ability to transfer one character to or from the peripheral device, and then be able to "go away" and perform other useful work while we are waiting for the device to come ready for the next character. For fast devices, we would like to be able to separate the formatting process from the actual transfer process, and thus increase the natural speed of the transfer process alone. For example, if we wanted to take 100 readings from a high-speed digital voltmeter using a programmed loop and normal ENTER statements, each reading would have to be formatted, put into internal floating-point representation, and stored in the specified programming variable before we could be ready for the next reading. It is this "overhead" that determines the natural speed of the ENTER operation. If, however, we had a means of merely collecting the "raw data" (the exact sequence of bytes or words that came in from the DVM) and could then go back at a later time and do the formatting and conversion into internal numeric representation, this simple data gathering process could proceed at a much higher rate.

The 9835/45 provides an interrupt mechanism for handling slow devices, as well as Fast-Handshake and DMA for handling fast devices. The use of the interrupt capabilities is discussed in the following sections, and fast read/write and DMA transfers are discussed in the High Speed I/O Programming sections.

Before we go on, lets take a look at the interrupt structures of the System 35/45 computers. There are two fundamental categories into which interrupts are classified, with each category being further divided into subgroups, as shown on the next page:

Category I: Hardware Interrupts

- These interrupts are absolutely transparent to the user program, and cannot be accessed.

A. Data Transfer Interrupts
These interrupts are used to signal "interface ready" during a data transfer.

B. Other Interrupts
These interrupts are used generally to signal some external event's occurrence.

Category II: Software Interrupts, or End-of-Line Branches

- These "interrupts" are set up by the ON INT statement to handle real-time events with a user program.

A. End-of-Data Transfer Branch (ENTER/OUTPUT)
This branch indicates to the program that a data transfer has completed.

B. Timeout Branch (SET TIMEOUT)
This branch indicates that a data transfer has waited too long for a new handshake, and that the transfer is probably hung up.

C. Data Transfer Branch (CARD ENABLE)
This branch indicates that an interface is ready to accept data — similar to the Data Transfer Interrupt (I.A. above).

D. Everything Else (CARD ENABLE)
This branch signals some external event to the user program, such as Request-to-Send (RTS, 98036A Interface), Service Request (SRQ, 98034A Interface), etc.

It is important to make the distinction between hardware interrupts and end-of-line branches, as the priorities for each differ and so do the times they can occur. A hardware interrupt can occur anytime the BASIC program is being executed; the user has no control over hardware interrupts. (This is not entirely true, as the select code of an interface can affect its interrupt priority.)

An end-of-line branch, however, can occur **only** if an ON INT statement has been executed for the select code being considered, and if a DISABLE statement has not been executed, which "turns off" all end-of-line branching. The user, then, has complete control over software interrupts (or, end-of-line branches), including priority, protected code segments, and environment switching.

Data Transfers with Slow Devices

When an I/O operation is done with a very slow device using the normal ENTER/OUTPUT operations, the computer spends a considerable amount of time merely waiting for the device to become ready for the next character transfer. Having interrupt capability gives the 9835/45 the ability to do other useful work while it is waiting on the device. In order to show just how this is accomplished, we will look at an example of doing output to a slow printer both with a normal output statement, and then using the interrupt structure of the computer. To make even more evident what is happening, we will not use the automatic mechanism provided by the BINT/WINT transfer types, but will simulate their operation using the direct-register access capability discussed in the "Direct Register Access" section.

Assume that we have a slow printing device that operates at 30 characters per second, and that we wish to send to it an 80 character string to be printed. If we were to do this with the simple OUTPUT statement, OUTPUT 6; A$, it would take approximately 2.67 seconds for the printing to complete. During this time, we could have executed hundreds of program lines and accomplished a considerable amount of useful work.

In the Binary I/O Operations section we looked at how the WRITEIO statement could be used to simulate the operation of the WRITEBIN statement in sending a single character to an output device. It was mentioned that if several characters were to be sent, it would be necessary to wait for the flag line on the interface to indicate ready before sending each character. If we wanted to do other useful work while waiting for the flag line to come ready, and we did not have interrupt available, we could simulate the interrupt process by the following scheme. We first provide a subroutine that will output A$ one character at a time each time it is called.

```
100 Send:          !  SEND ONE CHARACTER OF A$
110       IF I>LEN(A$) THEN Exit
120         WRITE BIN 6;NUM(A$[I;1])
130         I=I+1
140 Exit:    RETURN
```

The value of the pointer into the string, I, is initially set to 1. Each time the routine is called, the character in the string that is pointed to by I is sent, and the pointer, I, is incremented for the next time the routine is called. Also, if all characters in A$ have been sent (i.e., I is greater than the length of the string) the routine simply returns without sending any more data. If we now edited the lines in the rest of the program, so that each one alternated with the statement

IF IOFLAG(6) THEN GOSUB Send

we would have simulated an interrupt capability. That is, between each line of the rest of the program, the flag line for select code 6 is tested, and if it has come ready, we execute a subroutine call to the "Send" routine to output one more character. Although this would work, it is not a very attractive solution. What we would much prefer is not to have to test the flag line, but rather to have the interface inform the computer when the flag line again indicated ready. We would also rather have the branch to the subroutine be automatic; that is, we would like to tell the computer once where to go when the flag line indicated ready, rather than at the end of each line. The on-interrupt (ON INT) and card enable statements do just that. The following program would accomplish the same task as the previous example.

```
1       I=1
2       A$="Example data"
10    CONTROL MASK 6;128
20    ON INT #6 GOSUB Send
30    CARD ENABLE 6
      .
      .
      .
1000 Send:   ! OUTPUT ONE CHARACTER
1010      WRITE BIN 6;NUM(A$[I;1])
1015      I=I+1
1020      IF I<=LEN(A$) THEN CARD ENABLE 6
1030   RETURN
```

The ON INT statement in line 20 essentially says "if an interrupt should come in from select code 6, branch to the routine labeled "Send"." Notice that this operation is entirely local to the program and involves no communication with the interface card. It is the CARD ENABLE statement in line 30 that informs the interface about what is happening. This statement may be read as "any time your flag line indicates ready, generate an interrupt request."

From this point, the sequence of events would be as follows. Most probably, at the time line 30 is executed, the interface is ready since we have not yet done anything to cause it to go busy. Therefore, as soon as the CARD ENABLE statement is executed, it enables the interface for one interrupt. When this interrupt occurs, the I/O ROM automatically disables the card for further interrupts, thus preventing it from trying to interrupt again until its first interrupt has been serviced. Since we want another interrupt after the current character is finished processing by the peripheral, line 1020 reenables the card for interrupts. The return statement in line 1030 causes the program to branch back to the line it would have executed next if the interrupt had not occurred.

The rest of the program (lines 40 through 990) continues to execute while the peripheral is busy processing the current character. At some point in time, it will finish that processing and indicate (via the flag line) that it is ready for more data. Since we have again enabled the card to interrupt on that condition, it will generate an interrupt to the computer. This time, however,

the computer will probably be in the middle of executing some line from the main program. If it were to immediately branch to the "Send" routine, the operations performed in "Send" would use the internal "scratch-pad" registers and make it impossible to finish the interrupted program line correctly. So instead, the I/O ROM merely makes a note of the fact that an interrupt from select code 6 has occurred, disables the interface so that it does not keep trying to interrupt, and then allows the current program line to complete its execution. When the end of the line is reached and the scratch-pad registers are free, it then causes a branch to the service routine indicated by the ON INT statement. This sequence continues until the "Send" routine outputs the last character in A$, at which time line 1020 detects that there is no more data to send and as a result does not execute the CARD ENABLE statement. As before, line 1030 returns control to the main program. But this time when the flag line again indicates ready, the interface has not been reenabled and does not generate an interrupt, since the data transfer process is completed.

Although this program is much simpler than the previous method described, it still requires the user's program to handle each character, keeping track of the pointer to the next character to be sent and enabling the interrupts at the proper time. It should be possible to make this process still more automatic and transparent to the user. This further transparency is provided by the "INT" transfer type.

The printer we have been considering in our example is classified as a slow device because it requires about 33 milliseconds of wait time between characters. If this printer had some read/write memory built into it, it could then accept characters at a much faster rate, place them in that memory (called a data buffer), and then fetch them from the buffer and print them in the order that they were received. For short bursts of data, this slow printer with a built-in buffer would appear to be a medium speed device. Although the printer would not print the characters any faster, it can accept an entire line of data very quickly, and then process it out of its storage buffer at its normal printing speed. The figure below shows a schematic representation of this process.

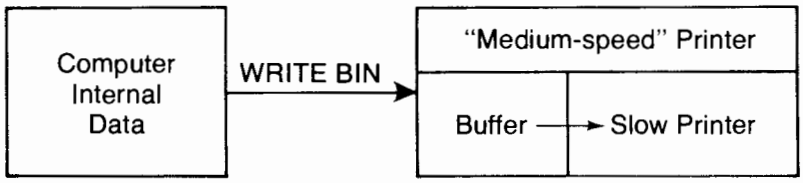


Figure 24

If a slow printer does not have its own built-in buffer, it can still be made to look like a medium-speed device by allowing it to use some of the computer's memory as its data buffer, which we'll call a transfer buffer.

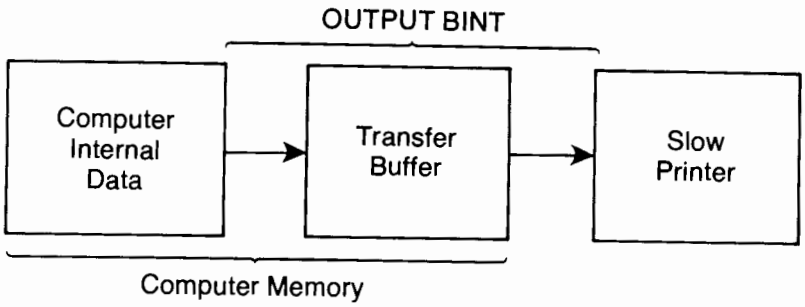


Figure 25

Using this scheme, the data to be output is normally written to the transfer buffer, just as though this buffer memory were contained in the peripheral itself. The OUTPUT...BINT statement is used to send the data to the transfer buffer and then to the printer — at the printer's own speed — using the interrupt capability. This transfer process is entirely automatic and handled totally by the I/O ROM. The statement to accomplish this is given below.

```
10    OUTPUT 6 BINT;A$
```

The BINT parameter in the OUTPUT statement specifies the transfer type which is discussed in the next section. Once the transfer buffer is established, the I/O ROM simply outputs to the transfer buffer as though it were a peripheral. Since this data has simply been moved to the buffer and not yet sent to the peripheral, this operation happens very quickly and does not depend on the speed of the peripheral device. Depending on the size of A$, the I/O ROM places all or part of the contents of A$ into the buffer. The data in the buffer is not in the internal representation of the computer, but represents the exact character sequence that would have been sent to the peripheral if a buffer were not being used.

Now that some data is in the buffer, the I/O ROM starts the process that transfers it to the printer under interrupt. From here, the process is automatic, and handled entirely by the I/O ROM. Thus, the user's program is free from having to set up for interrupts, manage the data pointers, and terminate the process when all the data is sent.

The execution of the OUTPUT statement itself may be complete even before the first character has been sent.[1] This means that the rest of the program can continue executing, even though there is still more data in the buffer to be transferred. Each time the peripheral device comes ready for the next character, the running program is momentarily interrupted[2] and another character is sent by the I/O ROM without the need for further program statements. Also, since this transfer of the next character can be done by the ROM without using the scratch-pad registers, it can be done when the interrupt occurs and does not have to wait until the current program line is completed. When the last character has been sent, the ROM automatically disables the interface from further interrupts. As a result, the entire burden on the user program is simply to initiate the transfer process.

1 The OVERLAP mode of program execution is assumed. See the section on Overlapped I/O page 74.

2 This is actually true only for the System 35 :the System 45 computers have a separate I/O processor and do not halt program execution.

Further Data Transfer Examples

We have seen how the OUTPUT BINT statement can be used to output data to a slow device using interrupt. This same buffer transfer mechanism can be used to input data from a slow device. Before looking at how this would be done automatically using the ENTER BINT statement, it would be instructive to first write a program to accomplish the same task using a user-programmed service routine which will show all of the steps involved.

```
 5     CONTROL MASK 3;128          ! Set up mask
10     ON INT #3 GOSUB Input       ! Set up service
20     WAIT READ 3,4;Z             ! Request input
30     WRITE IO 3,7;0              ! Send card busy
40     CARD ENABLE 3               ! Enable interrupts
       .
       .
       .
1000 Input:    !   READ IN ONE CHARACTER
1010    READ IO 3,4;C
1020    A$=A$&CHR$(C)
1030    IF C=10 THEN 1060
1040    WRITE IO 3,7;0
1050    CARD ENABLE 3
1060 RETURN
```

Comparing this program to the analogous one for the output case in the previous section, we see that the main difference is in the method of transferring the individual characters. In the output "Send" routine, each character was sent using a simple WRITEBIN statement. In this program, however, we cannot use the READBIN statement to input each character, but have to resort to the use of the direct register access statements. The reason for this will become clear if we refer back to the individual register sequences that make up the WRITEBIN and READBIN operations (pages 20 and 21). These sequences are summarized in the following diagram.

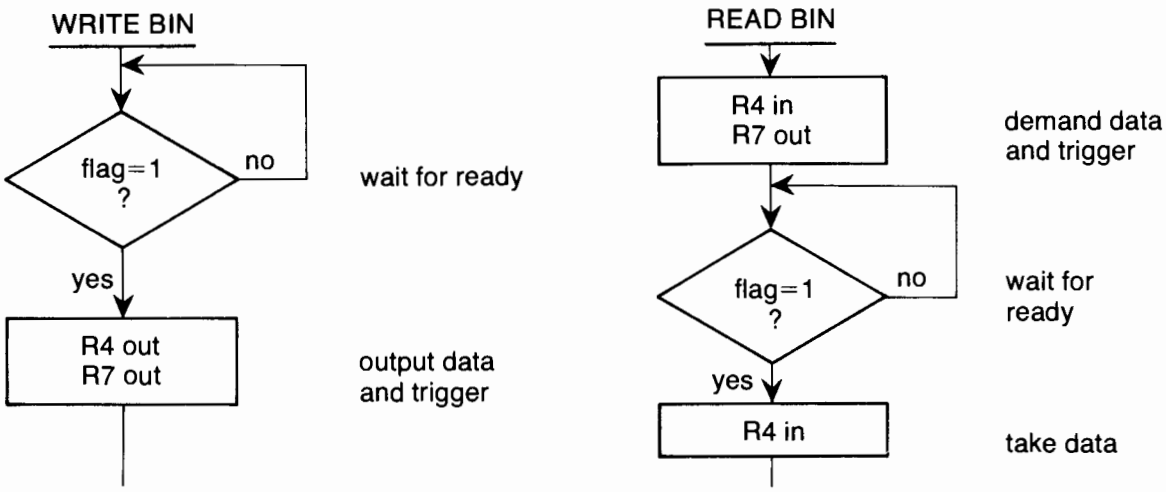


Figure 26

In the output example, when the device came ready for the next character it set the flag line to indicate ready, which caused the interface to generate an interrupt. The I/O ROM then caused a branch to the "Send" routine which did the WRITEBIN operation. Since at this time the flag line was indicating ready, the WRITEBIN statement did not have to wait and immediately performed the output of the next character.

Looking at the diagram for the READBIN function, however, we see that it must first demand the data, wait for the data to come ready, and then take in the data. If we had tried to use the same program as in the "Send" example, merely replacing the WRITEBIN with a READBIN, the following would have happened. The first CARD ENABLE statement would have caused an immediate branch to the service routine, since nothing had caused the device to go busy. The READBIN in the service routine would have demanded the next character and then waited for it to come ready. Thus, we would lose any advantage of being able to avoid the wait time through the use of interrupts.

What we would like to do is to demand the next character and then go away and do other useful work during the time we would normally be waiting for the character to come ready. This is what the program given above accomplishes. Line 10 sets up for a branch to the "Input" service routine whenever an interrupt occurs from select code 3. Line 20 tells the interface to demand the next character from the device. Since we do not know whether the last operation to the card was an input or an output operation, we first execute an R4 in so that the WRITE IO 3,7 will trigger an input operation. The data received from this R4 in operation is whatever was left on the data lines from the last input from the card, and does not represent useful information. Having triggered the input, the interface has now gone busy and we enable it to interrupt when it again comes ready. In the meantime, the program in lines 50 through 990 continues execution. The location of the CARD ENABLE statement in the sequence is very important. It cannot be executed until we have made the interface go busy; otherwise an immediate branch to the service routine would occur.

When the device indicates that the next character is ready, the interface generates an interrupt to the computer; and at the end of the current program line, the I/O ROM causes a branch to the "Input" service routine. This routine can now complete the last phase of the READBIN operation which is to take the new character from the interface using an R4 in operation. In this example, we are expecting an ASCII message from the peripheral device which will be terminated by a line-feed (LF = decimal 10) character. Thus, the input routine next converts the byte received into an ASCII character and concatenates it onto the string A$, which would have initially been set to a null string. In line 1020, we test the byte just received to see if it was a LF. If not, we have not yet received the entire message and so we trigger another input operation (i.e., demand another character) which again makes the device go busy, and then enable the interface for another interrupt. Notice that this time the "dummy" READIO 3,4 is not required, since we know that the last R4 operation to the card was an input. If the character received had been the LF terminator, the program would not have triggered another data input operation and would not have enabled the card to interrupt again. In either case, the RETURN statement causes the program to branch back to where it came from in the main program.

Again, this example was given to show the steps that are performed when data is input using interrupt. In practice, there is no need for the user's program to handle each character as it comes in, since the ENTER BINT statement provided by the I/O ROM makes the entire process automatic. The following statement would accomplish the same task.

ENTER 3 BINT;A$

The statement specifies that data should be transferred **from** select code 3 **to** the input buffer until there are enough characters to fill A$ or until a line-feed (decimal value 10) is received. From this point, the entire process is automatically handled by the I/O ROM, and the program is free to continue execution while the buffer is being filled. Once the transfer into the buffer is completed, the data buffer is read into the string A$ where it can be used by the rest of the program.

The only question remaining is, how do I know when the transfer is complete so that I can access the string A$? This is made possible by means of the ON INT capability, as demonstrated in the following program.

```
5     DIM A$[800],Buffer$[10]
6     OVERLAP
10    ON INT #3 GOSUB Complete
20    A$=""
30    ENTER 3 BINT;Buffer$
      .
      .
      .
      .
      .
1000 Complete:   !  SAVE NEXT COMMAND OR TEXT
1010     A$=A$&Buffer$
1020     ENTER 3 BINT;Buffer$
1030  RETURN
```

In this program, we use the ON INT statement to set up a branch to a service routine called "Complete." This is similar to the ON INT branch to the "Input" routine in the previous example except that instead of branching to the service routine each time the next character is ready, the branch to the "Complete" routine takes place only when the entire input operation specified by the transfer statement is complete. In other words, using the CARD ENABLE statement, the service routine is called when the interface indicates ready; using ENTER BINT, the service routine is called when the transfer is complete. Thus, the ON INT statement does not always cause a branch to the service routine when an interface interrupt occurs, since the I/O ROM automatically handles the interrupts associated with the transfer process.

User Programmed Service Routines

The tasks that a computer performs in which interrupts may be useful fall into three major categories: data transfer completion, timeouts, and everything else. The task of sending or receiving data is usually a well-defined process that can be specified by a small number of parameters; namely, how much data there is, where it is located, and the type of transfer to be performed (i.e., interrupt, fast read/write, or DMA).

If the interrupt is for a purpose other than simple data transfer, the scope of the tasks to be performed is so large and varied that it would be extremely difficult to provide pre-programmed (i.e., ROM) service routines to handle even a small portion of these tasks. As a result, provisions were made to allow the user to write the service routines required, using the same high-level language in which the rest of the program is written. Thus the service routines can perform any operations and execute any statements that can be done in the background (non-service routine) segments of the program.

Most of the interface cards only have one interrupting condition, and that is when the flag line indicates that the peripheral device is ready for another operation. Since this flag line is usually used in conjunction with data input and output, the associated user-written service routines are then used to specify what action is to be taken when a data transfer operation has completed. An example of this type of service routine was given in the previous section on Further Data Transfer Examples.

As a further example, consider a computer with a digitizer and a plotter attached. Each time a point is entered through the digitizer, we would like this point to be plotted, thus giving a hardcopy record of the points that were entered. In the meantime (i.e., while the program is waiting for the next point to be digitized) we would like the program to be performing some kind of analysis on the points that have been entered. The following program is an example of how that task could be performed using interrupt and a user-written service routine.

```
  10   DIM X(500),Y(500),Digitizer$[14]
  11   OVERLAP
  20   ON INT #4,1 GOSUB New_point
  30   N=0
  40   ENTER 4 BINT;Digitizer$
       .
       .
       .
1000 New_point:  N=N+1
1010   X=VAL(Digitizer$[1,6])
1020   Y=VAL(Digitizer$[8,13])
1030      PLOT X,Y
1040   IF N<500 THEN ENTER 4 BINT;Digitizer$
1050      X(N)=X
1060      Y(N)=Y
1070 RETURN
```

In this program, we dimension the X and Y arrays to hold up to 500 pairs of X, Y coordinates. Having specified the location of our service routine as "New-point" and initialized a counter, N, to zero, we then start a transfer operation from the digitizer (SC=4) to the temporary string variable Digitizer $. Since we know that the digitizer always delivers its readings as a 14-character ASCII sequence (±XX.XX,±YY.YY**LF**) it is not necessary to specify any terminating conditions in the ENTER statement.

Each time a new point is digitized the buffer transfer completes and a branch to the service routine is made. In this case, the service routine reads the X and Y values out of the buffer and immediately starts another transfer operation in order to be ready to receive the next point as quickly as possible. Before leaving the service routine, the values just read are placed in the next available position in the X and Y arrays as indicated by the pointer N. Lines 50 through 990 of the program would contain statements to perform the desired analysis of the digitized points, using the value of N to determine how many values in the X and Y arrays represented valid data.

Some applications may require the use of interrupt and a user-written service routine that has nothing to do with data input/output. For example, the computer might be connected to a remote temperature sensing device in a chemical processing operation. Normally, the computer is performing routine control functions throughout the rest of the system. But if the temperature in a critical location becomes too high, the sensing device would like to interrupt and have the computer make the necessary adjustments. Assume that the sensing device has an output line that is in one state when the temperature is normal, and goes to the other state when the temperature exceeds the normal operating range. This line could be connected to the flag line of the interface and the logic level jumper set to indicate BUSY in the normal range, and READY outside of this range. The following program would then be used.

```
   5      CONTROL MASK 2;128
  10      ON INT #2 GOSUB Too_hot
  20      CARD ENABLE 2
          .
          .
1000 Too_hot:     !  ROUTINE TO ADJUST THE TEMPERATURE
          .
          .
          .
1080    CARD ENABLE 2
1090  RETURN
```

In the previous examples, the branch to the service routine was caused by the completion of an interrupt transfer. In this case, the statement in line 20 specifies that interrupt should be enabled on select code 2 (the temperature sensor). Initially the temperature is within range and the flag line on the interface indicates "busy" or a zero. If the temperature goes out of range, we have wired the interface and sensor in such a way that the flag line will indicate "ready" or a one. This will cause the interface to interrupt the processor. The I/O ROM will detect this interrupt and cause a branch to the "Too hot" service routine.

In the previous examples of interrupt transfer operations, we did not use the CARD ENABLE statement, since the I/O ROM firmware automatically enables and disables interrupts from the interface at the proper times in the data transfer sequence. Normally, a program would use either an interrupt transfer or a CARD ENABLE statement for a given select code, but not both.

It is important to notice that it is the computer and not the peripheral device that initiates an interrupt process. The program must first establish a service routine, and then enable the interface to interrupt. When the actual interrupt is generated, it is merely signaling the completion of the process that was started by the computer. In the case of data transfer, the interrupt indicates that the last item sent or received is completed, and that the device is ready for another one. In the case of service routines which are accessed through the CARD ENABLE statement, the interrupt indicates that the condition which was specified as an interrupting condition has occurred. Unless the computer has initiated the entire interrupt process, it will have no idea what to do when the interrupt is received. Thus, one should never think in terms of a peripheral device issuing an interrupt independently of the program which is controlling that device.

Interrupt Priorities

Up until now, we have discussed interrupts from one interface at a time. When more than one interface is operating under interrupt, it is possible that two or more of these interfaces might generate interrupts at the same time. In this case a system of priorities must be established that will determine the order in which the service routines are performed.

Before discussing the rules that determine these priorities, it is important to have a clear picture of the various operations that use interrupt, and the parts of the system that handle each of these operations.

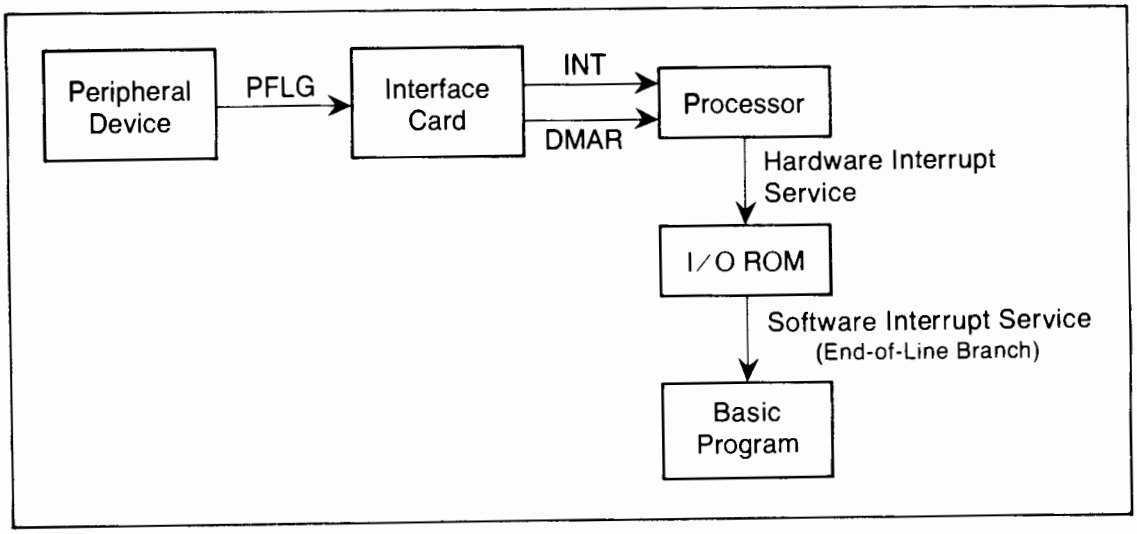


Figure 28

As mentioned in the last section, the peripheral device itself does not generate any interrupts to the computer. It merely indicates to the interface card on the peripheral-flag (PFLG) line that it is ready for more data or that a condition wired into the PFLG line is true. It is up to the interface to translate this signal into an interrupt if it has been enabled to do so by the program using a CARD ENABLE or a BINT transfer statement. Let us follow the sequence of events that occur in each of the four possible cases: CARD ENABLE, interrupt transfer, fast-handshake transfer, and DMA transfer.

If a service routine has been set up (using the ON INT statement) and a CARD ENABLE statement is executed, this enables the interface to generate an interrupt request whenever PFLG line indicates true. The processor is normally executing machine language (also called assembly language) instructions which are carrying out the operations specified by the lines of the user's BASIC program. When this interrupt is received, the processor suspends execution of the machine code sequence[1] it was in, and branches to another block of machine code in the I/O ROM called the ROM service routine. This ROM service routine must decide whether or not a transfer is in progress with the interface that generated the interrupt. In this case it is not, so the ROM merely logs in the fact that the interrupt occurred and allows the processor to resume execution[1] of the code it was working on before the interrupt came in. It also disables the interface so that it will not continue interrupting, since the interrupt has been noted and logged in. When the end of the current line of the user's program is reached, the I/O ROM is again given control by the processor. At this time it detects that the interrupt has been logged in, and so it forces a branch to the user's service routine. Notice that all that took place at the time of the interrupt was the log-in procedure, and that this happened immediately. The branch to the user's service routine happened later, and was the result of that select code having logged an interrupt and the end of the current program line being reached.

When a transfer (ENTER or OUTPUT) statement is executed using an interrupt buffer (BINT or WINT), the I/O ROM starts the first data item transfer and enables the interface to interrupt when the PFLG line again indicates ready. The processor is then allowed to continue execution of the program. Each time the device becomes ready, the interface generates another interrupt to the processor. The processor branches to the ROM's machine language service routine which detects that a transfer is in progress, and sends or requests the next data item. When the transfer is complete, the I/O ROM looks to see if the ON INT statement has been executed for this select code. If it has, the ROM logs in an end-of-line branch request, just as in the previous CARD ENABLE; and at the end of the current line of the user's program, a branch to the user's service routine is taken.

1 The System 45 computers do not actually suspend program execution to service interrupts, as they have a separate processor to deal with interface operations.

A fast handshake transfer operates in the same way, except that after the interrupt on the first data item, the ROM does not return control to the processor[1] but keeps control until the entire buffer is transferred. During this time, the processor is not executing any of the user's program and is not acknowledging any other interrupt requests from other interface cards. The machine is essentially dedicated to the transfer process. When the transfer is complete, a request to branch to the user's service routine is logged in if an ON INT statement was executed previously, and control is returned to the processor.

In the final case of a DMA transfer, the situation is somewhat different. When the transfer statement is executed, the I/O ROM first requests that the processor grant it use of the DMA resource. Since there is only one DMA channel, if it is already in use, the I/O ROM must wait until it is free. When the DMA channel is granted, the I/O ROM informs the processor which interface will now use it, what area of memory is to be used, whether an input or output operation is to be performed, and how many words are to be transferred. The ROM then enables the interface for a DMA transfer and returns control to the processor. Now each time the peripheral device comes ready, the interface does not generate an interrupt but responds on a special DMA request line (DMAR). This line causes the processor to send or receive one more word of data using the area of memory previously specified by the I/O ROM. These data transfers are handled entirely by the processor and the I/O ROM is not involved. When the transfer is complete, the processor informs the interface card of this fact, and the interface then generates a normal interrupt. This time the ROM service routine is called, giving it a chance to log in an end-of-line branch request if an ON INT statement has been executed for this select code.

Thus, there are two kinds of interrupts: a true interrupt generated by the interface to the processor and serviced by the I/O ROM, and a pseudo-interrupt generated by the I/O ROM at the end of a program line to force a branch to the user's service routine. We often speak of "generating an interrupt to the user's service routine," although in reality this is not a true interrupt but merely a program branch. This distinction is important since the two types of interrupt are completely independent of each other in the matter of interrupt priorities.

We will first look at the priorities for the true (interface generated) interrupts. This type of interrupt is also called hardware interrupt. These interrupts are assigned two levels of priority according to the select code of the interface that generates them. Select codes 0 through 7 are assigned a low level priority, and select codes 8 through 15 are given a high level priority. These levels are also called 1 (low) and 2 (high). We can also think of the processor as operating on a given level at any time. If it is executing machine code to carry out lines of the

1 The System 45 computers do not actually suspend program execution to service interrupts, as they have a separate processor to deal with interface operations.

user's program, we say that it is operating at level 0. When a low level interrupt comes in and the I/O ROM is in a machine-language service routine (i.e., transferring the data or logging in an end-of-line branch request) the computer is operating at level 1. If the I/O ROM is in a machine-language service routine for a high-level interface, the computer is operating at level 2. Thus, each interface has a priority level (1 or 2) depending on its select code, and the computer is in a certain state (0, 1, or 2) depending on whether it is executing the user's program lines or an I/O ROM service routine. We can now state the rule for hardware interrupts as follows: a hardware interrupt request will be granted by the processor if the level of the interrupt is greater than the current state of the computer. If the interrupt level is less than or equal to the current state, the interrupt will not be granted until the state becomes less than the interrupt level. Stated in other words, a low level interrupt has priority over the user's program, but not over a high level or another low level service routine. A high level interrupt has priority over everything except another high level service routine.

Completely independent of this priority scheme for hardware interrupts, there is a set of priorities for branches to user service-routines, or software interrupts. We can define the program state as 0 (not in a user written service routine), or from 1 (executing lines in a priority 1 user service routine) to 15 (the highest priority user service routine). At the end of each line of the program, if an end-of-line branch is logged in whose defined priority level is higher than the current program state, a branch to that service routine will be performed. Otherwise, program line execution will continue normally. If at the end of a program line, two or more branches are logged in on the same level, and that level is higher than the current program state, the one corresponding to the higher select code will have its user service routine executed first.

As an example, assume that the program is executing a user service routine with a priority level of 3. During one line of this routine, select code 9 with a priority level of 9 logs in for end-of-line service, followed by a select code 12 with a priority level of 12. When the end of the current line is reached, control will be given to the service routine with priority level 12. Notice that even though 9 logged in before 12, both branches were pending by the time the end of the line was reached and the one with the higher priority got service first. Thus two branch requests logging in during the same line of the program are considered to have come in simultaneously. When the RETURN statement for service routine 12 is encountered, the program drops from level 12 back to level 9. Since a level 9 branch for select code 9 is still pending, it will get service next.

During all of this processing of user written service routines, if true (hardware) interrupts had occurred they would have been serviced by ROM service routines according to the set of hardware priorities, independently of what user level service routines were in progress. That is, all lines of the user's program (background job, low level and high level user service routines) are the same as far as determining the processor state is concerned. Alternatively, we may say that hardware interrupts have priority over all levels of software interrupt.

High-Speed I/O Programming

Overview

As discussed in the previous section, peripherals can vary in their data transfer rates, with very slow or random rate devices being best serviced by interrupt transfers. This allows the program to do other useful work while waiting for a slow peripheral device to come ready to send or receive the next character of a data message. The problems encountered in dealing with very fast devices (transfer rates over 10000 characters per second) are of an entirely different nature, however, and will be discussed in the following sections.

Fast peripheral devices may be classified into two categories: those capable of operating at less than their maximum speeds and those that are not. The former category consists of devices that can deliver data on request from the computer; these are sometimes called asynchronous or non-periodic devices. The latter category consists of devices whose data transfer rate is synchronized to some external timing mechanism over which the computer has no control; these are called synchronous or periodic devices. An example of a synchronous device is a magnetic tape reader. To operate properly, the tape must move past the head at a constant velocity, so the device is not capable of starting and stopping within a block of recorded data at the command of the computer.

The distinction between medium-speed and high-speed devices may depend not only upon the transfer rate of the device, but upon the application as well. For example, a digital voltmeter (DVM) may be capable of taking 5000 readings per second. Now, this DVM is not forced to take readings at this fixed rate (in other words, is not synchronized to an external clock), but instead it simply delivers a reading each time the computer asks for one — up to its limit of 5000 readings per second. Surprisingly, the fastest method of reading and processing data from the DVM would be to use a simple ENTER statement. Interrupt transfers are primarily intended to avoid a wait for slow devices, and are of no use here. In this case, however, the device is sufficiently fast that the computer spends little or no time waiting for the DVM to come ready. The simple ENTER statement does not require the overhead necessary to set up a fast-handshake or DMA transfer, so its overall throughput is faster.

Suppose instead that we want to periodically measure a fast event whose duration is only a fraction of a second? This might require taking many readings in a very short period of time, with a transfer rate that is faster than the natural speed of the ENTER statement. The ENTER statement is required to perform several functions, including gathering the raw data (say the ASCII characters representing each reading), doing any specified conversions and formatting, putting the data into the internal machine representation, and locating the destination variable for final storage. All of these operations consume processor time and contribute to the overall

time required for the simple ENTER statement to take data. These and other processes determine the "natural" input speed of the simple ENTER statement. Now, if only the raw data could be gathered and further processing put off until some later time, the actual data collecting process could proceed at a much faster rate. This is the principle behind the fast-handshake and the direct-memory-access (DMA) modes of operation.

Fast-Handshake Transfers

I/O using fast-handshake transfers is very similar to I/O using interrupt transfers. Instead of specifying BINT or WINT, you specify BFHS or WFHS for byte or word transfers, respectively. The firmware must now set aside a buffer large enough to hold the required number of bytes or words for the transfer (or as much as available memory will allow). For an input operation, this is specified by the {count} parameter; for an output operation, this can be computed from the size of the data list. For example,

```
10      ENTER 3 BFHS 304;A$
20      OUTPUT 6 BFHS ;A$
```

The fast handshake input operates as follows. Once the input buffer is established, the firmware demands the first data item from the device. The I/O firmware stays in a tight loop of machine language code and gathers the data items as fast as it can. Interrupts from other interfaces are disabled for the duration of this transfer, and all other I/O is discontinued until the fast-handshake transfer completes. In essence, the computer has dedicated itself to the data transfer process to obtain maximum speed, but once the transfer completes, the computer resumes normal operation.

Once the raw data has been input, the program may do any specified formatting or conversions necessary.

If the program has executed an "ON INT" statement for this select code, a branch to the specified service routine is performed when the end of the current program line is reached. In certain models of computers (the 9845B for instance), program execution can continue during the transfer if NOFORMAT has been specified. This is possible because a dedicated processor is used for I/O transfers, thus freeing up the main processor to continue program execution. The "Advanced I/O Programming" section of this manual contains more details about NOFORMAT transfers.

The fast-handshake output functions in a manner similar to input. The firmware cycles in a tight loop transferring data items until the entire data list is transferred. As with the fast-handshake input, an end-of-line program branch is taken when the transfer completes if an ON INT statement is in effect for this select code. This branch informs the program that the transfer is complete and that a new transfer can be initiated if necessary, or that the variables can be accessed by the program.

Direct Memory Access Transfers

For extremely high-speed requirements, the 98032A 16-Bit Parallel Interface offers an even faster transfer mode known as DMA, or direct memory access. This mode always transfers 16-bit data items, but by specifying BDMA (byte mode DMA), only the lower eight bits of a word are considered valid. On BDMA input, these lower eight bits are taken from each word of the buffer and used for the formatting and conversions necessary to get the raw data into the correct form for the program variables in the data list. The converse applies to BDMA output, where each byte (eight bits) of data is placed into the lower half of a word in the output buffer.

DMA transfers are similar to fast-handshake transfers, except that each time the device is ready for (or with) the next data item, the interface interrupts on a special DMA Request line. The next item is transferred into or out of the buffer by the processor itself, and not even the I/O ROM is involved in this process. The fact that the processor itself handles the data transfer is the source of the speed of the DMA operation. This allows program execution to continue during the transfer, and also allows other I/O transfers to continue. However, some lack of versatility is the price paid for this speed. For instance, the invert-data jumper on the 98032A interface is ignored, and if positive-true data is being transferred by the device the result will be nonsense data. (In this case, the BINCMP function or a Variable-to-Variable transfer with a conversion table would have to be used to re-invert the data.)

It is strongly recommended that DMA transfers terminate because the specified byte count is satisfied. Otherwise you will have to always handshake when PCTL is true and EIR is enabled to terminate the DMA transfers. In otherwords, make sure that the proper number of bytes is sent or received.

Advanced I/O Programming

This section presents the functions of the data transfer in greater detail than was done in Chapters 2 and 3. We will be analyzing the tradeoffs necessary to optimize I/O for either higher overall throughput or for higher burst transfer rates. At the end of this section is a discussion of the effects of memory organization on I/O programming.

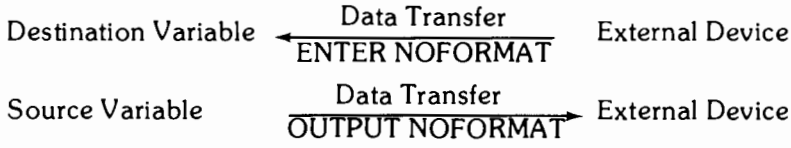
There are two functional processes that make up an I/O transfer: the formatting and conversion process, and the data transfer process. The figure below illustrates this functional split.

The data transfer process is the character-by-character data exchange between the computer and the external device. The formatting/conversion process is an internal function that is necessary to translate between internal data representations and the codes recognized by the external device. These two processes are normally transparent to the I/O programmer, however, it is sometimes necessary to separate these processes to optimize the I/O program.

The data transfer process can be separated from the formatting process by specifying a NOFORMAT transfer. The syntaxes for a simple data transfer without formatting are shown below[1]:

```
ENTER <select code> NOFORMAT; <data list>
OUTPUT <select code> NOFORMAT; <data list>
```

Pictorially, the NOFORMAT transfers look like this:



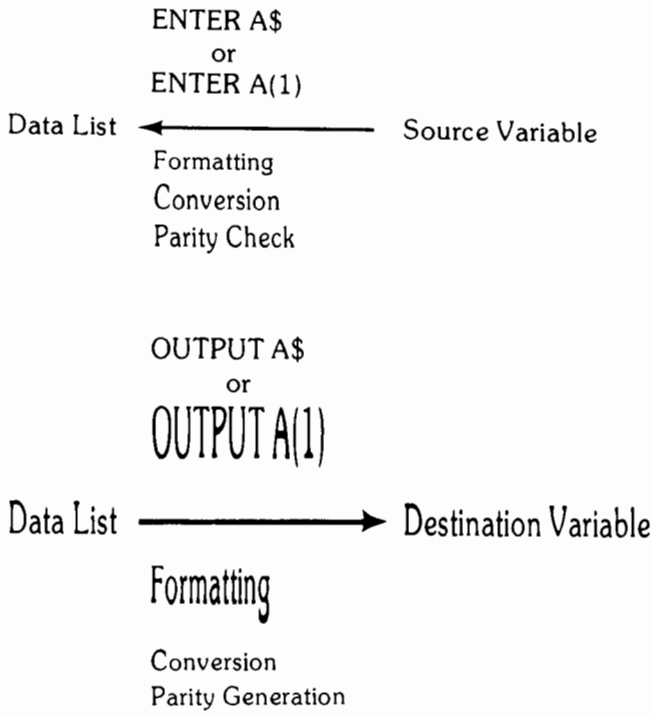
1 These syntaxes are the simplest case; refer to your I/O ROM programming manual for more detailed syntax descriptions.

(rather than a select code) as the source or destination of data. The syntaxes now look like this:[1]

ENTER <Source variable> [USING <image>];<data list>

OUTPUT <Destination variable> [USING <image>];<data list>

The transfers look like this pictorially:



The following two sections detail the NOFORMAT and the Variable-to-Variable transfers, and give the user some insight into the why and how of using them.

NOFORMAT Data Transfers

In an earlier discussion about fast-handshake transfers, we noted the transfer rate improvement possible by separating the raw data transfer from the formatting and conversion process. In a normal ENTER or OUTPUT operation, this formatting and conversion process must still be done so the data is in a form suitable for the computer to use. But suppose the data is to be sent to another identical computer (9835/45), or at least one whose internal data format is identical to our own. It doesn't make sense to use the sending processor's time to deformat its internal data and then use the receiving processor's time to reformat this data. NOFORMAT transfers allow us to bypass this formatting and conversion process, so the data list variables are sent and received in their internal binary representation.

1 These syntaxes are the simplest case; refer to your I/O ROM programming manual for more detailed syntax descriptions.

What kind of speed improvement does a NOFORMAT data transfer have over a formatted data transfer? The bottom line improvement depends on so many factors that it is impossible to predict the outcome for a particular application, but for a very simple test the improvements for input are shown below.

Noformat Transfer Rate: Formatted Transfer Rate

	Transfer Type	16 Word Integer Array	1000 Word Integer Array
ENTER	WHS	2.6:1 improvement	4:1 improvement
	WFHS	6.0:1 improvement	65:1 improvement
	WDMA	4.6:1 improvement	143:1 improvement

The above table suggests that a considerable amount of processor time is spent doing data conversions and formatting, and that transferring large amounts of data per transfer reduces the effect that the system overhead (necessary to execute a BASIC statement) has on overall transfer rates. This is especially evident in the fast-handshake (WFHS) and the DMA (WDMA) transfers.

All these speed improvements may seem impressive, but how often is our computer going to be involved in computer-to-computer data transfers? (Rhetorical question). It turns out that NOFORMAT transfers are useful in applications where high transfer rates **and** data formatting are necessary as well. The formatting and conversion process is accomplished by the Variable-to-Variable transfer, which is covered in the next section.

Another benefit to NOFORMAT transfers is that the user can obtain true overlap capabilities for ENTER. The incoming data can be placed directly into a string variable or numeric array where it can be kept until a Variable-to-Variable transfer is used to format the data into the correct internal representation. (The Variable-to-Variable transfer is necessary only if the data needs to be converted to numerics, or if any code conversions/parity checking need to be done.) ASCII strings can be input NOFORMAT with no other processing necessary, however, the user should be aware of the difference between the following two program segments:

```
50      ENTER 6;A$                ! Normal input terminates on LF
                                    or full string.
60      ENTER 6 NOFORMAT;A$       ! NOFORMAT input terminates
                                    only on full string.
```

The NOFORMAT transfer (line 60) terminates only when the dimensioned length of A$ has been filled. To terminate the transfer upon receipt of a line-feed, use the TRL (not available on the 9845A) function as shown on the following page.

```
60     ENTER 6 NOFORMAT;TRL<10),A$
```

Now the transfer terminates when A$ is full or when an ASCII LF (decimal 10) is received. We should note here that the Fast-Handshake and DMA transfer types allow a <count> parameter, which specifies the total number of bytes or words of data to transfer.

Although a string variable is the suggested method of buffering a NOFORMAT input, it may be necessary to read in more data than a string will allow. The maximum size of any string variable or array is 32 767 characters, or bytes, so an alternative variable type is needed for our large buffer.

A numeric array provides us with the extra size needed, as shown here:

```
DIM Buffer_array (1:32767)
```

The actual size of Buffer_array is 32 767 elements of eight bytes per element, or 262 136 bytes total for our buffer. This buffer size is large enough for most purposes, but a different type of question now comes to mind...what's going to happen to the computer if we put ASCII character data into a numeric array when the array is supposed to be in an internal BCD format?

The computer doesn't crash like one might expect, at least as long as we don't try to perform numeric computations on those ASCII characters. (What's the square root of "BILL"?). It's even possible to output the array, print it, and assign it to other numeric variables without causing an error. The data in the array can be checked and formatted by means of a Variable-to-Variable transfer, as done with the string variable buffer.

Variable-to-Variable Transfers

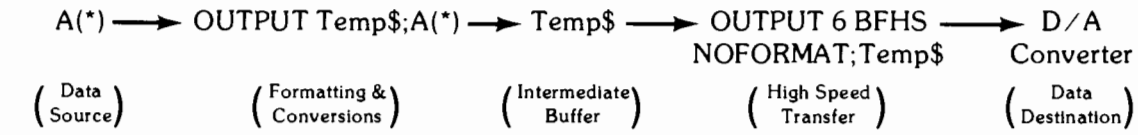
In the preceding section, we talked about the transfer portion of a data exchange, the NOFORMAT transfer. In this section we will deal with the formatting and conversion portion of a data exchange, the Variable-to-Variable transfer. The purpose of this type of transfer is to "translate" between the ASCII data being transferred and the internal binary representation used by our computer for variables. It can also take care of any conversions, such as ASCII to EBCDIC, that need be done to communicate with the external device. Parity checking (input) and generation (output) are also done at this stage. You can see that even though the Variable-to-Variable transfer is done at full processor speed, there are many operations performed that consume a large proportion of the overall time needed to actually complete a normal input or output data transfer.

Unlike Fast-handshake, Interrupt, or DMA transfers, the Variable-to-Variable transfer does not provide end-of-line branching upon completion of the transfer. The reason for this is easy enough to see: the processor can execute no program lines when it is busy formatting data. Therefore, no program lines are executed during a Variable-to-Variable transfer. The processor can, however, do other I/O (except Fast-handshake) because the formatting can be interrupted for Interrupt transfers, and because DMA transfers require no processor attention once initiated.

The actual Variable-to-Variable transfer is quite simple. The primary difference is that for input, our source of data is not an external device, but is a variable such as a string array element (where it was typically put by a NOFORMAT transfer). For output, the data is not sent to the external device, but to a variable. From there it can later be sent to the external device by means of a NOFORMAT transfer — at very high speed.

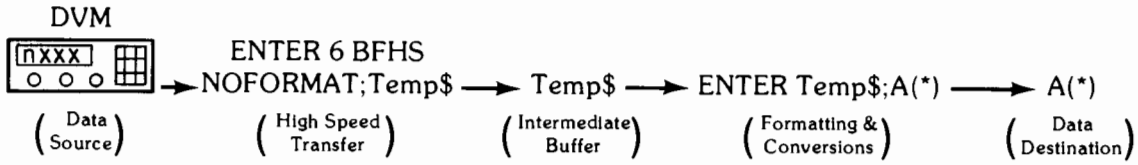
The diagram below depicts a combination of NOFORMAT and Variable-to-Variable transfers to output a waveform data base to a d/a (digital-to-analog) converter:

OUTPUT



The diagram below shows the relationship between NOFORMAT transfers and Variable-to-Variable transfers for an input operation.

INPUT



Let's take the case of a high-speed DVM for an example. Our program is to take 1000 readings per burst, and we want five bursts of data to process. To increase the speed of the transfers, we will use a NOFORMAT transfer. Since the DVM transfers six characters of data per reading, we need a string array with five elements of 6000 characters each. Our program has a pointer into the string array, I, which keeps track of the string element currently receiving the data. As a burst transfer completes, the pointer is incremented and another transfer is initiated, until all five string elements are filled. At that time, the data is stored on the tape for future processing.

Our program merely inputs the data as rapidly as possible: any formatting or conversions to be done must be handled later, which is the subject of the next section.

```
10      OPTION BASE 1
20         OVERLAP
30      DIM Temp$(5)[6000]
40      ON INT #6,3 GOSUB Dvm_complete
50       I=1
60       ENTER 6 BFHS 6000 NOFORMAT;Temp$(I)
70        IF I<=5 THEN 70
80        !  Further processing could be done here.
90        !
100      ASSIGN #1 TO "Save"
110      PRINT #1;Temp$(*),END
120      !
130      PRINT "ALL DATA SAVED"
140      STOP
150      !
160      !
170 Dvm_complete:       I=I+1
180       IF I>5 THEN Exit
190         !  ELSE  Start next data transfer.
200         ENTER 6 BFHS 6000 NOFORMAT;Temp$(I)
210 Exit:       RETURN
220      END
```

The following program takes the data saved by the previous example and processes it one burst (1000 readings) at a time:

```
1           OPTION BASE 1
10       DIM Dvm_array(1000),Temp$[6000]
20       ASSIGN #1 TO "Save"
30        FOR I=1 TO 5
40         READ #1;Temp$
41         ! Note that each value is separated by a LF.
42         ! This LF delimits the numbers in Temp$.
50         ENTER Temp$;Dvm_array(*)
           .
           .    Process the data here
           .
120       NEXT I
130      END
```

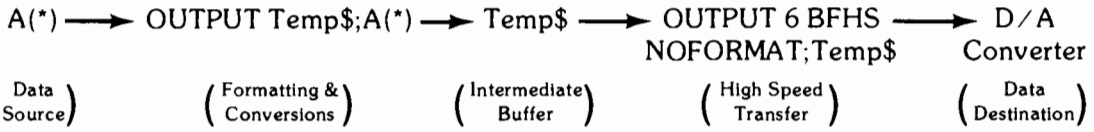
Instead of appending the L/F to Temp$, we could also have specified an IMAGE reference in our ENTER statement that disabled the normal L/F requirement to terminate the ENTER statement.

Unlike Fast-handshake, Interrupt, or DMA transfers, the Variable-to-Variable transfer does not provide end-of-line branching upon completion of the transfer. The reason for this is easy enough to see: the processor can execute no program lines when it is busy formatting data. Therefore, no program lines are executed during a Variable-to-Variable transfer. The processor can, however, do other I/O (except Fast-handshake) because the formatting can be interrupted for Interrupt transfers, and because DMA transfers require no processor attention once initiated.

The actual Variable-to-Variable transfer is quite simple. The primary difference is that for input, our source of data is not an external device, but is a variable such as a string array element (where it was typically put by a NOFORMAT transfer). For output, the data is not sent to the external device, but to a variable. From there it can later be sent to the external device by means of a NOFORMAT transfer — at very high speed.

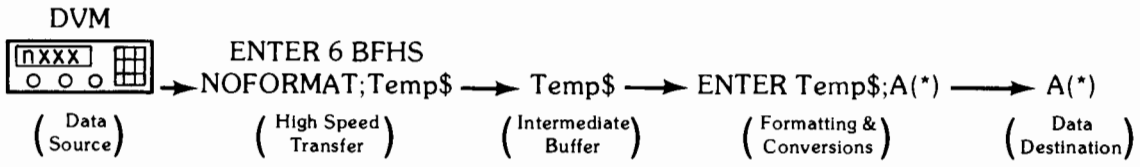
The diagram below depicts a combination of NOFORMAT and Variable-to-Variable transfers to output a waveform data base to a d/a (digital-to-analog) converter:

OUTPUT



The diagram below shows the relationship between NOFORMAT transfers and Variable-to-Variable transfers for an input operation.

INPUT



Let's take the case of a high-speed DVM for an example. Our program is to take 1000 readings per burst, and we want five bursts of data to process. To increase the speed of the transfers, we will use a NOFORMAT transfer. Since the DVM transfers six characters of data per reading, we need a string array with five elements of 6000 characters each. Our program has a pointer into the string array, I, which keeps track of the string element currently receiving the data. As a burst transfer completes, the pointer is incremented and another transfer is initiated, until all five string elements are filled. At that time, the data is stored on the tape for future processing.

Our program merely inputs the data as rapidly as possible: any formatting or conversions to be done must be handled later, which is the subject of the next section.

```
10      OPTION BASE 1
20         OVERLAP
30      DIM Temp$(5)[6000]
40      ON INT #6,3 GOSUB Dvm_complete
50       I=1
60       ENTER 6 BFHS 6000 NOFORMAT;Temp$(I)
70        IF I<=5 THEN 70
80        !  Further processing could be done here.
90        !
100      ASSIGN #1 TO "Save"
110      PRINT #1;Temp$(*),END
120      !
130      PRINT "ALL DATA SAVED"
140      STOP
150      !
160      !
170 Dvm_complete:        I=I+1
180       IF I>5 THEN Exit
190         !  ELSE  Start next data transfer.
200         ENTER 6 BFHS 6000 NOFORMAT;Temp$(I)
210 Exit:       RETURN
220      END
```

The following program takes the data saved by the previous example and processes it one burst (1000 readings) at a time:

```
1           OPTION BASE 1
10        DIM Dvm_array(1000),Temp$[6000]
20        ASSIGN #1 TO "Save"
30         FOR I=1 TO 5
40           READ #1;Temp$
41           ! Note that each value is separated by a LF.
42           ! This LF delimits the numbers in Temp$.
50           ENTER Temp$;Dvm_array(*)
             .
             .     Process the data here
             .
120        NEXT I
130       END
```

Instead of appending the L/F to Temp$, we could also have specified an IMAGE reference in our ENTER statement that disabled the normal L/F requirement to terminate the ENTER statement.

As we have seen, the Variable-to-Variable transfer extends the capabilities of Noformat transfers beyond the scope of computer-to-computer data communications. It also allows the programmer to separate the actual data exchange from the formatting and conversion process. There are some incidental ramifications to this separation, like the fact that during a NOFORMAT transfer, processor time is not being utilized for data formatting so program execution rates increase during Interrupt, standard-Handshake, and DMA transfers. (Recall that Fast-handshake transfers dedicate the processor for the duration of the transfer, so a computer like the System 35 will not be doing any program execution at all once a Fast-handshake transfer is initiated.)

The user should be aware, however, that when a NOFORMAT transfer is used in conjunction with a Variable-to-Variable transfer, the overall I/O throughput is decreased. This is because more system overhead is necessary to complete the transfer. The following two program segments are equivalent, but the second segment takes longer to execute:

```
50      ENTER 6;A,B,C,D
```

```
50      ENTER 6 NOFORMAT;TRL(10),A$
60      ENTER A$;A,B,C,D
```

The actual raw **data transfer** from select code 6 is faster in the second segment, but the **throughput** is decreased because of the necessity of executing two statements rather than one. Such a trade-off is desirable in certain cases, however, such as overlapped ENTER (for taking data from multiple devices simultaneously) or maximum transfer rate (such as taking readings from an HP3437 DVM for waveform analysis). The I/O programmer must weigh the advantages against the disadvantages of each method and design the program to accomplish the desired result.

One last item should be mentioned here, and that is: what happens when the variable name is an expression? In other words, assume A(1) = 4. How does

```
10   ENTER A(1);V
```

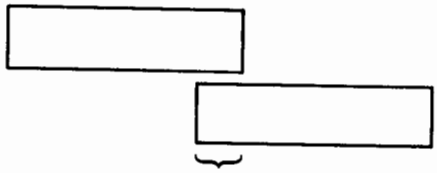
differ from

```
20   ENTER A(1)*1;V
```

In line 10, a Variable-to-Variable transfer takes place — the source is A(1) and the destination is V. In line 20, a normal select code ENTER occurs — the source is select code 4 and the destination is V. The difference is caused by specifying an **expression** (any expression will do: A(1) + 0 too) rather than a simple variable name.

Overlapped I/O

"Overlap" is a term generally used to express the concept of an object extending over and covering part of another object in space, like so:



As the term is used in the BASIC language of the System 35/45 desktop computer, "overlap" means the extension of one process over another in time. The processes involved can be program execution and one or more I/O transfers.

You have already dealt with the concept of overlap when reading about interrupt transfers. In the simplest case, the processor executes the BASIC language program, occasionally pausing in the process to transfer the next item of data to the external device as that device indicates ready (by interrupt). This is how program execution can overlap with I/O transfers.

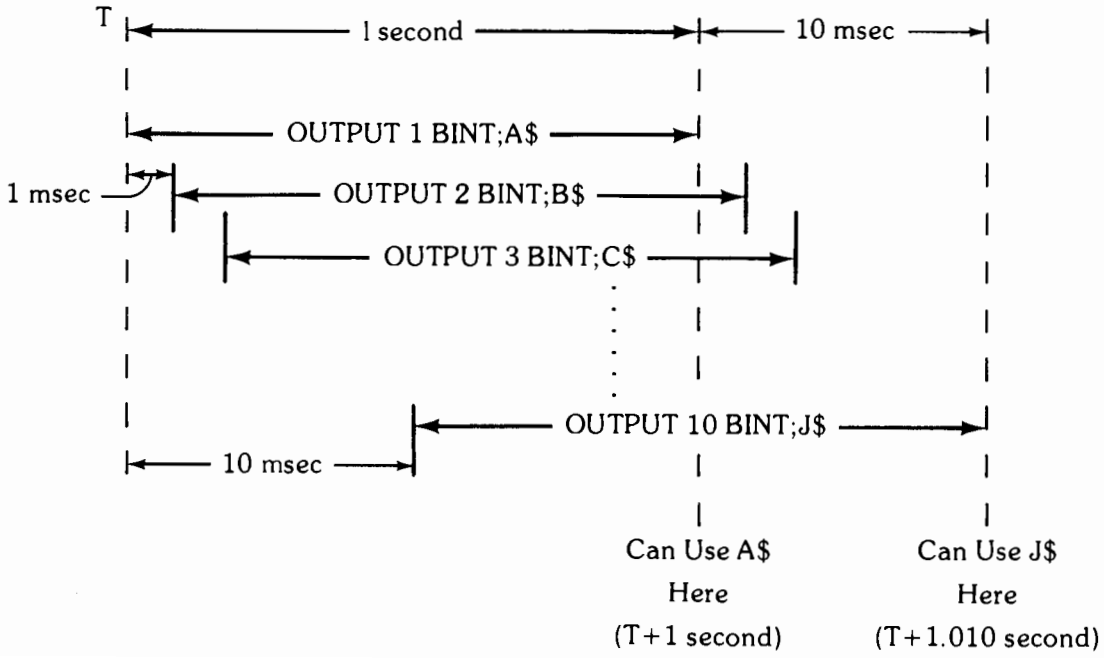
Taking this concept a little further, picture the processor dealing with several external devices by means of interrupt transfers. The processor now spends less of its time executing the user's BASIC program and more of its time shuttling back and forth transferring data with each device as that device indicates that it is ready for (or with) data. The System 45 computers have separate processors for I/O and program execution, so program execution is not usually affected by I/O transfers. The I/O processor takes care of the shuttling back and forth servicing the various transfers. Now we can see in our minds how various I/O transfers might overlap with each other.

When an OVERLAP[1] statement is executed, the processor can allow various I/O transfers to occur concurrently. This is possible because in Overlap, an ENTER or OUTPUT statement merely initiates the transfer process, leaving the processor free to continue program execution — which may include further ENTER and OUTPUT statements! This can be illustrated by a simple example:

1 OVERLAP must be specified in your program or operations are performed serially.

```
1000 OVERLAP
1010 OUTPUT 1 BINT;A$
1020 OUTPUT 2 BINT;B$
1030 OUTPUT 3 BINT;C$
1040 OUTPUT 4 BINT;D$
1050 OUTPUT 5 BINT;E$
1060 OUTPUT 6 BINT;F$
1070 OUTPUT 7 BINT;G$
1080 OUTPUT 8 BINT;H$
1090 OUTPUT 9 BINT;I$
1100 OUTPUT 10 BINT;J$
1110 !   Now use the variables...
1120 A$="NEXT DATA FOR A$"
1130 B$="NEXT DATA FOR B$"
      .
      .
      .
      .
1210 J$="NEXT DATA FOR J$"
1220 GOTO 1010
```

Lets assume that each output transfer takes one second to complete, and that each program statement takes one millisecond to execute. If we did not execute the OVERLAP statement, the total time might be expected to be 10.010 seconds to output the data and reach line 1120. In OVERLAP, however, the time required is closer to 1.010 seconds, almost an order of magnitude faster. Each one second transfer is now "overlapping" with the others and with the program execution, as shown below:



Naturally, certain constraints must be placed on this course of events. The processor's firmware must determine if successive I/O statements are directed to the same device, in which case these successive statements are put in a "queue", or a waiting line. They are waiting for a "system resource," in this case the external device. As the first transfer completes, the firmware checks to see if any transfers are still pending to the now free device, and takes the first one in line.

Suppose your program looked like this:

```
10        DIM A$[80],C$[80]
20       OVERLAP
30       ENTER 6 BINT;A$
40        C$=A$[4;10]
50       PRINT C$
60       END
```

Line 20 frees the processor to continue program execution and other I/O even if an I/O transfer is in progress. Line 30 initiates an input transfer from the device on select code 6 into variable A$, Line 40 takes a substring of A$ and places the result in C$, which is printed in Line 50.

What happens when the processor tries to execute Line 40?

If the input transfer started in Line 30 is not yet complete, the processor has to wait. The variable A$ may be in some intermediate state of being modified by the transfer in progress, and it may change even as it is being accessed for the substring operation! Therefore, variables being written into by an input transfer are flagged as "busy." Now, any time the processor accesses a variable, it can check to see if that variable is "busy" before using that variable. This avoids the possibility of getting "garbage" values from a variable. As you might have guessed, program execution in the above example pauses at Line 40 until the input transfer from select code 6 completes.

Notice the presence of a string variable rather than a numeric variable in the above example. The string input was not chosen by chance for an example of overlapped input — it was chosen because neither freefield nor formatted numeric ENTERs can overlap. This fact is indicated by the qualification "string variable only" in the Capabilities table at the end of this section.

The reason that numeric input does not overlap (unless it is NOFORMAT) is because the numeric formatting, (or changing ASCII characters into internal number formats) is done by the processor that executes the BASIC program. The I/O processor simply takes the incoming ASCII characters and delivers them to the language processor for conversion. This means that the language processor must wait for the character input to complete before it (the processor) can finish assembling the numbers in the ENTER statement and continue on with the next program line.

Looking at the System 35 ENTER/OUTPUT Capabilities tables at the end of this section, some of the yes/no's start to make more sense. The Overlap Compute column tells us which of the transfer types allow the processor to continue program execution once the transfer has been initiated. The regular and the Fast-handshake transfers require the processor's attention to the data handshake line of the external device, so you might expect that program execution would be held up. The "No's" in the Overlap Compute column verify this.

You'd also expect that the processor could not afford the time to service an interrupt from an external device while it is in a Fast-handshake transfer. And indeed, our table bears this out. Interrupts are disabled for the duration of a Fast-handshake transfer. With DMA transfers, however, there seems to be a discrepancy between input (ENTER) and output (OUTPUT) DMA transfers in the overlap compute column. Since DMA transfers require no firmware intervention, the system should be able to continue program execution while the transfer is in progress, yet it doesn't overlap computation with a DMA input transfer. Why is this? The actual reason lies in the difference between the way the ENTER and the OUTPUT formatting is handled by the I/O ROM firmware: although the language processor (executing the user program) is not dedicated to the ENTER's data transfer, it must wait for the data characters to be input so they can be put into correct internal format. Thus, no program lines will be executed. OUTPUT transfers differ in that the language processor simply delivers a copy of the variables and constants in the data list to the I/O processor for outputting. The language processor is then free to execute the user's BASIC program.

Looking at the table section labeled "NOFORMAT," we can see that it is possible to execute a NOFORMAT DMA input transfer that allows computation to overlap with the transfer. Again, why is this? It's because we have done away with the need to format the incoming data, so that — in a way — we have circumvented a restriction imposed by the system. The I/O ROM's flexibility can go to work for us, and by using the NOFORMAT transfer in conjunction with the Variable-to-Variable transfer, we can obtain true input overlap capabilities.

SYSTEM 35
I/O CAPABILITIES TABLES

ENTER Overlap Modes

Transfer Type	Freefield/USING		NOFORMAT	
	Overlap Compute:	Overlap with Transfer Types:	Compute:	Transfer Types:
BYTE (Byte handshake)	No	INT,DMA (Note 1)	No	INT,DMA (Note 1)
WHS (Word handshake)	No	INT,DMA (Note 1)	No	INT,DMA (Note 1)
BINT (Byte interrupt)	For string variable only.	INT,DMA, Handshake(HS)	Yes	INT,DMA, Handshake(HS)
WINT (Word interrupt)	For string variable only	INT,DMA, Handshake(HS)	Yes	INT,DMA, Handshake(HS)
BFHS (Byte Fast-handshake)	No	None	No	None
WFHS (Word Fast-handshake)	No	None	No	None
BDMA (Byte DMA)	No	INT, Handshake(HS)	Yes	INT, Handshake(HS)
WDMA (Word DMA)	No	INT, Handshake(HS)	Yes	INT, Handshake(HS)
Var-Var Variable-to-Variable	No	All (Note 1)	Not Applicable	Not Applicable

Note 1: These transfers must already be in progress.

OUTPUT Overlap Modes

Transfer Type	Freefield/USING		NOFORMAT	
	Overlap Compute:	Overlap with Transfer Types:	Overlap Compute:	Overlap with Transfer Types:
BYTE (Byte handshake)	No	INT,DMA (Note 1)	No	INT,DMA (Note 1)
WHS (Word handshake)	No	INT,DMA (Note 1)	No	INT,DMA (Note 1)
BINT (Byte Interrupt)	Yes	INT,DMA, Handshake(HS)	Yes	INT,DMA, Handshake(HS)
WINT (Word interrupt)	Yes	INT,DMA, Handshake(HS)	Yes	INT,DMA, Handshake(HS)
BFHS (Byte Fast-handshake)	No	None	No	None
WFHS (Word Fast-handshake)	No	None	No	None
BDMA (Byte DMA)	Yes	INT, Handshake(HS)	Yes	INT, Handshake(HS)
WDMA (Word DMA)	Yes	INT, Handshake(HS)	Yes	INT, Handshake(HS)
Var-Var Variable-to-Variable	No	All (Note 1)	Not Applicable	Not Applicable

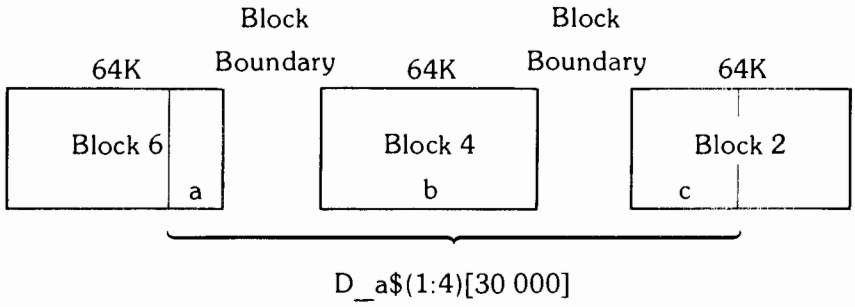
Note 1: These transfers must already be in progress.

Memory Organization and I/O Programming

Transfer Glitches

The memory organization of the desktop computer has a definite impact on certain I/O transfers, notably DMA and fast-handshake transfer types. The 9835/45B I/O ROM manuals mention the need to have a variable reside totally within one memory block (64K bytes of memory) in order to avoid pauses, or "glitches" in the transfer rate.

Due to the method of addressing used by the I/O processor in the 9835/45B, any transfer that includes data outside of the current 64K memory block being addressed must be done in segments that terminate on the block boundaries. Assume, for example, that we have a 120K character string array to send to a D/A converter (using DMA) in order to simulate a complex waveform. Due to the order in which variables were dimensioned, our string array, D_a$ straddles both boundaries of Block 4 as shown below.



The DMA transfer of D_a$ proceeds in three stages, first segment "a" is set up and transferred, then segment "b" and finally segment "c." Although each segment is transferred at the DMA rate, there is a pause, or "glitch," in the transfer as the computer sets up new pointers for the transfer of the next segment. (In the hypothetical example above, our complex waveform becomes more complex than we intended!)

The general guidelines to follow in order to avoid these glitches are given in Appendix F of the apppropriate I/O ROM manual for your computer. There is, however, another less likely possibility that can occur with DMA transfers crossing block boundaries, and is the subject of the next section.

DMA Termination

This event can create a problem if a DMA transfer is prematurely terminated by the peripheral exactly at a block boundary. At the block boundary, the hardware DMA counter decrements to zero and interrupts, indicating the block transfer is complete. If an interrupt from the peripheral is received simultaneously (through the EIR line of the 98032A interface) to force early termination of the transfer, that early termination interrupt cannot be differentiated from the interrupt from the DMA counter.

Hardware DMA transfers can only be a maximum of 64K bytes (32K words, or 1 block). Therefore, any transfer that includes a variable or array which crosses block boundaries must be done in segments. Each segment is treated as a complete, logical DMA transfer by the processor, so each segment must be set up, initiated, and terminated. If the transfer terminates with the DMA counter equal to zero, then the whole block or segment was transferred successfully. The next segment (if there are any remaining) is then set up and initiated.

If the data segment being transferred is simultaneously terminated by a zero DMA count and by the peripheral interrupting on the EIR line, then the I/O ROM firmware cannot detect the peripheral's request for early termination. The firmware will go on with the process of setting up and initiating the DMA transfer for each remaining data segment.

Obviously, the logical methods to use to avoid missing an early termination interrupt from the peripheral are either to transfer 64K bytes or less, all within a memory block, or to use fast-handshake, if possible, to transfer these larger amounts of data.

Chapter 3
HP Interface Cards

Interfacing and the Computer I/O Bus

In Chapter 2, we discussed the various methods of programming for I/O operations, and treated the interfaces themselves as "black boxes" which could be described by the register model (R4, R5, R6, and R7). All input and output operations were described in terms of sequences of reads and writes between the computer and the interface registers; and indeed this register model is sufficient for writing interfacing programs. In the following sections, we will go into more detail about the actual structure of the interface cards and how this register model is implemented. This information will be helpful in actually connecting peripheral devices to the interface cards and configuring I/O systems.

Each of the interfaces has its own installation and service manual which contains detailed information about the circuits used, the lines available, and the general operational characteristics of the card. It is not the intent of this guide to duplicate the information contained in those manuals, but rather to describe and give examples of the intended use of the various capabilities that exist on each card so that the user may understand and make maximum use of these capabilities. This information should be helpful in deciding which interface card to use in a particular application, and to recognize which control features of that interface are best suited to serve the needs of that application.

Of the four interfaces described in the following sections, the 98032A 16-Bit Parallel Interface is the most versatile and general purpose. This is the card used to interface with many of the current HP peripherals available for the desktop computers such as printers, plotters, tape punches and readers, card readers, and flexible disk drives. It is also the interface that is used most often when the user wants to connect his own special-purpose or customized peripheral into the system. The versatility of this card allows it to support a wide range of special requirements in interfacing to such devices.

Many instruments and measuring devices present their data in a special format called BCD or binary-coded-decimal. This format is frequently found in digital voltmeters, multimeters, and other measuring instruments. The 98033A BCD Interface was specifically designed to accept inputs from these devices and to convert those inputs into a format that can be read by the computer. Whenever a device with BCD outputs is used, this card will usually prove to be the easiest one to interface with.

The task of interfacing a peripheral device to the computer would be greatly simplified if the use of data and control lines, logic levels, connector configurations, and operating protocol were standardized. If a group of computing controllers and peripheral devices were to adopt this standard, then these devices would be "plug to plug" compatible; and the job of interfacing them to one another would be reduced to simply plugging them together. The HP-IB(Hewlett-Packard Interface Bus) provides this kind of compatibility. The structure and the format of the HP-IB has become so popular that the Institute of Electrical and Electronic Engineers has adopted it as a standard (IEEE 488-1978) and today dozens of manufacturers provide hundreds of devices which are compatible with that standard. The 98034A HP-IB card is also available to allow HP desktop computers to participate on this standardized bus. The section of this guide covering that interface goes into more detail concerning the intent and the use of the HP-IB.

A fourth broad area of interfacing deals with data communications. This area is used primarily for information exchange between computers over long distances, although many applications are found for peripheral communications (e.g., remote terminals connected to a central computer) and local computer networks. The special requirements of this type of interfacing are discussed in the section on the 98036A card which provides HP desktop computers with an access link into the area of data communications.

Before describing each of these four types of interfaces in detail, we will first look at that portion of these cards which is common to all of them — namely, the edge of the card that connects to the computer I/O backplane. We saw in Chapter 2 how high-level BASIC statements such as WRITE BIN, OUTPUT and ENTER are translated by the I/O ROM into sequences of read and write operations with the interface registers. A special segment of the computer's system called the I/O processor is responsible for converting the machine language instructions which address these registers into a set of electrical signals which will cause the interface to perform the desired operation. These signals are made available to the interfaces at a connector called the computer backplane or simply the I/O bus. It is called a bus because many interfaces can be connected to it in parallel and all of them use the same bus over which to communicate their signals. All of the interfaces are connected to this bus in a "wire-and" configuration and passively allow the line to float high. The one interface that is currently selected to put its

information on this line allows it to remain high if it requires it to be high, or pulls it to ground if it requires it to be low. Thus, to the I/O processor, it appears as though only the selected interface is connected to the bus at the time information is requested from that card.

The mechanism by which one card on the bus is selected to present its information to the I/O processor is called the select code method. Each interface is assigned a select code in the range 0-15. Internal devices (keyboard, display, printer, tape cartridge) have their select codes preset or "hard wired." External interfaces have their select codes set by an externally accessible rotary switch on the card itself. When the I/O processor wishes to communicate with a given interface, it takes the select code parameter from the program's BASIC statement (e.g., OUTPUT 6,. . .) and converts it to a 4-bit binary equivalent (in this example, 0110) which it presents on the four peripheral address lines (PA0 through PA3) to the interface. Only that card whose select code switch matches the bit pattern being presented on the peripheral address lines will respond.

But what is the nature of this response? This question is answered by looking at the remainder of the lines used for communication between the I/O processor and the interface.

The first immediate response is to set the state of the status (STS) and the flag (FLG) lines. The status line is a 1-bit indicator that tells the I/O processor whether the selected interface (and possibly the peripheral device connected to it) is operational or not. The flag line indicates whether the interface is still busy processing the last task given to it by the I/O processor, or is ready for another operation. (See the Handshake Process section.) If the status and flag lines are both low[1], the I/O processor may proceed with the next operation.

As we discussed in the previous chapter, all operations with the card are accomplished by writing to or reading from the 8 interface registers. The I/O processor has 16 data lines (DIO0 through DIO15) available for this purpose. We will see later that some interface cards use all 16 of these lines while others may use only 8. Indeed, some interfaces may use different numbers of these lines for different registers. For example, the 98032A Bit-Parallel Interface uses all 16 lines for data transfer (R4IN and R4OUT), but only 8 for taking its control byte (R5OUT) and presenting its status byte (R5IN).

Since the same set of data lines are used to exchange information between the I/O processor and all eight interface registers, some means is necessary to inform the interface which register is being addressed (R4, R5, R6, or R7) and whether the required operation is IN or OUT. This is accomplished by the IC1 and IC2 lines which indicate the register number, and the DOUT line which indicates the direction. The following table gives the status of the IC1 and IC2 lines used to address the four register numbers.

1 Since all lines on the I/O bus use negative-true logic, High = 0 = False and Low = 1 = True.

The DOUT line is high for input (interface to I/O processor) and low for output. When the I/O processor requests an input from an interface register, the card uses the FLG line to indicate when the information on the data lines is valid (high = busy, low = ready). The I/O processor also needs a means of telling the interface that information on the DIO lines is valid when it is conducting an output operation to the interface registers. It does this by momentarily pulsing the I/O Strobe (IOSB) line low. On this signal, the interface routes the information on the DIO lines to the proper register and latches it.

These lines then provide the basic operations of the interface: selecting a specific card, checking that it is operational and ready, and exchanging information between the I/O processor and the interface registers. The remaining lines on the I/O bus are used for implementing special functions.

The first of these special functions is the ability to initialize the cards. When power is turned on or the RESET key is pressed, an initialization line (INIT) is momentarily pulsed low. This tells all of the interfaces connected to the I/O bus to reset their latches and flipflops to some standard initial state. (This signal is also made available on the peripheral side of some interfaces so that the attached device is also given a chance to re-initialize.)

Three other lines are used to provide interrupt capability. If an interface has been enabled to interrupt on a certain condition and that condition occurs, the card responds by pulling and holding low either the IRL or the IRH line. If its select code is 0-7, it will request a low level interrupt on the IRL line; if it is 8-15, it will request a high level interrupt on the IRH line. In either case, if that level of interrupt is not already in use (see "Interrupt Priorities," Chapter 2), the I/O processor will poll the interfaces to determine which one requested the interrupt. It does this by setting the interrupt line (INT) low. During this interrupt poll, the PA0 line is used to indicate whether the I/O processor is polling the low level interrupts or the high level interrupts. During a low level poll, each interface responds on the DIO line that corresponds to its select code (select code 7 on DIO7, etc.), setting it high if it was not requesting interrupt and low if it was. The I/O processor thus determines which of the cards was requesting service. If two or more cards on the same level are requesting service at the same time, the one with the higher select code will be granted the interrupt first. During a high level poll, each card responds on the DIO line that corresponds to its select code minus eight (select code 15 on DIO7, etc.).

Finally, a special line (DMAR) is used for the interface card to request a DMA cycle if it has been enabled to do so by the I/O ROM (see "Interrupt Priorities," Chapter 2).

The remaining three lines are for providing electrical power to the card. These are the + 5 volt supply, a ground line, and a shield line. All of the interfaces get their power from the computer's main power supply. The capacity of this power supply was designed to accommodate the computer itself plus the number of interfaces that can be plugged directly into it. As a result, the power supply line on the I/O bus should not be used to provide power for any external devices. Doing so could result in erratic operation or even damage to the computer's power supply circuitry. The logic ground line is meant to provide a zero volt reference point from which other voltages are measured. This logic ground is brought out on the peripheral side of the card and should be connected to the logic ground line of the attached device so that all signals are measured from a common reference point. The shield line is provided in order to ground the metal casing used in the cable connecting the interface to the peripheral, thus reducing the amount of radio frequency interference (RFI) emitted. The shield line is connected to the logic ground inside the computer and should not be connected to the ground or any other line on the peripheral. Otherwise, ground loops will be created leading to erratic operation.

Up to now, we have discussed the operation of the interface cards from the computer's I/O bus, and the description given is common to all of the interface cards. When information is exchanged with the interface registers on the cards, the resulting action at the peripheral side of the card depends on which interface is being used. Indeed, it is the difference in requirements at the peripheral side that necessitates having the various types of interface cards. What these differing requirements are and how they are implemented is the topic of the following sections.

Interface ID and Card Types

Each of the HP 9803x series of interface cards has unique characteristics suited for the needs which they were designed to satisfy. As a result, even though they can all be described by the same register model as presented in an earlier section, the specific use of these registers and the order in which they are addressed will differ from one card to the next. For example, when data is read into a buffer under interrupt, the exact sequence of R-register operations will differ slightly depending upon whether the data is coming through a 98032 GPIO card or a 98034 HP-IB card. For this reason, the input driver (i.e., ROM instructions for reading from the interface) must be able to distinguish among the various interface types.

In order to do this, two bits of the status byte (R5-in) have been assigned as identifier or ID[1] bits. These are bits 5 and 4 of the status byte and have the following meaning.

1 These ID bits are contained in the R5-in register which is not the register through which HP-IB status bytes are received. As a result, you do not see the ID bits for the HP-IB card when the STATUS function is executed. See the description of the HP-IB card for more details (page 112).

Card Type	ID Bits 5	4	Interface Card Type
0	0	0	None (no interface on this select code)
1	0	1	Serial I/O Interface (98036)
2	1	0	Gen Purpose Card (98032,98033,98035)
3[1]	1	1	HP-IB Interface (98034)

As the table shows, most interface cards are type 2 and all use the same protocol or register access sequence. The HP-IB interface is different and requires a special protocol that allows it to perform the wide variety of bus functions. Although the Serial I/O card is functionally the same as the type 2 cards, giving it a unique interface ID type allows it to be distinguished from the others.

If the ID bits are both zero, this identifies a so-called empty slot; that is, there is no interface set to the specified select code. This means that all properly-operating interface cards of the 9803x series will never return a value of zero for the status byte since at least one bit of the status byte is always a one.

The Use of the Control Register

In the following sections dealing with the interface cards themselves, we will see that the R5 OUT control register is used to access the programmable capabilities contained on those cards. For example, the control register of the 98032A Interface is used to reset the card, turn on and off the interrupt, DMA, and auto-handshake options, and to set and clear the two user-defined control bits. The BASIC language provides three statements for output to this control register; namely, the wait write (WAIT WRITE), the card enable (CARD ENABLE[1]), and the write interface (WRITE IO) statements. Each of these statements has slightly different operational characteristics which will be discussed in this section. These characteristics are summarized in the table below.

Statement	Masked	Immediate	Saved	Wait for FLG (flag)
WRITE IO	no	yes	no	no
WAIT WRITE	no	yes	no	yes
CONTROL MASK	---	no	yes	---
CARD ENABLE	yes	no	---	yes

Under different conditions the control mask is written into R5 OUT by OUTPUT and ENTER statements.

Simple ENTER and OUTPUT statements do not write the control mask to R5 OUT. ENTER or OUTPUT using an INT or DMA option both copy the control mask at the beginning of the

1 The CARD ENABLE statement takes the current value of the control mask (set by CONTROL MASK) and writes it to the R5 OUT register.

transfer. An OUTPUT with the FHS option does not write the mask to R5OUT. An ENTER with FHS options writes the control mask after the transfer has completed (when the ENTER is satisfied).

When the I/O ROM is present, the program may take advantage of the interrupt capability of the interface cards. In this case, the CARD ENABLE statement is used to allow the card to interrupt when it finishes the last operation and again comes ready. The CARD ENABLE statement "arms" the I/O ROM firmware for interrupt service. For example, if we were to do something to make the interface go busy, and then execute the statements

```
ON INT #6, 1 GOTO User-service
CONTROL MASK 6; 128
CARD ENABLE 6
```

the interface on select code 6 would interrupt (and the program would be sent to the user's service routine by the I/O ROM) when the operation was completed and the card came ready. A mask value of 128 was used, which set only the interrupt enable bit (see Figure 32). If we also wanted bit 0 set (CTL0), we would have used instead

```
CONTROL MASK 6; 129
CARD ENABLE 6
```

When data buffers are being transferred under interrupt (BINT), the CARD ENABLE statement is not normally used. That is, the I/O ROM automatically takes care of setting and clearing the interrupt-enable bit at the appropriate times during the transfer process. In this case, to set a control bit, we would use the WAIT WRITE statement. To show the difference between WAIT WRITE and CARD ENABLE, let's consider an example where data is to be read from a peripheral using an interrupt transfer, and where CTL0 is used to set the peripheral device into some desired state. That is, if CTL0 is set the device will operate one way, and if it is clear it will perform in another manner. Suppose then that we execute the statement WAIT WRITE 6,5;1 to set CTL0, which is the mode we want the device to be in for this particular application. If we now execute an interrupt[1] transfer (OUTPUT 6 BINT;A$), the I/O ROM will send a 128 to the control register to enable interrupts, and as a result **the CTL0 bit will be cleared** causing the device to switch to the other mode of operation, which is not the one we require for this task. How then do we do the transfer operation without losing the setting of the CTL0 bit?

The CONTROL MASK statement provides the solution. If we had used the statements

```
Control_byte = 1
CONTROL MASK 6;Control_byte
WAIT WRITE 6,5;Control_byte
OUTPUT 6 BINT,A$
```

1 DMA transfers also write a control byte to the interface. A CONTROL MASK statement should also be used with DMA transfers to avoid unwanted changing of the control bits.

to set the CTL0 bit, two things would have happened. The control byte (1 in this case) would have gone to the R5 OUT control register, and CTL0 would be set. But in addition, a copy of this control byte would be saved by the I/O ROM because the CONTROL MASK statement was executed with the Control_byte value. Then, anytime the I/O ROM needed to change the state of the interrupt bits, it would automatically retain the state of the lower four bits from the saved value of the last CONTROL MASK statement to that select code. Thus, the setting of CTL0 would remain unchanged during the entire transfer process.

It is important to note that while the CONTROL MASK statement is used primarily with the CARD ENABLE statement to enable interrupts, here is a case where it is not. The CONTROL MASK statement here merely sets up a control byte value in such a way that it is retained by the I/O ROM, and does not actually enable the card for interrupt.

The 98032A Bit-Parallel Interface

General Operational Characteristics

The 98032A Interface is the most versatile, general purpose card and is used most often to connect the computer to those devices which do not conform to some standard format and protocol such as BCD or HP-IB.

It can output data to a peripheral using up to 16 bits at a time in parallel, and it can input data from the same or a different peripheral over an independent set of 16 parallel input lines. These input and output lines can be further partitioned into two sets of 8-bits each for handling special applications. The card can be configured to accept a wide variety of signals from the peripheral and indicate when input data is ready or output data has been accepted. These and other capabilities will be discussed in the following pages.

There is no restriction on the interpretation to be placed on the data being sent or received. If the card is being used, for example, to interface to an A/D (analog to digital) converter, the bits received would probably be interpreted as a binary number whose value is proportional to the voltage being measured. When interfacing to a printer, the bits would represent some alphanumeric character to be printed using some code that the printer can recognize such as ASCII. Or if the data were being input from a card reader, it would simply represent a pattern of ones and zeros corresponding to the presence or absence of punched holes or pencil marks at specific locations on that card. It would then be up to the program in the computer to translate these patterns into meaningful information, based on the design of the card being read. As far as the interface is concerned, each data item is merely a set of 16 bits to be sent or received; and any meaning to be placed on that data is based entirely on an agreement between the computer and the peripheral as to how they will interpret it.

In the previous section, we developed a register operational model that was general to all of the interfaces. The table below gives the specific use of each of these registers by the 98032A Interface.

	IN	OUT
R4	DATA IN	DATA OUT
R5	STATUS	CONTROL
R6	HIGH BYTE DATA	HIGH BYTE DATA
R7	(not used)	TRIGGER

Figure 31

The R4 register is the one through which data is normally sent and received. In Chapter 2, we saw programming examples of how this register is used to exchange data between the computer and the peripheral device. These examples were based on the register model of the interface. Later in this section when we discuss the handshake process, we will see what actually takes place on the card when these registers are accessed. The R6 registers are used on the 98032A when it is operating in the optional "byte mode" and will also be discussed later.

The R7 input register is not used by this interface, and if a READIO<sc>,7;A operation is performed, the result will always be a zero. The R7 out register is used to trigger either an input or an output operation. The actual byte output to the R7 register does not matter. It is the act of writing to this register itself that causes the read or write operation to be triggered. This register is also covered in the following section on the handshake process.

The R5 register is always used as a communication link between the computer and the interface itself. The 98032A has certain modes of operation that are programmable by the computer. The I/O processor can make the card go in and out of these modes by outputting specific bit patterns called the control byte to the R5 register. The bit assignments for the control byte on the 98032A are given in Figure 32.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INT	DMA	RESET	AH	X	X	CTL1	CTL0

INT: Interrupt Enable on FLG = Ready
DMA: Direct Memory Access Enable
RESET: Reset the Card to Its Power-on State
AH: Auto Handshake Enable
X: These bits are not used and may be a 1 or a 0
CTL1,0: General User-definable Control Bits

Figure 32

When bit 7 of the control byte is set (= 1), the interface is enabled to request an interrupt to the I/O processor whenever the FLG line on the card indicates ready. We have already discussed in the Interrupt I/O Section how the I/O processor uses this interrupt request to carry out data transfer operations with buffers and other interrupt activities.

Setting the DMA enable (bit 6) programs the card to request a DMA access each time the FLG line comes ready. Thus we see that the interface will do one of three things when the FLG line comes ready. If bits 6 and 7 of the control register are both zero, the card will merely indicate the ready condition on the FLG line but perform no other action. If bit 7 is set, the card will request an interrupt. Or if bit 6 is set, it will request that another word of data be sent or received via the DMA channel. If both bits 6 and 7 are set, the DMA request will override the interrupt request until the DMA transfer is complete (i.e., the count of words to be transferred has been satisfied). When the transfer completes, the card will automatically disable DMA, and the next time the FLG line comes ready, a normal interrupt request will be generated. INT remains enabled until disabled by the I/O processor, while DMA automatically disables itself when the DMA transfer is complete. Since the DMA transfer is a complex operation and is handled automatically by the I/O ROM, the user's program would rarely set or clear the DMA enable bit.

The RESET bit (bit 5) is used to return the card to its power-on or "wake-up" state. On the 98032A, this causes the PCTL handshake line to return to high (control not set), and the programmable conditions of INT, DMA, and AHS to be cleared or disabled. A low pulse is also generated on a peripheral reset line (PRESET) so that the attached peripheral device also can receive an indication that a reset operation has been performed. The action taken on this signal is determined by the peripheral itself. For example, the 9866A/B Thermal Line Printer clears out its built-in data buffer when it sees this signal. This reset action can also be initiated by the INIT line from the I/O processor, which is done whenever the Control-Stop keys on the computer are pressed. While the use of the INIT line resets all interface cards connected to the I/O bus, the RESET bit of the control byte is used to selectively reset only one interface card. It should also be noted that the RESET bit of the control byte overrides any other bits (INT, DMA, AHS) that may be set in that control byte.

Bit 4 of the control byte is used to set a special mode of operation called the "auto handshake" mode. In this mode, the use of the R7 OUT trigger operation (see examples in Chapter 2) is not required. For data output, as soon as the data is placed in the R4 OUT register, it is automatically triggered. For data input, when the current data item is read from the interface data latches (R4 IN), this automatically triggers a demand for the next data item from the peripheral. Again, this mode of operation is used primarily by the I/O drivers in the I/O ROM and is not something with which the user normally need be concerned.

Finally, the control byte provides two general-purpose control bits called CTL1 and CTL0. These lines are made available at the peripheral side of the interface card and may be used for whatever purpose and meaning the user may wish to assign to them. For example, the user may have designed an input device which can either deliver a data item whenever the program requests one, or when an operator at the device presses a GO button. If the device were designed to switch between these two modes of operation based on the state of a control signal externally supplied to that device, one of the two control lines (CTL1 or CTL0) could be connected to the external control line on the peripheral so that the mode of operation could be selected by the computer program. Other examples of the use of these control lines will appear later in this guide.

The interface is also capable of delivering information back to the I/O processor concerning the states that it is currently in. This is known as status information and is read by the processor through an R5 IN operation. For most interface cards this status information is contained in eight bits and is usually called the status byte. This should not be confused with the status line (STS) which is a one-bit indicator of whether or not the interface card and its associated peripheral are operational. The information contained in the status byte is information about the current state of the interface card itself, and not about the peripheral device (except for STI1 and STI0 as explained later).

The status byte is obtained by the program through the use of the read-status statement, STATUS <select code>; <return variable>. This statement takes as its parameter the select code of the desired interface and returns a value which is the decimal equivalent of the status byte. Since the function for testing the 1-bit status line (STS) is contained in the I/O ROM, this bit is added to the status byte as a ninth bit.

The table below gives the meanings assigned to the bits in the status byte by the 98032A Interface, and does not include the ninth bit, STS, added by the STATUS statement.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INT	DMA	1	0	IID	IOD	STI1	STI0

INT: Interrupt Enabled Indicator
DMA: DMA Enabled Indicator
IID: Invert Input Data Jumper Installed
IOD: Invert Output Data Jumper Installed
STI1,0: General User-definable Status Bits

Figure 33. The 98032A Status Register

Bits 7 and 6 of the interface control register (R5 OUT) are used for enabling and disabling interrupt and DMA. The corresponding bits of the status byte are indicators of whether these modes are currently enabled or not; a one indicating enabled and a zero indicating disabled.

Bit 5 is always a 1 and bit 4 is always a 0 for the 98032A card. These are the interface identification bits as explained in the previous section. They allow the I/O processor to identify the type of interface with which it is communicating so it will know what protocol (sequence of register operations) to use, since this protocol is different for the various card types. The examples given in the previous chapter all assumed a 98032A class (type 2) of interface.

The 98032A Interface uses negative-true logic on its data input and output lines. This means that it associates the + 5 volt level with a logic value of zero, and the ground level with a logic value of one. If the particular peripheral attached uses the opposite sense of these logic levels (positive-true logic) then the data needs to be inverted (zeros changed to ones and ones changed to zeros) before the data is used. If this is necessary for either the input data or for the output data, or both, there is a provision on the 98032A Interface to install jumper wires that will indicate this fact. The presence (1) or absence (0) of these jumpers is indicated by bits 3 and 2 of the status byte. It is up to the computer to read these bits and perform the data inversions if necessary. Normally, this is handled automatically by the I/O ROM (the exception being FHS and DMA).

The last two bits of the 98032A Interface are general purpose status bits called STI1 and STI0. They may be connected to any output lines from the peripheral device to monitor any signals that the user finds convenient in his particular application. For example, a paper tape punch might have a line coming out that indicates when the amount of paper tape left is running low. This line could be connected to one of the general-purpose status bits, and then monitored periodically by the program in the computer to warn the operator when the tape is running low. For example, with ST0 connected to the tape low indicator:

```
410       STATUS 6;Check
420       IF BIT(Check,0)=1 THEN Tape_low
```

It is important to note that the contents of the input inferface registers are not related to that of the output interface registers. They serve different functions and are not like memory locations. In general, the bits read from the status register (R5 IN) are not related to any bits in the control register (R5 OUT). The INT and DMA bits were assigned corresponding bit locations for convenience. In particular, setting a control bit such as CTL1 does not affect the state of STI1, since they are usually connected to two different lines on the peripheral and serve different purposes. If he wishes, however, the user can make such an association by physically connecting the STI1 line to the CTL1 line so that the status bit can indicate whether the control line is currently set high or low.

The Handshake Process

In the first chapter, we discussed the handshake process from a user's point of view, giving only enough detail to be able to explain the concept of a handshake. In this section, we will look at that process from a designer's point of view giving the additional information required to be able to actually connect a peripheral device to the 98032A Interface.

Figure 35 shows the complete timing diagrams for both the output and the input handshake operations. In addition to the data lines, four other lines are involved in the handshake process. The meanings and uses of these lines is given in Figure 34.

Name of Line:	I/O	FLG	PCTL	PFLG
Driven by:	Computer	Interface	Computer	Peripheral
High State:	Input	Busy	Clear	Busy
Low State:	Output	Ready	Set	Ready

Figure 34

The I/O line is used by the computer to tell a peripheral device whether an input or an output operation is in progress. For an input only (e.g., paper tape reader) or an output only (e.g., printer) device, this line would not be used. The I/O bus flag line (FLG) is used by the computer to test whether or not the interface is ready for the next operation. The peripheral control line (PCTL) is used to tell the peripheral that the information on the data lines is valid for an output operation, or to request the next data item on an input operation. The peripheral flag line (PFLG) is the ready/busy indicator from the peripheral itself.

The reader may wonder why it is necessary to have separate FLG and PFLG lines. In order to see the reason for this, let's look again at the simplified timing diagram, Figure 12. Here, only the PCTL and PFLG lines are shown in addition to the data lines. For the moment, let's assume that the interface merely connects the computer's FLG line directly to the peripheral's PLFG line, and consider a typical output sequence. After the data is placed on the lines, control is set (PCTL is set at time t_2) to tell the device that it can take the data. At some later time, t_3, the peripheral acknowledges that it has seen control go set by making PFLG go busy, and it begins to read and process the data on the lines. Since there is no restriction on the length of this time interval (t_2 to t_3), it is quite possible that during that interval the computer could be ready to output the next character. It would test the FLG line (which here is the same as PFLG), see that it is indicating ready, and place the next character on the data lines. Since the peripheral had not yet taken the last character, it would be lost. In other words, the computer testing the FLG line only sees a ready or a busy state. It cannot tell whether the PLFG line is indicating ready because it has completed the processing of the data, or because it simply hasn't gotten around to making it indicate busy yet. To avoid these timing problems, the FLG and the PFLG lines are separated. As soon as control goes set, the interface itself makes the FLG line go busy without any response required from the peripheral. The FLG line remains busy until the PFLG line has gone from ready to busy and back to ready again.

We are now ready to follow the complete sequence of events shown in the timing diagram for the output operation in Figure 35. As we do, we will also relate these events to the register operations that are being performed by the output drivers in the I/O ROM.

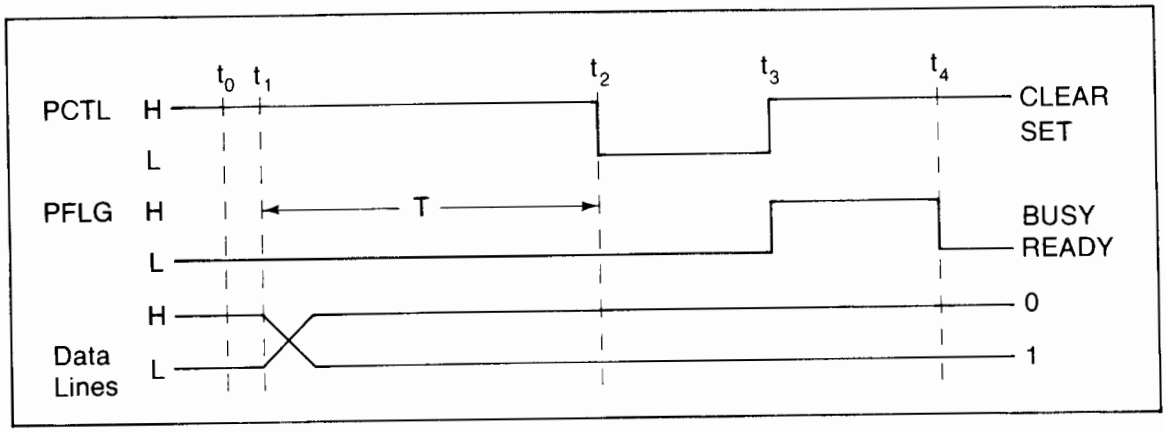
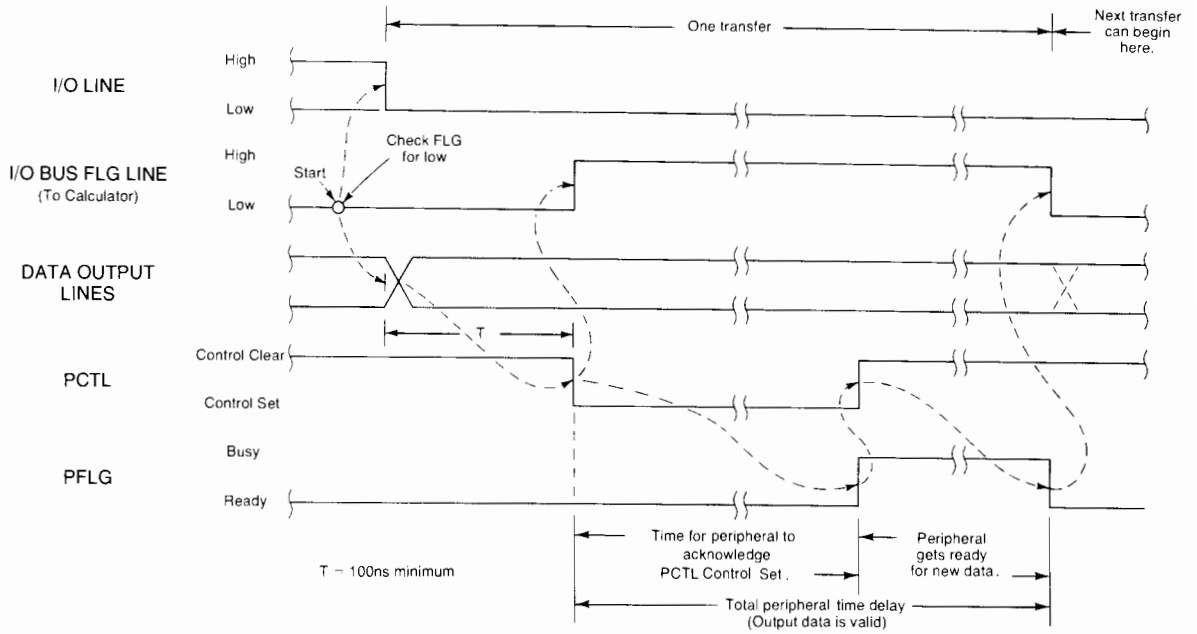
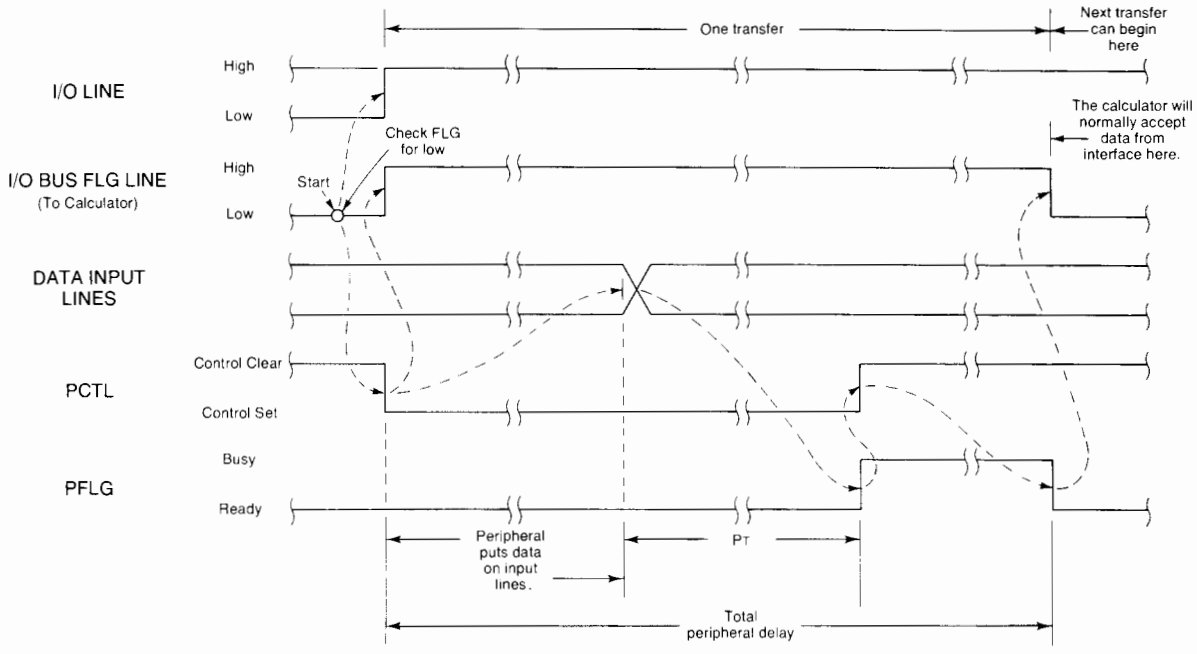


Figure 12. Repeated

Notes





PT = Peripheral time delay to allow data to settle.

1 Interface latches data here if jumper E (Low Byte Clock) or jumper 8 (High Byte Clock) are installed.

2 Interface latches data here if jumper D (Low Byte Clock) or jumper 9 (High Byte Clock) are installed.

3 Interface latches data whenever the register is read by the calculator if Jumper C (Low Byte Clock) or Jumper A (High Byte Clock) are installed (Data Input Lines must be stable).

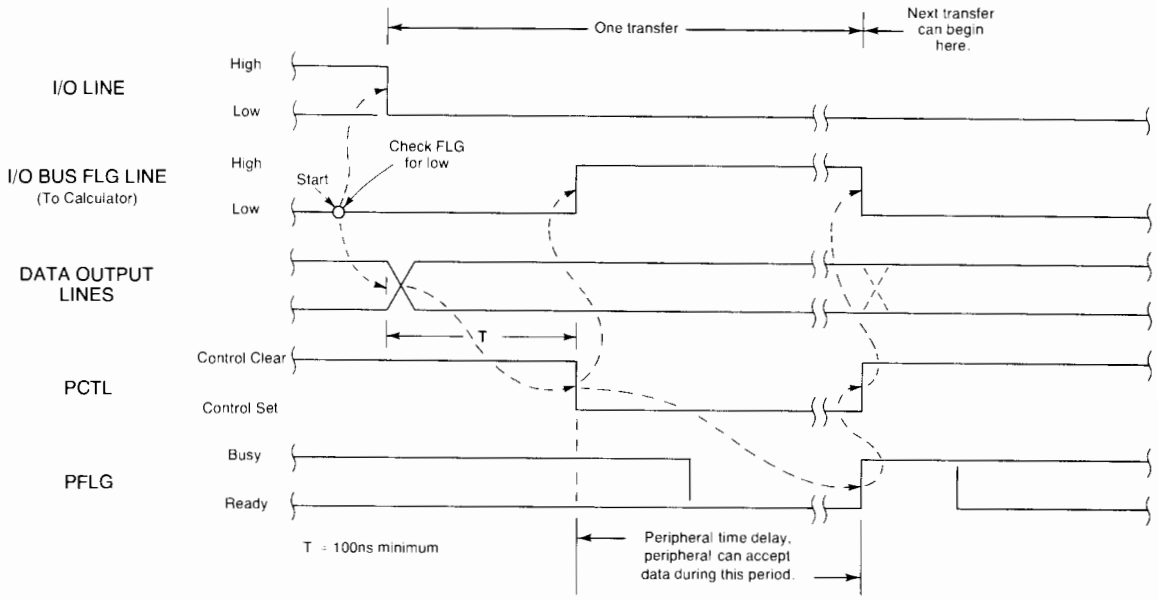
Figure 35. Full Mode Timing Diagram

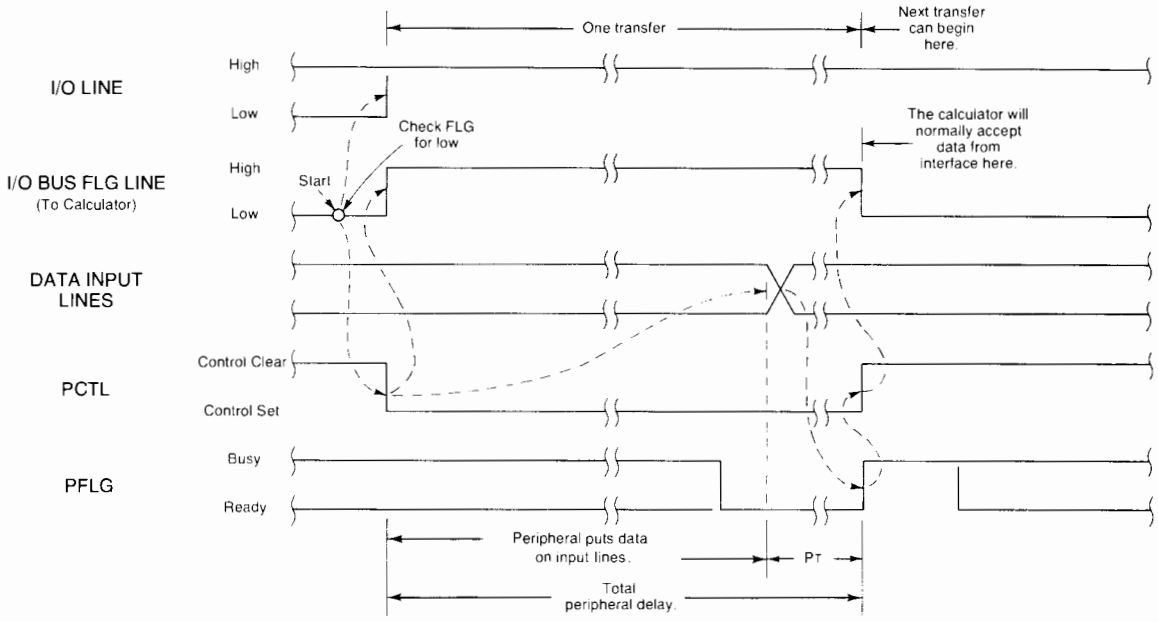
Full Mode Timing

The output drivers first wait for the FLG line to indicate ready, giving the last operation with the peripheral time to complete. When FLG is ready, the data is placed on the lines (R4 OUT operation). Since this is an output operation, the interface sets the I/O line low, to tell the peripheral that an output is about to take place. The I/O ROM then issues the R7 OUT trigger, which causes the interface to set the PCTL line, after delaying long enough to allow the signals on the data lines to settle out. At the same time, the interface makes the computer's FLG line go busy so that it will not try to initiate another operation before this one is completed. At some later time, the peripheral detects that PCTL is set and that I/O is indicating output. It sets its PFLG line to busy, takes the data, and begins to process it. This tells the interface that it can now return the PCTL line back to the clear state, since the peripheral has seen the data and begun its processing. Finally, when the peripheral has completed processing the data, it returns its PFLG line to the ready state. The interface sees this and allows the computer's FLG line to also go back to the ready state and the entire handshake process is complete.

An input operation proceeds in a similar manner. The I/O ROM again waits for the FLG line to indicate ready before initiating any action. When the FLG line is ready, the ROM does an R4 IN operation to set the I/O line to the input state. It then does an R7 OUT operation to demand a data item. This causes the interface to set the FLG line busy, and to set PCTL to tell the peripheral that a data item is being requested. Normally the peripheral would indicate busy on the PFLG line, put the next item on the data lines, and then return PLFG to ready. This will cause PCTL to return to a clear state, and allow FLG to indicate ready. Meanwhile, the I/O ROM has been waiting to see FLG indicate ready. When it does, the ROM does an R4 IN operation to take the information from the data lines and returns the value read to the program.

Because the type of ready (PFLG) signal varies from one peripheral device to another, the 98032A allows for a variety of such signals. By setting jumpers on the interface card, the user may specify that the information on the input data lines be clocked on the ready-to-busy transition, on the busy-to-ready transition, or whenever the R4 IN operation is executed by the I/O ROM, independent of the state of the PFLG line.





PT = Peripheral time delay to allow data to settle.

1 Interface latches data here if jumper E (Low Byte Clock) or jumper 8 (High Byte Clock) are installed.

2 Interface latches data here if jumper D (Low Byte Clock) or jumper 9 (High Byte Clock) are installed.

3 Interface latches data whenever the register is read by the calculator if Jumper C (Low Byte Clock) or Jumper A (High Byte Clock) are installed (Data Input Lines must be stable).

Figure 36. Pulse Mode Timing Diagram

Pulse-Mode Timing

Figure 36 shows the sequence of operations for pulse-mode operations, which is programmed on the 98032 interface by installing jumper 6. The pulse mode is selected when the PFLG signal from the external device indicates "transfer complete" rather than ready/busy.

The primary difference between the handshake mode and the pulsed mode of operation is that in the pulsed mode the state of the peripheral's PFLG line is not reflected by the FLG line to the computer. Instead, the ready-to-busy transition of the PFLG line is used to tell the computer that the peripheral either has data available or has accepted data from the interface. (The actual transition used, ready-busy or busy-ready, is determined by the BUSY, READY, and ALWAYS jumpers; use the ones that correspond to your device's handshake scheme.)

The software operations used in the pulse mode of operation are the same as those used for the handshake mode. For output, the R4 OUT operation puts the data on the lines and sets the I/O line low, indicating an output to the peripheral. Next, an R7 OUT trigger causes the interface to set the PCTL line and to make the FLG line to the computer indicate busy. When the peripheral detects PCTL set it accepts the data and pulses the PFLG line to indicate that it has accepted the data. The interface allows the computer's FLG line to go ready on the appropriate transition of the PFLG line, and the handshake is complete.

For input, the firmware waits for the FLG line to indicate ready then does an R4 IN operation to set the I/O line to the input state. Next, an R7 OUT operation causes the interface to set the PCTL line and to take the FLG line busy. When the peripheral has valid data on the input lines, it pulses the PFLG line to indicate this. The FLG line to the computer is allowed to go ready on the appropriate transition of PFLG, and the firmware then executes an R4 IN operation to read the information from the data lines. The handshake process is repeated until the input is complete.

Word and Byte Modes of Operation

The data lines of the 98032A Interface are divided into 16 input lines and 16 output lines. Each of these sets of 16 lines may be further subdivided into groups of eight for use in special applications. In this section we will look at some of the intended uses of this byte mode of operation.

In all of our previous examples of data exchange using the 98032A Interface, we were operating in the words mode in which we used the R4 OUT operation to place 16-bit data on the output latches, and the R4 IN operation to read 16-bit data from the input latches. If, however, jumpers B or F (see 98032A Installation and Service Manual) are not in place, the input or output latches may be operated in the bytes mode, in which the upper 8 bits and the lower 8 bits of the 16 data lines may be separately addressed by the computer. In this case, the R4 register is now used to access only the lower byte, while the upper byte is accessed through the R6 register. For example, if the 16 input lines contained the 16-bit pattern 0011010101001001, then executing a WAIT READ 4,4;A would give a 73 (binary 01001001) while a WAIT READ 4,6;A would return the value 13568 (binary 0011010100000000). Notice that the high byte is still positioned in bits 8-15 of a 16-bit binary pattern. When either the high byte or the low byte is read, the other byte is replaced with eight zeros. To convert the result of the R6 IN operation to the true decimal representation of that byte, this result must be divided by 256 ($= 2^8$) or shifted right eight places using the bit manipulation functions.

This capability to separately address the high and low bytes is used by the I/O ROMs in implementing drivers for certain peripherals. For example, the 9862A Plotter requires sequences of 12-bit instructions to raise and lower its pen and to move to a new location. Figure 37 shows the format of these plotter control instructions. This protocol is presented merely as an example of the use of the bytes mode on the 98032A card, and the reader need not follow the details of the meanings for the individual bits.

15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
4-bit control	not used	X,Y-position (0,9999)

bit 15: format of data bytes (bits 7-0) is BCD (0) or binary (1).

bit 14: sync bit, set to 1 for the first of a four-word move instruction, and for pen up/down instruction.

bit 13: pen up (0) or down (1) specifier when bit 12 = 1.

bit 12: instruction type bit, move (0) or pen up/down (1).

Figure 37

Raising and lowering the pen is done by sending a control word with bits 12 and 14 set and bit 13 indicating pen up or down. To move the pen to a new location, a four word sequence is required. Since each of the X and Y coordinates is in the range 0 to 9999, two bytes are required to specify each of them. These are sent in four instructions containing X high byte, X low byte, Y high byte, and Y low byte. Since the upper bits contain control information and the lower byte has coordinate information for pen moves, the plotter drivers take advantage of the ability to separately address the high and low bytes of the data register.

If the user required this same capability from a BASIC level program, the following program segments could be used. In both examples, the variables Hi and Lo contain the high byte and low byte data respectively. To output data in the bytes mode, we would use the following statements.

```
10      WAIT WRITE 3,6;SHIFT(Hi,-8)    ! When IOFLAG = 1 Store Hi
20      WRITE IO 3,4;Lo                ! THEN Store Lo
30      WRITE IO 3,7;0                 ! Trigger
```

Notice that the high byte data is shifted left 8 bits before being sent to the R6 OUT register. Also, lines 20 and 30 could be replaced by the statement WRITEBIN 3;Lo since this does the R4 OUT, R7 OUT sequence.

Input operations in the bytes mode is similar.

```
10      WAIT READ 3,4;Z          ! Demand new data item.
20      WRITE IO 3,7;0           ! Trigger input request.
30        WAIT READ 3,4;Lo       ! Get low byte.
40        WAIT READ 3,6;Hi       ! Get high byte.
50      Hi=SHIFT(Hi,8)
```

This program segment shows the entire sequence of events used to read data in the bytes mode. Lines 10-30 can in practice be replaced by a simple L = READBIN(3) function. After this operation is complete, the high-byte data may then be taken in using the R6 IN operation.

From these examples, we see that the 98032A was designed to allow the computer to separately address the upper and lower bytes of the input and output data latches. It should be noted, however, that the interface card is still exchanging 16-bit data with the peripheral device. From the fact that the 98032A has a byte mode of operation, it is often mistakenly inferred that a single 98032A can directly interface two 8-bit devices. Although this is possible, it does require that the user provide some external hardware to properly control the handshake operations.

As an example, let's assume that we wish to interface an 8-bit paper tape reader and an 8-bit paper tape punch to the desktop computer using a single 98032A card. Notice that in this case, we do not require the use of the bytes mode since one device is an output only device and the other is an input only device. If we merely connected the PCTL line to the control lines for each device and the ready/busy lines from each device to the PFLG line, these devices would not operate independently. For example each time we try to take a reading from the tape reader, the PCTL line would go set to demand the reading. But the punch would also see this signal and respond by taking whatever happened to be in the output latches and punching this information on the tape. In addition, if the punch completed its operation before the tape reader, it would indicate ready on the flag line. Depending on the levels used (positive or negative true logic) the interface could interpret this transition on the PFLG line to mean that the tape reader had finished its operation; and it would take a reading from the input data lines which might not yet be valid.

In order to prevent this, an external circuit similar to the one shown in Figure 38 could be used.

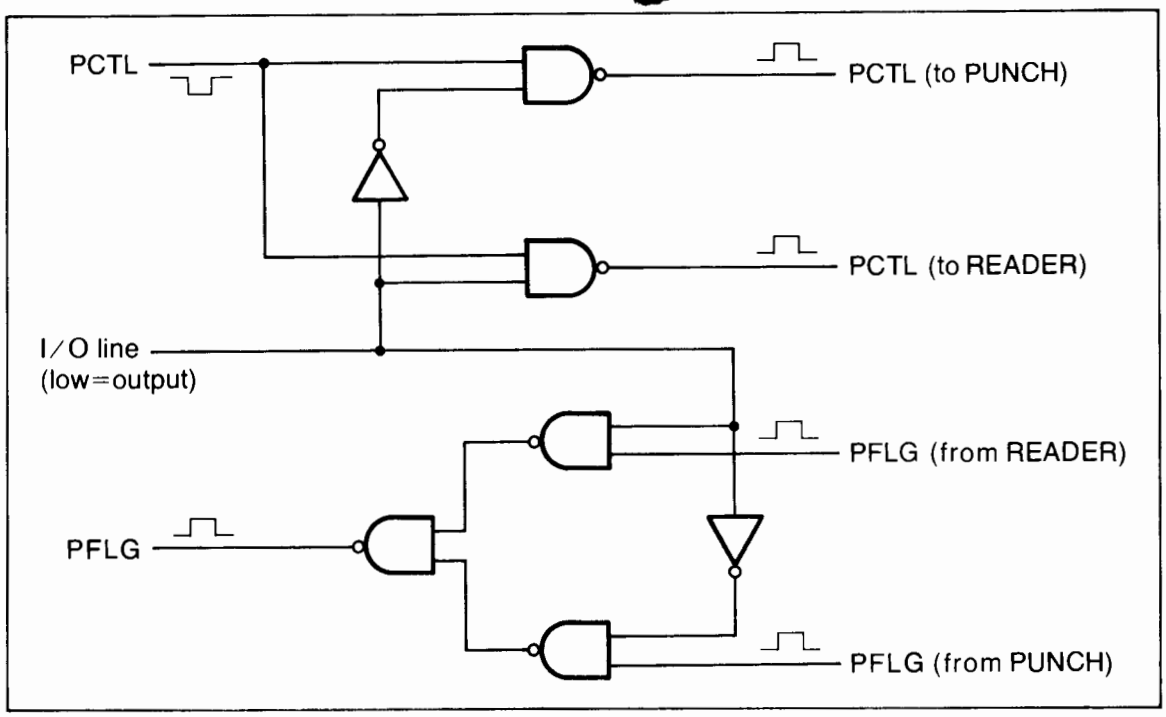


Figure 38

This uses the I/O line on the 98032A to gate the flag and control signals to only the device being addressed. For example, when the I/O line is low (output operation), only the punch sees the control pulse, while the PCTL line to the reader remains high throughout the entire operation. Also, no matter what transitions take place on the PFLG line from the reader, only the transitions generated by the punch are passed on to the PFLG line on the 98032A.

If the user wanted to use a similar circuit to allow two 8-bit input devices or two 8-bit output devices to be interfaced with one 98032A card, the I/O line shown in Figure 38 would instead be connected to the CTL0 line. The program could then select one of the two devices by executing either a WAIT WRITE 3,5;0 or a WAIT WRITE 3,5;1 (3 is the select code of the interface card) to set the CTL0 line high or low respectively to cause the PCTL and PFLG lines to be gated to one or the other of the two devices. Of course this application would require the use of the byte mode, with one device using the high byte data lines and the other using the low byte data lines.

Because of the external circuitry required and the added complexity needed in the program to select the device and shift data to and from the high byte, it is normally more convenient to simply use two 98032A cards to interface the two devices. Only in the case of extreme cost sensitivity or I/O slot limitations would such a scheme be practical.

Data Inversion and the Transfer Process

The 98032A Interface can accommodate either positive true or negative true logic on the input and output data lines. The I/O backplane of the desktop computer itself uses negative true logic and no modification is required for a peripheral device which also uses negative true logic. For a device which uses positive true logic, the sense of the data lines must be inverted before the data is interpreted by the program (for input) or sent to the device (for output). Because of the complexity of the 98032A card and the limited space available, the interface does not have room for the hardware to perform this inversion directly. It merely contains the provision for two jumpers (1 and 2) to indicate to the computer that the input and/or the output data must be inverted. The actual inversion is done by the drivers in the I/O ROM by complementing the data; that is, changing the ones to zeros and the zeros to ones before an output operation or after an input operation. These I/O drivers know that this operation is to be performed by checking bits 2 and 3 of the status byte (see Figure 32) from the 98032A card to see if either or both of the inversion jumpers are installed on the interface.

For the normal data operations and for transfers using interrupt buffers, this inversion process is handled automatically by the I/O ROM and the operation is totally transparent to the user. This is not the case for Fast-handshake (FHS) and DMA buffer transfers. In the case of the FHS transfer, this inversion is not done in order to obtain maximum I/O rates. In the case of DMA transfers, it is the hardware processor, not the I/O ROM, that handles the DMA transfer, and this processor only operates with negative true logic. Thus, in these special cases, any required data inversions must be done by the BASIC program itself. It should also be remembered that in the case of a FHS input buffer transfer where a terminating character can be specified, this character will also be inverted. Referring back to the NOFORMAT transfer type discussed under Advanced I/O Transfers, this complementing process can be done using a conversion table with a Variable-to-Variable transfer, and the actual data transfer done by a NOFORMAT transfer.

The 98033A BCD Interface

BCD Instruments

In the last section, we discussed using the 98032A Interface to connect peripheral devices whose outputs were in the form of binary data (up to 16 bits wide), or sequences of ASCII characters. There is a class of devices, however, known as BCD instrumens for which the 98032A is not a satisfactory interface for connecting them to the computer. To see why this is so, we need to look at some of the characteristics of these BCD devices.

Devices which fall into the BCD class are typically measuring instruments such as digital voltmeters and multimeters, scanners, frequency counters, gain-phase meters, digital panel meters, and so forth. These instruments usually display the readings they take to be read by an operator, typically showing from three to eight digits depending on the precision of the measuring device itself. They also indicate other information about that reading. For example, a digital multimeter (DMM) may also indicate the function being read (voltage, current, resistance), the range being measured (100 volts, 10 milliamps, 1 kilohms) and have an overload or out-of-range indicator. All of this information is useful to the operator taking down the readings.

In these instruments, each digit of the reading drives one digit in the display. The value of this digit is sent over four wires in a code called binary coded decimal or BCD. This format is also known as 8-4-2-1 code, since these are the values or weights given to each of the four lines. The encoding of the ten decimal digits is shown in Figure 39.

(8)	(4)	(2)	(1)	DIGIT
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9

Figure 39

These BCD lines are then sent to decoder circuits which convert them into seven-segment or dot-matrix patterns for driving the individual display digits.

With the advent of controlling computers, it became desirable to make it possible for the computer to directly read these measurements, collecting large numbers of readings for processing and analysis. At the time this was done, hardware circuits for converting these readings into a form the computer could understand were very costly. As a result, most designers merely made all of the data lines available to the computer directly with no attempt at conversion, and left it for the versatility of the computer program to sort out the meaning of the various lines. Output lines are also brought out to indicate the sign of the reading (plus or minus), a power-of-ten multiplier or exponent digit for accommodating the various ranges, an overload indicator, and a set of lines to indicate the mode of operation for multi-function instruments. Thus, an instrument which supplies six or eight digits of precision can have forty or more distinct lines on its output connector just to represent the reading, in addition to any control lines used.

If an interface such as the 98032A were used to connect such a BCD device to the computer, either two or three cards would have to be used in parallel, or a multiplexing scheme would have to be used. In addition, the computer would then have to sort out of all these bits, read the ones that represented digits, signs, exponent, function codes, etc. Instead, the 98033A BCD Interface was designed to accept all of these parallel bits, and translate them into a sequence of ASCII characters which represent the reading being taken. That is, the 98033A translates the data from a 43-bit parallel reading to a 16-byte, ASCII serial representation.

98033A BCD Formats

Figure 40 shows the input lines that are available on the 98033A Interface.

1 bit Mantissa sign	Sm
4 bit Mantissa digit 1	D1
4 bit Mantissa digit 2	D2
4 bit Mantissa digit 3	D3
4 bit Mantissa digit 4	D4
4 bit Mantissa digit 5	D5
4 bit Mantissa digit 6	D6
4 bit Mantissa digit 7	D7
4 bit Mantissa digit 8	D8
1 bit Exponent sign	Se
4 bit Exponent digit	De
1 bit Overload indicator	Ov
4 bit Function code	Fc

Figure 40

The number of lines available is usually more than sufficient to handle most BCD instruments. If a device is connected that has fewer than eight digits, the unused digits may be connected to the + 5 volt reference (for negative true logic) or to ground (for positive true logic) so that they will always be read as zeros.

When a reading is taken using the 98033A Interface, the card converts the data on the input lines into a sequence of 16 ASCII characters in the format shown in Figure 41.

Character:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ASCII:	±	X	X	X	X	X	X	X	X	E	±	X	,	O[1]	X	LF
Data Used:	Sm	D1	D2	D3	D4	D5	D6	D7	D8		Se	De		Ov	Fc	

Figure 41

The one-bit mantissa and exponent signs are converted to ASCII characters for plus or minus, and the 4-bit BCD digits are converted into the ASCII characters for the corresponding digits. Notice that the card itself provides the ASCII characters for the exponent notation (E), the comma to separate the reading from the overflow indicator and the function code, and a final line feed character (LF) to terminate the reading.

The characters indicated by an X in the ASCII representation of the format shown in Figure 41 normally correspond to digits connected from the instrument to the BCD input lines. If it is desired, however, they may be used for other purposes. Notice that in Figure 39, only ten of the sixteen possible binary patterns are used to represent the ten decimal digits. Figure 42 shows the ASCII characters that have been assigned to the other six binary patterns.

(8)	(4)	(2)	(1)	ASCII	
1	0	1	0	LF	line feed
1	0	1	1	+	plus sign
1	1	0	0	,	comma
1	1	0	1	−	minus
1	1	1	0	E	exponent
1	1	1	1	.	decimal point

Figure 42

1 The Overload character is normally an ASCII "0", but changes to an ASCII "8" when the overload bit is set.

Any character marked X in Figure 41 may be made to correspond to any of these ASCII characters by connecting its four BCD lines high or low to give the required pattern. For example, if we had a BCD instrument that has an implied decimal point to the right of the first digit, this would be indicated on the instrument's display panel, but this information is not part of the reading itself (unless the instrument adjusts the exponent to account for this). If the instrument uses negative true logic, we would connect all four lines of D2 to ground, giving it the binary pattern for a decimal point. Then we would connect digit 1 to D1, digit 2 to D3, digit 3 to D4, and so on. Now when the computer reads the ASCII sequence from the 98033A card, it will see a decimal point between the first and second digits of the reading. The large number of input lines provided, combined with the ability to redefine any digit to one of the ASCII characters in Figure 42 gives the 98033A a wide degree of flexibility for reading instruments with diverse formats.

The 98033A provides input lines for BCD instruments having up to eight digits in their readings, to accommodate high-precision instruments. Most BCD instruments will typically only have three or four digits. For added versatility, the 98033A provides an optional format (selected by a switch on the card) to allow two BCD instruments, or a single dual-output instrument, to be connected to the computer using only one interface card. In this format, the input lines are converted to the 16-character ASCII sequence shown in Figure 43.

Character:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ASCII:	±	X	X	X	X	,	±	X	X	X	X	X	E	Ov	X	LF
Data used:	Sm	D4	D2	D6	D8		Se	Fc	D1	D5	D3	D7		Ov	De	

Figure 43

In the optional format, pairs of readings are taken from two separate sources. This would be particularly useful in, for example, testing electrical circuits where voltage and current readings need to be taken simultaneously. If two separate interfaces were used, the time delay between the execution of the two ENTER statements in the computer program would make it difficult to take simultaneous readings.

The 98033A Interface Registers

Operationally, the 98033A BCD card is very similar to the 98032A Bit Parallel Interface. In fact, they are both type 2 cards (see the "Interface ID and Card Types" section) and the I/O drivers in the computer make no distinction between them. Figure 44 shows the register assignments for the 98033A Interface.

	IN	OUT
R4	DATA IN	(not used)
R5	STATUS	CONTROL
R6	(not used)	(not used)
R7	(not used)	TRIGGER

Figure 44

Since the BCD interface is for input only, it does not respond to any output operations. All data is input through the R4 IN register, using R7 OUT as a trigger in the same way as described for the 98032A operating in the words mode. Because all data from the BCD card is 8-bit ASCII, the upper eight bits of the 16-bit word received are always zeros.

Figure 45 shows the bit assignments in the R5 OUT control register, accessed by a WAIT WRITE to R5 or a CARD ENABLE statement.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INT	X	RESET	X	X	X	X	X

INT: Interrupt Enable on FLG = Ready
RESET: Reset Card to Its Power-on State

Figure 45

The reset and interrupt enable bits operate in an identical manner to those operations on the 98032A card. The other bits are not used and may be sent as ones or zeros.

Similarly, only the interface identification bits (4 and 5) and an interrupt-enabled indicator are significant in the R5 IN status byte (accessed by the WAIT READ and STATUS statements), as shown in Figure 46.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INT	X	1	0	X	X	X	X

INT: Interrupt enabled indicator
1,0: Interface Identification Bits (type 2)
X: Don't care

Figure 46

The remaining bits are not assigned meaning and will always return zeros. As with the 98032A Interface, when the STATUS statement is executed, the current state of the 1-bit status line (STS) is also included in the result returned as an artificial bit 8.

The 98033A Handshake Process

The handshake process for the 98033A BCD card is very similar to that described for the 98032A Bit Parallel Interface. It sets a control line to tell the BCD instrument to take a reading, and waits to see a response from the device on the peripheral flag line before converting the data on the input lines into the sequence of ASCII characters to send to the computer. That is, the computer will do 16 data byte demands from the interface before the card will set control to request another reading from the BCD device.

Figure 47 shows the normal sequence of events that takes place during this handshake operation.

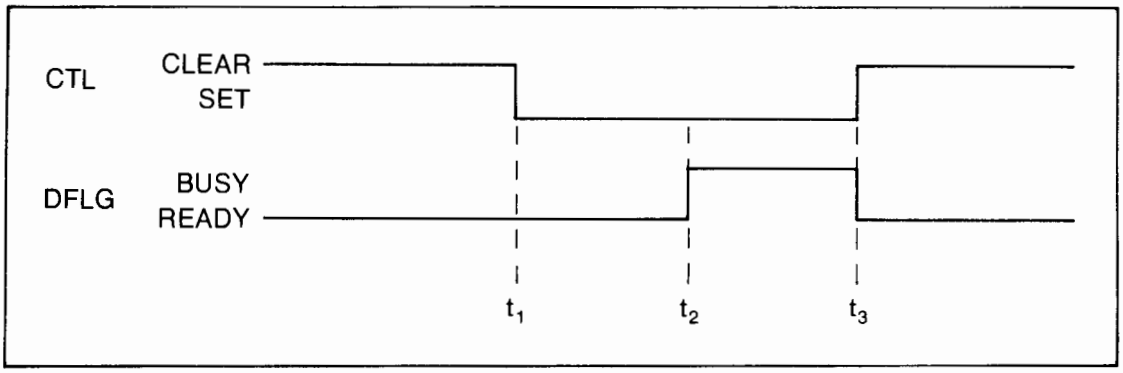


Figure 47

When the interface (in response to a request from the computer) requires the next reading, it will set the control line low to indicate to the BCD device that it should take another reading (t_1). At some later time (t_2), the device will indicate that it has seen this data request by setting its flag line (DFLG) busy, and proceed to take the reading. When the data on the input lines is valid, the device will then (t_3) indicate this fact by setting the DFLG line back to the ready state. This causes the interface to return its control line back to the clear state and begin translating the reading for sending to the computer. Since this process requires enough time for the computer to input 16 bytes of data, the BCD device must maintain the data on the input lines from the time it indicates ready (t_3) until the next time the control line goes set. That is, **the device can only change the data on the input lines during the set state of the CTL line**.

Some BCD devices are designed in such a way that they are armed for the next reading by the control line going set, but will not actually take the reading until it goes clear again. In order to accommodate these devices, the 98033A Interface provides an optional control mode (option 2) in which the CTL line will return to the clear state when the DFLG line goes from ready to busy (t_2 in Figure 47). The data is still read by the interface when the DFLG line returns to the ready state (t_3). A switch on the 98033A card allows the user to select the normal mode (option 1) or this special mode (option 2) of operation for the CTL line. In addition, both the CTL and DFLG lines can have their senses (high or low) inverted by other switches on the interface card to accommodate positive-true or negative-true logic levels.

In the section on 98033A BCD Formats, we discussed the optional data format which allows the 98033A to connect two BCD instruments using one interface card. As a result, two sets of control and flag lines are provided. CTLA and DFLGA are used to handshake with one BCD device, while CTLB and DFLGB are used for the other one. If only one device is being interfaced using the 98033A, CTLA and DFLGA are connected to this device in the normal manner (discussed below), and CTLB must be connected directly to DFLGB.

When two BCD devices are being used with one interface, both control lines (CTLA and CTLB) go set at the same time. Following this, CTLA returns to the clear state based on DFLGA alone, according to which option (CTLA-1 or CTLA-2) has been set in the configuration switches for channel A. Channel B operates in the option mode for which it has been set, independently from channel A. In any case, not until both channels have indicated ready on their respective DFLG lines will the interface begin to translate the reading and send the result to the computer. Readings are always taken from both channels simultaneously, and not until both devices have indicated ready. In this sense, the two BCD instruments are not treated as two independent devices. When only one BCD instrument is being interfaced, connecting CTLB to DFLGB makes channel B appear to be immediately ready, and the reading rate is determined by channel A alone.

Connecting BCD Devices to the 98033A

Finally, the question arises as to which lines on the BCD instrument should the control and the flag lines be connected. This question does not have a simple answer since BCD instruments made by different manufacturers (and often different instruments made by the same manufacturer) give various names to their control and flag lines. Most BCD devices made by Hewlett-Packard call the control line an "External Trigger," and the flag response line a "Print Command." Other common names for the control line are Trigger, External Encode, and Sample. The line to be connected to the flag line might be called Print, Print Enable, Ready, or Data Flag.

Often, the only way to tell which lines of the BCD instrument should be connected to the flag and control lines is to read the description of these lines in the operating manual for that instrument. For example, the following descriptions are taken from the reference manual for the HP 3480A/B Digital Voltmeter.

External Trigger — LOW for >50 microseconds initiates a measurement period. LOW state must be preceded by HIGH state for >50 microseconds.

Print Command — Goes HIGH at beginning of measurement period and LOW to indicate completion of measurement. HIGH to LOW transition constitutes a command to print.

Here, the external trigger line is identified with the control function by the key words "initiates a measurement." In addition, the fact that the low state of this line initiates the measurement indicates that this line has the same sense as the CTL line on the 98033A (low state is control set), and that the invert CTLA option should be left off. The statement that the print command line which responds in the way required by the DFLG line on the 98033A. Its logic sense is also correct without setting DFLG inversion.

The 98034A HP-IB Interface

An Introduction to the HP-IB

In Chapter 2 we said that the purpose of an interface is to provide mechanical, electrical, data, and timing compatibility between a peripheral device and the computer which controls that device. If a standard existed which specified all of these characteristics, then two devices which conformed to that standard would be "plug-to-plug" compatible. We would merely plug their connectors into one another and they would be ready to communicate. A major step in this direction was taken in 1978 when the Institute of Electrical and Electronics Engineers adopted

the IEEE-488-1978 standard which specifies many of these characteristics for a general purpose interfacing bus, sometimes called the GPIB. The HP Interface Bus, or HP-IB, is Hewlett-Packard's implementation of the IEEE-488 standard.

The major advantage of this standard is that it allows devices to be designed by various manufacturers which are immediately compatible with any other IEEE-488 device, requiring no interfacing operation on the part of the end user.

Data messages are sent from one device to another on the bus in an 8-bit parallel, byte serial manner. The standard does not specify how these data messages are to be encoded, although most devices that operate on the HP-IB use standard ASCII codes. In general, data messages most commonly consist of a sequence of ASCII characters, usually terminated by a line-feed character (LF). Thus, the only device-dependent information necessary for the user to know is the particular sequence of ASCII characters that cause the device to carry out each of the functions which it was designed to perform (see "Addressing the Bus Devices").

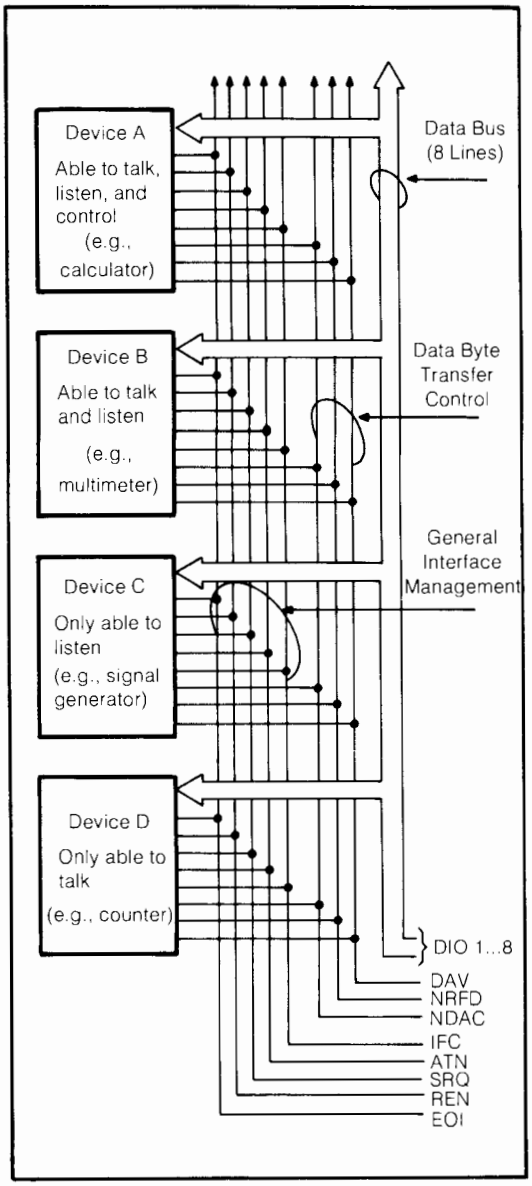
The HP-IB definition also allows several devices to be interconnected on the same bus. In the following sections, we will look at the structure of the HP-IB, the method of transferring data messages over the bus, several extended control features provided, and finally at some specific characteristics of the 98034A Interface. It is this interface which provides HP desktop computers with the necessary electronics to meet the specifications of the IEEE-488 standard and to be plug-to-plug compatible with all other such devices.

The Structure of the HP-IB

Physically, the bus itself is merely a set of sixteen wires (along with a few assorted ground wires and an electrical shield) to which all devices on that bus are connected (see Figure 48). Eight of these wires serve to carry the data messages back and forth over the bus. To maintain order, only one device at a time can place information on these data lines, and that device is known as the active talker. Any or all of the other devices on the bus may sense the information on these data lines and act on that information. Such a device is known as an active listener. By the nature of the actions which they perform, some devices may be only talkers (e.g., a paper tape reader) or only listeners (e.g., a printer). Other devices such as a digital voltmeter can be either a talker or a listener. That device is made a listener so that it can be programmed for the correct voltage range and told when to take a reading. It is then made a talker so that it can put the results of that reading on the data bus.

Thus there is a need for one device on the bus to set up talkers and listeners at the proper time, issue instructions to the other devices on the bus, and in general to make sure that all traffic on the bus proceeds in an orderly fashion. This device is called the active controller. Although any device can be designed with controller capability, usually it is a calculator or computer with its flexible capability that is assigned this task.

Finally, there is one and only one special device on the bus known as the system controller. This capability is established by the hardware of the device itself (usually by the setting of a slide switch or a jumper) so that when power is turned on or the bus is reset, the device set to be the system controller will also assume the role of the active controller. At any time, the current active controller may pass control to any other device on the bus that is capable of performing the functions of a controller. (All devices are not required to have this capability.) The role of system controller, however, stays with the device which is physically set for that function and cannot be passed off. At any time when the system controller determines that something has gone wrong with the normal bus operations, it can reset the bus and get back active control.



HP-IB Signal Lines

Figure 48

Figure 48 shows the meanings given to the other eight 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:

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

Using these lines, a typical data exchange would proceed as follows. All devices currently designated as active listeners would indicate (via the NRFD line) when they are ready for data. A device not ready would pull this line low (ground), while a device that is ready would let the line float high. Since a low overrides a passive high (see Chapter 1), this line will stay low until all active listeners are ready for data. When the talker senses this, it places the next data byte on the data lines and then pulls DAV low. This tells the listeners that the information on the data lines is valid and that they may read it. Each listener (at its own speed) then takes the data and lets the NDAC line go high. Again, only when all listeners have let NDAC go high will the talker sense that all listeners have read the data. It can then remove DAV (let it go high) and start the entire sequence over again for the next byte of data. A more detailed description of the handshake process is given in several of the HP-IB references (see Bibliography). It is not necessary for the user to understand the details of the handshake in order to operate the HP-IB.

The five remaining lines are called control or bus management lines. Their meanings are:

ATN — Attention
IFC — Interface Clear
REN — Remote Enable
EOI — End or Identify
SRQ — Service Request

Each of these lines will be discussed in one or more of the following sections.

Before leaving this overview of the HP-IB and discussing the operation of the bus, some of the limitations of the HP-IB should be considered. The first limitation is that a maximum of 15 devices may be connected together by one HP-IB. This limitation arises from electrical specifications for the line driver and receiver circuits, and how much current they can provide or sink. Another limitation is that the total cable length connecting all of the instruments on one bus cannot exceed 20 meters in length. Voltage levels on the various lines do not change instantaneously, but require a certain amount of time proportional to the length of the cable. A limit is placed on the cable length to insure that the bus will operate properly at its rated maximum speed. In general, then, the HP-IB is intended to provide a simple means of interconnecting local instrumentation clusters. Other means of interfacing (such as serial I/O to be discussed later) are better suited to long distance communications.

Addressing the Bus Devices

The primary use of the HP-IB is for the transfer of data messages from one device to another on the bus. While the HP-IB does provide a wide variety of extended control features (such as serial and parallel polling, service requesting, etc. which are discussed in the next section), many instruments can be fully operated through simple data transfers alone. For example, sending the ASCII character string "F2R3" to the HP 3490A Digital Multimeter would cause it to be programmed into function 2 (AC Volts) and range 3 (100 Volts). Another simple ASCII message, "M3E", would tell it to go into mode 3 (single sample with output) and to execute a reading. The result of this reading would also be sent back to the listening device as a stream of ASCII characters representing the value read. Thus, a great many HP-IB devices can be programmed and operated by merely knowing how to send and receive data messages on the bus, and the list of messages that a particular device on the bus can respond to. Since these "command" messages are not specified by the IEEE-488 standard, the operating manual for each device should be consulted to find the list of commands to which it will respond. How then are messages sent and received over the HP-IB using the System 35 or 45? In order to isolate the user from the required bus protocol (i.e., setting up the talker and listener, sequencing the handshake lines, etc.) the I/O ROM and the interface take care of these tasks, leaving to the user only the requirement of specifying what the data message should be and which device on the bus is to receive it. For this purpose, the same OUTPUT statement used to send data to a printer or other output device can be used. If we wished to send a message to a printer on select code 6, we would simply execute the statement OUTPUT 6; "Hello". The process is slightly complicated, however, by the fact that each HP-IB can have several devices attached to it. If a particular HP-IB interface were set to select code 7, execution of the statement OUTPUT 7; "F2R3" to program the 3490A Multimeter would be ambiguous, since the I/O ROM would not know which device on the bus should receive the message. Thus, to completely specify a destination for such a message, it is necessary to give not only the select code of the HP-IB interface, but also some way of indicating one of the many devices on that bus. For this purpose, each device is assigned an address or a device number. This device number can be in the range 0 to 30 and each device on the bus must have a different address in this range. A unique device on the bus may now be specified by giving both its select code and device number. For example, if the 3490A in the previous example were set to device number 23, the statement OUTPUT 723; "F2R3" would specify that the data message "F2R3" should be sent to device number 23 on the HP-IB set to select code 7. Since the normal select code range is [0,16], the I/O ROM would interpret this three-digit select code as specifying a device on the HP-IB, and automatically use the proper HP-IB protocol. This protocol would consist of setting up the computer as the talker, the instrument set to device number 23 as the listener, and then sending the data message.

Both the addressing information and the data message are sent over the same set of eight data lines. In order to distinguish one from the other, one of the bus control lines called the attention (ATN) line is used. When this ATN line is false, the 8-bit pattern on the lines is interpreted as a character (usually ASCII) in the data message. When the ATN line is true, the pattern on the data lines in interpreted as control or addressing information. In this mode, only seven of the eight data lines are used. Depending on the setting of bits 5 and 6, the character sent in the ATN true mode will fall into one of four classes, shown in Figure 49.

Bit#:	7	6	5	4	3	2	1	0
Bus Command	X	0	0	C	C	C	C	C
Listen Address	X	0	1	L	L	L	L	L
Talk Address:	X	1	0	T	T	T	T	T
Secondary Address:	X	1	1	S	S	S	S	S

(X = "don't care," 1 or 0)

Figure 49

If the class bits (5 and 6) are both zeros, the remaining five bits (4-0) are used to encode various bus commands which are discussed in a later section. When they are 01, the following five bits specify one of the 31 possible listen addresses; and when they are 10, bits 4-0 specify one of the 31 possible talk addresses. These addresses are in the range [0,30]. Address 31 (bits 4-0 all set to ones) is not a legal device address, but is interpreted as an unlisten (0111111) or an untalk (1011111) command to cancel any currently addressed talker or listeners.

Returning to our previous example, execution of the statement OUTPUT 723; "F2R3" would cause the sequence of message bytes shown in Figure 50 to be sent over the bus.

ATN	Data Lines	ASCII	Meaning
T	01010101	U	Computer (device 21) is a talker
T	00111111	?	Unlisten any previous listeners
T	00110111	7	Device 23 is a listener
F	01000110	F	First data byte
F	00110010	2	Second data byte
F	01010010	R	Third data byte
F	00110011	3	Fourth data byte
F	00001101	CR	Carriage return
F	00001010	LF	Line feed

Figure 50

Notice that the computer (which is also the controller in this case, since it is doing the bus addressing) has a device number, 21, just like any other device on the bus. With the ATN line true, it sends out its own talk address, an unlisten to unaddress any listeners from previous operations, and sets up device 23 as the listener for the data message that will follow. Notice that the controller did not have to send an "untalk" command. Since there can be only one talker addressed at a time, the bus standard requires that a device addressed to talk must become unaddressed as a talker as soon as any other device is designated as a talker. Also, the bytes on the data lines appear as normal ASCII characters. They are given their special addressing interpretations shown in Figure 49 only because the ATN line is true while they are being sent. Once the addressing is complete, the controller sets the ATN line false (data mode) and begins to output the ASCII data message. The listening device (3490A) receives this message and decodes it to set the specified function and range. Notice that while all characters sent in the ATN true mode have meanings specified by the bus standard, those sent in the ATN false (data) mode are defined by the device itself as to what action they will cause. In this case, the 3490A has been designed to interpret these data bytes as programming information for setting its function and range. Finally, most HP-IB devices send and recognize CR/LF (or sometimes just LF) as marking the end of a data message. Some devices, however, may choose other end-of-message delimiters and the user should consult the individual operating manuals for these devices.

When it is time for the 3490A to deliver the voltage reading it has taken, the sequence shown in Figure 51 is generated.

ATN	Data Lines	ASCII	Meaning
T	00111111	?	Unlisten
T	00110101	5	Computer is a listener
T	01010111	W	Device 23 is a talker
F	(ASCII characters for voltage reading)		
F	00001101	CR	Carriage Return
F	00001010	LF	Line Feed

Figure 51

To take the reading, the computer (controller) sends out the unlisten, listen address 21, and talk address 23 in the ATN true mode. It then sets ATN false (data mode) and then waits for the talker (3490A) to place the data bytes on the data lines. Notice that even though the 3490A is the talker in this case, it is the computer acting as the controller which sets up the talker and listener and then gives the 3490A "permission to start talking" by setting the ATN line false.

The controller is always responsible for determining the sequence of events on the bus! From the computer, this input operation would have been initiated by the execution of the statement ENTER 723; A; this would specify that a numeric reading should be taken from device 23 on the HP-IB set to select code 7, and the result stored in the program variable A.

There is a common misconception when using the HP-IB that a device on the bus has a talk address which is different from its listen address. For example, when addressing the 3490A in this example, an ASCII "7" was used for the listen address, and an ASCII "W" for the talk address. In looking at the 5-bit pattern (10111 = decimal 23) that forms its actual device number, it is the same for both. It is merely the difference in the talk (10) or listen (01) bits that gives rise to a different ASCII representation for each. The fact that the device has only one address is more evident in the high-level specification for its address, 723, used in both the ENTER and OUTPUT statements.

From Figure 49 we see that the class bits (5 and 6) can also be both ones. In this case, the remaining five bits (4-0) are interpreted as a secondary command or extended address. The device receiving this secondary command is the one whose primary (talk/listen) address immediately preceded it, and the device is free to choose how it will interpret this additional addressing information. This information will be found in the individual operating manuals for those HP-IB devices which use secondary addresses. To send a secondary address (assuming that the System 35 is the active controller) the user simply appends a decimal point and two more digits in the range 0 to 31 to the normal select code and device number. For example, the statement OUTPUT 723.05;<data> would cause the I/O ROM to issue a listen address of 23 (00110111) followed by a secondary address of 5 (01100101) to the HP-IB on select code 7 during the addressing portion of the output sequence.

Data Operations on the HP-IB

In the last section, we discussed the use of normal read and write statements to send and receive data messages over the HP-IB. If the instrument that is being addressed is a slow one, and the program can do other useful work while the data exchange is taking place, the interrupt transfer methods discussed in Chapter 2 can also be used with an HP-IB device. For example, assume that the System 35 is connected to a digital voltmeter (DVM) on the HP-IB with a device number of 5, and that each reading consists of a string of 16 ASCII characters. The following program segment

```
1          OVERLAP
10       DIM Noformat$[1600]
20       ON INT #7,3 GOSUB Read
30       ENTER 705 BINT NOFORMAT;Noformat$
         .
         .
100 Read:        !  Process the readings.
```

would set up a string buffer called Noformat$ large enough to hold 100 readings. The input statement in line 30 would automatically start reading data bytes into the buffer until it is filled. Setting up device 5 as the talker, the computer as the listener, and servicing the interrupts as each byte comes ready are all handled by the I/O ROM while the remainder of the program continues executing. When the string buffer has filled, a branch to the user's service routine labeled "Read" will take place, where the program can read the data out of the buffer and process it. It should be carefully noted, however, that during this Noformat transfer the main program should not attempt to do any I/O operations to the HP-IB on select code 7. Even though the System 35 is capable of doing other programming tasks while the buffer transfer is taking place, the HP-IB itself can only handle one data exchange at a time. For example, if during the data transfer the main program were to execute an OUTPUT 723;... statement, the computer would be addressed as a talker, thereby unaddressing the DVM (device 5) as a talker, and the 100 readings would never be completed.

In all of the previous examples we have assumed that the System 35 was the active controller on the HP-IB, and treated devices on that bus like any other peripheral device by merely appending the device number to the select code. We also assumed that the devices we were addressing as the controller would properly respond by taking the data we sent when we addressed them as listeners, and would not place their own data on the bus until we addressed them to talk. In short, we assumed that as the controller we were running the show!

The System 35 and 45 computers are also capable of acting as a non-controller; that is, acting just like any other talker/listener on the bus. Two new questions arise when the computer is connected to the HP-IB in this non-controller mode. How does the computer know when it should talk and listen? And how does it read from and write to the bus without the automatic setting of talkers and listeners which would be illegal when it is not the controller?

Two solutions are provided to the first problem. The first way is to check the status byte from the 98034A interface itself, obtained as the result of the status statement; e.g., STATUS 7;A. Figure 52 shows the meanings assigned to the various bits in this status byte.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SRQ	ACT	TLK	LST	SAC	1	SPL	EOR

Figure 52

Bit 5 is set (= 1) when the computer is addressed as a talker, and bit 4 is set when it is addressed as a listener. (The meanings of the other bits will be discussed in a later section.) Thus the program can periodically read the status byte and test the appropriate bits to see if it has been addressed to talk or listen.

A more convenient method makes use of the interrupt capability so that the program does not have to periodically sample and test the status byte. While the other interface cards have only one interrupting condition (flag line indicating ready) the HP-IB interface can be set to interrupt on any combination of seven conditions specified in an interrupt-enable mask (see Figure 56 page 133). In this mask, bits 4 and 5 being set to one enable an interrupt on the conditions addressed-to-listen and addressed-to-talk respectively. Thus, the program can enable the interface to interrupt and have the I/O ROM branch to a user-written service routine whenever the computer is addressed as a talker or listener. As an example, assume that the System 35 is on an HP-IB as a non-controller, and is also interfaced to a DVM using a 98033A BCD Interface (Figure 53).

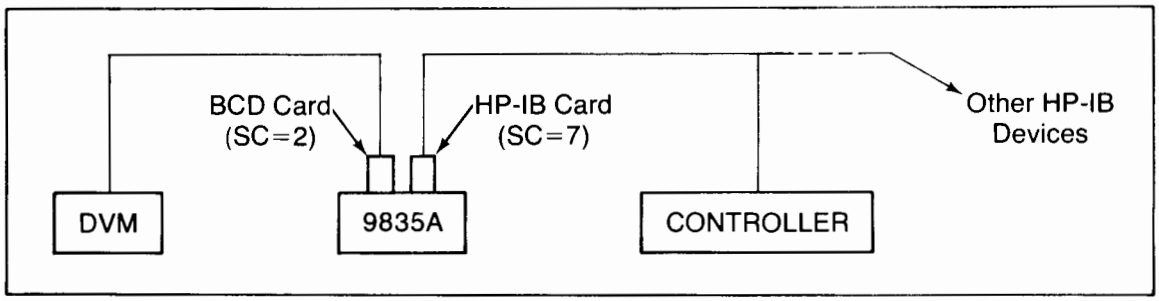


Figure 53

Normally the System 35 is running a local computation program (background job). But when the controller asks for a reading (i.e., makes the System 35 a talker) the System 35 is to take a reading from the DVM and place the result on the HP-IB. Also, the DVM is operating as a two-channel device (see "98033A BCD Formats") and the controller can tell the System 35 whether it wants a reading from channel 1 or channel 2 by addressing it as a listener and sending it the ASCII character "1" or "2". The following program in the System 35 would accomplish this task.

```
5                PASS CONTROL 720
10       ON INT #7,3 GOSUB Service
20       CONTROL MASK 7;48       ! Enable for TALK, LISTEN Interrupts.
30       CARD ENABLE 7
         .
         .
         .       BACKGROUND JOB EXECUTES HERE.
         .
440 Service:        STATUS 7;A
450      IF BIT(A,5)=1 THEN Talk
451      !
460 Listen:         ENTER 7;Read_var
470                 CONTROL MASK 7;32      ! Enable for TALK Inter.
480                 GOTO Exit
481      !
490 Talk:           ENTER 2;A,B            ! Take readings.
500      IF (Read_var<1) OR (Read_var>2) THEN Error
510      ON Read_var GOTO Write_a,Write_b
511      !
520 Error:          OUTPUT 7;"Error",Read_var
530      CONTROL MASK 7;16                  ! Enable for LISTEN Inter.
540      GOTO Exit
541      !
550 Write_a:        OUTPUT 7;A
560      CONTROL MASK 7;16                  ! Enable for LISTEN Inter.
570      GOTO Exit
571      !
580 Write_b:        OUTPUT 7;B
590      CONTROL MASK 7;16                  ! Enable for LISTEN Inter.
591      !
600 Exit:     CARD ENABLE 7
610      RETURN
620      END
```

Line 10 specifies that if an interrupt occurs on select code 7, the program should branch to the routine labeled "Service." Line 30 then enables the HP-IB interface to interrupt on either being addressed to talk or listen. The interrupt enable mask, decimal 48, corresponds to a binary pattern of 00110000 (i.e., 16+32) which sets bits 4 and 5. The main program in lines 40 through 430 then proceeds with its execution.

When the System 35 is addressed as either a talker or a listener by the controller, the program branches to the service routine at line 140. Since the interrupt enable mask specified either of two conditions (talker or listener), line 450 then tests the status byte to determine which condition caused the branch to the service routine. If the System 35 was addressed as a listener, the program merely reads and saves in Read_var the new channel number, and then returns to continue with the background job. If it was addressed as a talker, it goes out to the DVM on select code 2 and takes readings from channels 1 and 2 into the variables A and B. Then depending on the value of Read_var, one of the two readings is output to the bus.

The other important point to notice in this program is that all input from and output to the bus used only the interface select code. This is the standard procedure when the computer is not the controller on that bus. Since no device number is specified, the I/O ROM merely inputs or outputs data on the bus without attempting to do any automatic addressing.

Extended HP-IB Control Features

In the previous sections, we have discussed exchanging data messages on the HP-IB. To this extent, devices on the HP-IB only differ from devices connected to the computer by other interfaces in that more than one device may be connected to the computer using a single 98034A Interface card. The real power of the HP-IB comes from its implementation of extended control features. If a measuring instrument is connected to the computer using for example, the 98032A General-Purpose 16-Bit Interface, any remote control of that instrument's extended functions (such as resetting it to its power-on state, disabling its front panel controls, or detecting when it requires service) is most probably done by setting external control lines high or low. Each instrument's capabilities and method of controlling these functions will be different, and a good deal of skill and knowledge of interfacing is required to properly control these functions using the lines available on the interface chosen. With the HP-IB, on the other hand, many of these functions have been standardized and all instruments that provide for these extended control features have them accessed by the controller in the same way. It is the nature and use of these extended control features that make up the topic of this section.

In general, the types of operations that can be remotely controlled or programmed for a device on the HP-IB fall into two categories: those that are specific to that device, and those that are general to all devices. For example, the setting of the type of measurement to be taken (e.g., voltage, current, resistance, etc.) and the range (100 volts, 10K ohms, etc.) make sense for a digital multimeter on the bus, but have no meaning for a printer or a frequency counter. Thus, function and range setting would be an example of a device-dependent control operation. To make this type of control as general as possible, data messages are used and each device is free to interpret these data messages as it chooses. We saw in a previous example how a 3490A interpreted the data message "F2R3" as a command to switch to the 100 Volts AC range. Each device on the bus has some set of operations that can be programmed through these data messages, a list of which is found in the operating manuals for that specific device.

In this section, we will look at the other category of device control messages which are common to all devices on the bus. For example, if we wish to reset a device on the bus, the IEEE-488 standard defines a message called device-clear which is recognized by all devices on the bus. It should be noted that the standard does not require all devices to implement the device-clear operation; but for those that do implement it, it is always accessed in the same way using the device-clear message. Also, using the same example, the standard does not define what exactly is to be cleared or reset. This is left up to the individual device. Some devices on receiving this message may reset everything to the power-on state, while others may only clear selected conditions. In any case, the controller does not require any device-dependent information in order to issue the device-clear message. The remainder of this section will discuss these device-independent messages that can be sent, and the general types of action that will take place if the device implements a response to that message.

When we refer to these as device-independent messages, we simply mean that all devices on the bus will recognize that a particular message (for example, device-clear) is being sent, regardless of how it chooses to respond. These command messages are encoded on the data lines as 7-bit ASCII characters, and are distinguished from normal data characters by the setting of the attention (ATN) line. That is, when the ATN line is false, bytes on the data lines are interpreted as simple data characters. But when the ATN line is true, the data lines become the carriers of command information. The set of 128 ASCII characters that can be placed on the data lines during this ATN-true mode are divided into four classes as shown in Figure 49 and Appendix A. We have already seen how three of these classes are used to generate talk addresses, listen addresses, and secondary addresses. The fourth class, bus commands, is the one used to encode these device-independent control messages.

In addition to data and command messages, there are five other bus messages that, because of their importance and timing considerations, have hardware lines dedicated to them. These are shown in Figure 48.

We have already seen how the attention line (ATN) is used to distinguish between simple data and command information of the eight data lines. The meanings of the four remaining lines are explained next.

Interface Clear
(IFC): Only the hardwired system controller can issue the IFC message. By pulling the IFC line low, all bus activity is unconditionally terminated, the system controller regains (if it has been passed to another device) the status of active controller, and any current talker and listeners become unaddressed. Normally, this message is only used to abort an unwanted operation, or to allow the system controller to regain control of a bus where something has gone wrong. It overrides any other activity that is currently taking place on the bus.

Remote Enable
(REN): This line is used to allow instruments on the bus to be programmed remotely by another device on the bus, usually (but not necessarily) the active controller. Its use is discussed in more detail later in this section.

End or Identify
(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 (LF = decimal 10). A device (e.g., a disk) may wish to send blocks of information in 8-bit bytes which represent general binary patterns; and no specific 8-bit pattern can be designated as a terminating character since it could occur anywhere in the data stream. In this case, the EOI line is used to mark the end of the data message. When the listeners detect that the EOI line is set, they recognize that the byte on the data lines is the last one of the data message.

The EOI line is also used during an identity (parallel poll) sequence to be discussed later.

Service Request
(SRQ: The active controller is always in charge of the order of events on the HP-IB. If a device on the bus has some information of which the controller should be aware, it can use the service request line to ask for the controller's attention. For example, a printer might request service to inform the controller that it is out of paper. Or a digitizer might assert service request to tell the controller that its sample button was pressed by the operator and a reading is ready to be taken. This represents a request (NOT a demand), and it is up to the controller 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 each individual device and will be contained in the operating manual for that device.

Figure 54 shows the device-independent control messages that can be sent, and the statements used by the I/O ROM to generate these messages. The two columns in Figure 54 show the results of these statements when they are sent to the entire bus (select code only specified) or to a particular device on the bus (select code and device number specified.)

Extended Bus Control Messages

BASIC Statement	<SC> only	<SC> <DN>
CLEAR	DCL	SDC
ABORTIO	IFC	(error)
TRIGGER	GET	<L> + GET
REMOTE	REN on	REN + <L>
LOCAL	REN off	<L> + GTL
LOCAL LOCKOUT	LLO	(error)
STATUS	98034A status byte	serial poll
PPOLL	parallel poll	(error)
PPOLL CONFIGURE	(error)	<L> + PPC + PPE
PPOLL UNCONFIGURE	PPU	<L> + PPC + PPD
REQUEST	SRQ	(error)
PASS CONTROL	(error)	<T> + TCT

<L> = specified device addressed as a listener
<T> = specified device addressed as a talker
<SC> = interface select code only
<SC><DN> = select code and device address

Figure 54

When the CLEAR statement is executed, all devices on the bus execute their clear operation in response to the **device clear** (DCL) message. If a device number is specified (e.g., CLEAR 711), then that device is addressed as a listener and it alone responds to the **selective device clear** (SDC) message. Each device on the bus may choose how it will respond to the selective (SDC) or universal (DCL) clear instruction. If the computer is set to be the system controller, it may also execute the clear interface (ABORT IO) statement which causes the IFC line to be pulsed low issuing the **interface clear** message discussed above.

In some applications it is desirable to have two or more instruments start their operations at the same time. For example, we might like to apply a step voltage function to a circuit under test and measure the transient response at some node in that circuit. A signal generator would be programmed to apply the voltage step and a digital voltmeter would be programmed to take the voltage measurements. In order to start both instruments off at the same time, the TRIGGER statement would be executed which would issue a **group-execute-trigger** (GET) message over the bus.

Many bus instruments such as digital voltmeters can have their various functions and ranges selected either locally by manually setting their front panel controls, or remotely by programming messages from a controller. In order to program such an instrument via the bus, the **remote enable** (REN) line must be set. When the REN line is set, addressing the device as a listener makes it capable of receiving programming instructions from the bus. When the 98034A Interface powers up (or following the IFC message) the REN line is automatically set. It may be set by using the REMOTE statement or cleared by using the local (LOCAL) statement. If the LOCAL statement is executed specifying both a select code and a device number (e.g., LOCAL 715), the REN line is not cleared, but the specified device is addressed as a listener and that one device receives a **go-to-local** (GTL) message. The instrument responds to the GTL message by switching control from the bus to the front panel manual controls, allowing an operator to set its programming controls. In many instruments, the operator can switch from remote operation to local operation by pressing a return-to-local button on the instrument. If the program controlling the instrument wishes to prevent manual operation by an unauthorized operator, it can execute the LOCAL LOCKOUT statement. This issues a **local lockout** (LLO) message on the bus that makes the return-to-local buttons on the bus instruments inoperative. In this state, the only way to transfer control to manual operation on a particular instrument is through the GTL bus message. Using combinations of these remote/local messages, a controller can set up any combination it chooses of instruments operating either remotely or locally as determined by the particular application.

We have already seen how the STATUS statement is used to obtain a status byte from the 98034A Interface. This status byte contains information such as the current addressing state (talker, listener, controller, etc.) of the card. Each instrument on the bus can also have a status byte which contains useful information about that device itself. The meaning of the information in this status byte is determined by each device and found in the operating manual for that device. In order to obtain this device status byte, the same STATUS statement used to get the status byte of the interface is executed, but a select code and the device number is given. For example, the statement STATUS 713;A would return the status byte from device 13 on the HP-IB set to select code 7. On the HP-IB, reading this status byte is referred to as a **serial poll** operation. The device is addressed as a talker, a special control message called serial poll enable (SPE) is issued, and the bus is placed in the data (ATN false) mode. Because of the SPE message, the device addressed as a talker knows not to put normal data on the lines, but rather its serial poll (status) byte. The controller reads this byte, and then issues a serial poll disable (SPD) message to cancel the SPE message. All of the bits in this status byte are defined by the device itself to encode any information it chooses, with the exception of bit 6. If this bit is set, it identifies that device as being one which is currently asserting a service request. Thus, when the controller recognizes that some device on the bus is requesting service (by the SRQ line being set) it can serially poll each device to find out which one (there may be more than one) requires service.

When several devices on the bus are capable of requesting service, the controller does not have to poll each device serially to determine which one it is. Another operation called a **parallel poll** is capable of polling up to eight devices at one time. Each device is assigned one of the eight data lines on which to respond when a parallel poll is conducted. When the controller senses SRQ, it conducts a parallel poll by setting both ATN and EOI true at the same time. (Note: In the data mode the EOI line has the meaning of "end-of-message." In the ATN true mode, it has the meaning of "identify" in the sense of a parallel poll.) All devices currently requesting service will then respond on their assigned data line. By checking the bits in this poll byte, the controller can immediately determine which devices require service without serially polling each one. If the device requesting service has more than one possible reason for asserting SRQ, the controller may also conduct a serial poll on that one device; and its status byte could contain more detailed information about why SRQ was asserted.

Most devices that are designed to respond to the parallel poll operation determine which data bit to respond on and what logic sense (high or low) to use by switches or jumpers set on the instrument itself. Some devices, however, allow the controller to program them for this information. This is done using the parallel poll configure statement PPOLL CONFIGURE which addresses the device as a listener, sends the **parallel poll configure** (PPC) bus message, followed by a **parallel poll enable** (PPE) byte as specified in the PPOLL CONFIGURE statement. The bits of this byte are 0110SPPP, where PPP is the binary equivalent of the data line on which the device should respond (0 through 7) and S is a 1 or a 0 response on that line. A device that has been programmed for its parallel poll response may be disabled for parallel poll response by executing the parallel poll unconfigue PPOLL UNCONFIGURE statement. This sends the **parallel poll disable** (PPD = 01110000) message to that device which cancels any previous PPE message. If the PPOLL UNCONFIGURE statement is executed using a select code only, with no device number, a **universal parallel poll unconfigure** (PPU) message is sent. This deactivates all devices on the bus whose parallel poll response can be remotely programmed.

If the System 35 is not the active controller on the bus, it too may wish to set a **service request** (SRQ) to get the controller's attention. This is done using the request service (REQUEST) statement. This statement has two parameters which specify the select code of the HP-IB, and the serial poll response byte, with bit 6 determining whether the SRQ line should be set. For example, executing the statement REQUEST 7;37 would set a decimal 37 (binary 00100101) as the serial poll response byte. This byte is stored on the 98034A Interface to be delivered anytime the controller conducts a serial poll. The statement REQUEST 7;67 would set a decimal 67 (binary 01000011) as the serial poll response byte, and set the SRQ line, since bit 6 is set.

Finally, executing the pass control statement (PASS CONTROL) specifying a select code and a device number will cause that device to be addressed as a talker and the **take control** (TCT) message to be sent. This will result in passing active control from the computer to the specified device, which will then have responsibility for sequencing bus activity. By addressing the device as a talker (which automatically unaddresses any previous talker), this protocol guarantees that only one device will respond to the take control message and that there will be only one active controller on the bus at any time.

Using the 98034A Interface

Most of the HP-IB operations discussed in the preceding sections are implemented automatically by the I/O ROM and by a microprocessor contained on the 98034A Interface card itself. Since these operations are well defined by the IEEE-488-1978 standard, and have been made transparent by the high-level programming language, it is less important that a user of the HP-IB understand the detailed workings of the interface card.

There are, however, a few operational characteristics of the 98034A which the user should understand in order to properly program the interface for such activities as interrupt operation, acting as a non-controller, using the EOI capability, and so on. These characteristics will be discussed in this section.

NOTE

The HP 98034A Interface has been revised to alleviate certain irregularities which can become apparent when the 98034A Interface is used with System 35/45 desktop computers. To avoid problems, use the HP 98034A Revised Interface for HP-IB operations. This is only of concern to users who purchased a System 35 or 45 for use with an older version of the 98034A Interface.

Because of the increased complexity of the 98034A Interface, four status bytes are required to contain all of the information about the card which might be of interest to the computer controlling that interface. Figure 55 shows the meanings assigned to the various bits in these four status bytes. The information which is most often used is collected in the fourth[1] of these status bytes, and is the one returned as the result of executing the STATUS statement. The other three status bytes contain less-frequently used information, and can be obtained from the status operation by specifying additional return variables (see System 35 or 45 I/O ROM Programming Manual).

First Status Byte:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	DCL	0	ERROR

Bit 0: Is 1 when error detected.
Bit 2: Is 1 when Device Clear received.

Figure 55a

Second Status Byte:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	1	0	A_5	A_4	A_3	A_2	A_1

Bits 0-4: HP-IB address

Figure 55b

1 Note that the order in which these status bytes is returned by the status statement is different than their actual order on the 98034A interface. Refer to the I/O ROM programming manual for further details.

Third Status Byte:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
EOI	REN	SRQ	ATN	IFC	NDAC	NRFD	DAV

Logical 1 indicates corresponding signal line is true.

Figure 55c

In the first status byte, the error bit (bit 0) is set whenever an illegal operation on the bus is attempted. This would include attempting to talk or listen when the card has not been addressed to do so, or attempting to specify bus addressing information when the 98034A is not the active controller on the bus. Normally, these operations are handled automatically by the I/O ROM and the user need not be concerned with this error indicator.

If the 98034A is not the controller on the bus, and the controller sends a device clear message, bit 2 of the first status byte will be set to indicate that this condition occurred. Both the error and the device clear bits will remain set until the status is read, at which time they will automatically clear to be ready for the next occurrence of these conditions.

The second status byte contains the bus address (in the range 0 to 30) that has been set on the 98034A card, in bits 4 through 0. This information is normally used by the I/O ROM when it needs to issue its own talk or listen address as part of the automatic addressing sequence associated with ENTER and OUTPUT statements. It is available to the user, however, if he should wish to check the address that the interface card has been set to.

The third status byte simply contains a direct mapping of the five bus control lines and the three handshake lines (Figure 48). Again, this information is required by the automatic bus drivers in the I/O ROM and does not normally represent information that is directly useful to the user's program.

Fourth Status Byte:

Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STS	SRQ	ACT	TLK	LST	SAC	1	SPL	EOR

Bit 8: Peripheral Status
Bit 7: Is 1 when the SRQ signal line is true.
Bit 6: Is 1 when the calculator is the active controller.
Bit 5: Is 1 when the calculator is the active talker.
Bit 4: Is 1 when the calculator is an active listener.
Bit 3: Is 1 when the calculator is the active system controller.
Bit 2: Is always 1.
Bit 1: Is 1 when a serial poll is in process.
Bit 0: Is 1 when the EOI (end of record) line is true.

Figure 55d

The information which is most useful to the user's program is contained in the **fourth** status byte, which is the one returned as the result of the read-status operation when only the select code of the HP-IB card itself is specified.

Bit 7 of this byte is an indicator that a service request is currently active. Notice that bit 5 of the **third** status byte also deals with the service request line (SRQ). Bit 5 is a one whenever the SRQ line itself is set, and becomes a zero whenever the SRQ line is cleared. The service request bit in the **fourth** status byte, however, is only set if SRQ is set **and** the 98034A card is the active controller. Thus it indicates that this is a request which the System 35, as active controller, is being asked to service.

Bits 6 through 3 indicate which combination of the four possible bus roles (talker, listener, active controller, and system controller) is currently true for the 98034A card. Bit 1 indicates that a serial poll operation is being conducted on the 98034A card by the active controller on the bus.

Bit 0 is set whenever a data character is received by the 98034A (as a listener) with the EOI line set. While the EOI indicator (bit 7 of the third status byte) is a direct indicator of the state of the EOI line, the EOR bit (bit 0 of the fourth status byte) is set only when data is received with EOI true, and is cleared when the status byte is read by the computer.

Unlike the other interface cards whose only interrupting condition is the ready state of the flag line (see "The 98032 Bit Parallel Interface"), the 98034A can interrupt on seven distinct conditions. The most common of these is an interrupt for a service request (SRQ) from another device on the bus.

R5 OUT

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SRQ	ACT	TLK	LST	IRF	ORE	OTHER	EOI

Bit 7: Logical 1 enables interrupt on SRQ.
Bit 6: Logical 1 enables interrupt on active controller.
Bit 5: Logical 1 enables interrupt on active talker.
Bit 4: Logical 1 enables interrupt on active listener.
Bit 3: Logical 1 enables interrupt on input register full.
Bit 2: Logical 1 enables interrupt on output register empty.
Bit 1: Logical 1 enables interrupt when error detected, device clear or selective device clear/received (when not active controller), or EOI received.
Bit 0: Enable EOI to clear status line (STS).

Figure 56

Figure 56 shows the eight conditions which can be specified in the interrupt enable mask. Bit 6 indicates that an interrupt should be generated whenever the 98034 card is made the active controller (i.e., a take-control message is sent from the current active controller). Bits 5 and 4 enabled an interrupt upon being addressed as a talker, or addressed as a listener. The interrupt for these conditions will be generated by the 98034 as a result of two possible circumstances. Either the interrupt enable bit is set and the corresponding condition becomes true, or the interrupt bit is enabled and that condition is already true (that is, the condition is true at the time the interrupt enable mask is sent to the 98034). Thus, for example, the fact that the talker-enabled bit is set and the card addressed as a talker generates an interrupt. As long as the card is addressed as a talker and the bit is set the interrupt is generated. As a result, it is necessary to enable and disable the interface for interrupts on these conditions. If the "interrupt on addressed to talk)) bit is set, an interrupt will be generated each time the 98034 received its talk address from the controller. These three bits remain set until the user's program clears them with another interrupt enable mask containing a zero in these positions (or when the interface is reset from the computer).

Bit 1 of the interrupt enable mask allows an interrupt to occur if the device-clear or error bits (status byte one) are set. The remaining interrupt conditions (bits 3,2, and 0) are used by the I/O ROM during buffer transfer operations. Their correct use is highly dependent on timing and protocol considerations; and as such, they do not represent interrupting conditions which can be useful to a high-level program. Bit 0 cannot actually interrupt the computer, but when set, is used to clear the STS line.

Figure 57 shows the register assignments used by the 98034A Interface card.

	IN	OUT
R4	DATA IN	DATA OUT
R5	STATUS	CONTROL
R6	STATUS/DATA	COMMANDS
R7	PARALLEL POLL	DIRECT BUS CONTROL

Figure 57

Most HP-IB operations use complex sequences of these register operations, which are handled automatically by the I/O ROM in response to high-level statements discussed in previous sections. As a result, in most cases it is neither practical nor desirable for the user's program to attempt to carry out HP-IB operations by using the interface IO statements to directly access these registers.

The one exception to this is in the use of EOI. We have seen that the EOI line is used to indicate the end of a data message when binary data is being sent over the bus. Normally, when ASCII data is being sent, a special character such as LF (line feed) is used to terminate the message. If EOI must be used, buffer transfers will recognize this condition as a termination of the input transfer operation. The 9845A does not, however, send EOI automatically[1] with any data messages. If the user's program wishes to set EOI, this can be done using an R7 OUT operation. In fact, all five of the bus control lines can be set or cleared using the bit mapping shown in Figure 58.

1 The System 35 and later versions of the System 45 desktop computers have an EOI statement that generates the EOI message automatically.

R7 OUT

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	0	EOI	IFC	ATN	REN	SRQ

Figure 58

When the upper three bits of the R7 OUT register are 100, the lower five bits directly address the bus control lines. In each position, a 1 will set and a 0 will clear the corresponding line. For example, to send 100 bytes of data using EOI with the last byte, the following program could be used.

```
100     OUTPUT 713 USING "#,K";A$[1,99]
110     EOI 7;NUM(A$[100;1])
120     !
130     !  NOTE THE EOI STATEMENT IS EQUIVALENT TO...
140     !       WAIT WRITE 7,7;144
150     !       WRITE BIN -7;NUM(A$[100;1])
```

Before sending the last byte, the program addresses select code 7 and outputs a 144 (binary 10010000) to the R7 register to set EOI. Then the last byte is sent with EOI set. Remember that all five bus lines are set or cleared by this operation. Thus, for example, if we wanted to set EOI and leave REN set (assuming that it was set before this operation) we would have used a 146 (binary 10010010) instead of the 144.

It should also be kept in mind that not every device on the bus is allowed to set the bus control lines. Figure 59 shows the role that a device must currently have to set each of these lines.

EOI	Talker
IFC	System Controller
ATN	Active Controller
REN	System Controller
SRQ	Non-controller

Figure 59

Finally, Figure 60 shows the responses of the 98034A when it receives the various bus control messages.

ATN:	As a non-controller, the 98034A gives its attention to the controller and will not respond (flag indicates busy) to the computer during ATN true.
IFC:	Clears all registers and indicators to the power-on state except for the interrupt-enable mask and the serial poll response byte.
REN:	No response.
EOI:	Terminates data input transfer to a buffer. Does not terminate simple read statement.
SRQ:	Sets the service request bit (bit 7 of fourth status byte) and interrupts if bit 7 of interrupt mask is set.
DCL, SDC:	Sets bit 2 of first status byte and interrupts if bit 1 of interrupt mask is set.
GTL,LLO:	No response.
GET:	No response.
Serial poll:	98034A delivers the currently set serial poll response byte without computer intervention.
Parallel poll:	98034A responds to a parallel poll using the line and sense set by the switches on the card.
PPU, PPC:	Parallel poll response is switch settable and not programmable by the controller. No response.
TCT:	98034A assumes active control of the HP-IB.

Figure 60

The 98036A Serial I/O Interface

An Introduction to Serial I/O

In the previous sections we have discussed interfacing peripheral devices to the computer in various formats including 16-bit parallel (98032A), BCD (98033A), and the HP-IB instrumentation bus (98034A). In all of these cases, the cards are used to interface local peripherals and instrumentation clusters which are physically located near the computer itself. Some applications, however, may require the use of peripheral devices which are located at considerable distances from the computer.

Historically, this need arose when the size and speed of computers made it practical for them to do multitasking; that is, being shared by several users at the same time. To do this, each user required his own port into the computer, called a terminal, through which he could enter programs and data and get back printed results. This so-called time-sharing made it possible for each user to access a central computer from a terminal located in his own office or work space. The standard methods of interfacing, however, were not practical in this case since the cost of running cables containing many wires over these distances would quickly become prohibitive. A method of interfacing was needed that would require the fewest number of wires to connect the terminal to the computer.

The solution to this problem was found in a new method of data transmission called serial I/O. In this method, all data is sent and received over a single pair of wires in a bit-serial manner; that is, a word or byte of data is transmitted on a single wire, and received on a second wire, one bit at a time. We will see later that in some cases, more than two wires are used to achieve special features. But in all these cases, the transmission of data one bit after the other is a characteristic of serial interfacing.

This method of connecting terminals to a computer soon led to connecting one computer to another so that they could exchange programs and data. And it became possible to connect terminals and computers located in different buildings, cities, and even countries by making use of the already existing telephone lines. But because telephone lines were not designed to transmit digital (i.e., discrete voltage level) signals, a device that would translate the digital signals produced by a serial interface into analog (i.e., modulated audio tones) signals that could be carried over telephone lines was required. Such a device is known as a data set or a modem (**mo**dulator-**dem**odulator). Figure 61 shows how a pair of such modems would be used to connect a computer to a remote terminal or to another computer.

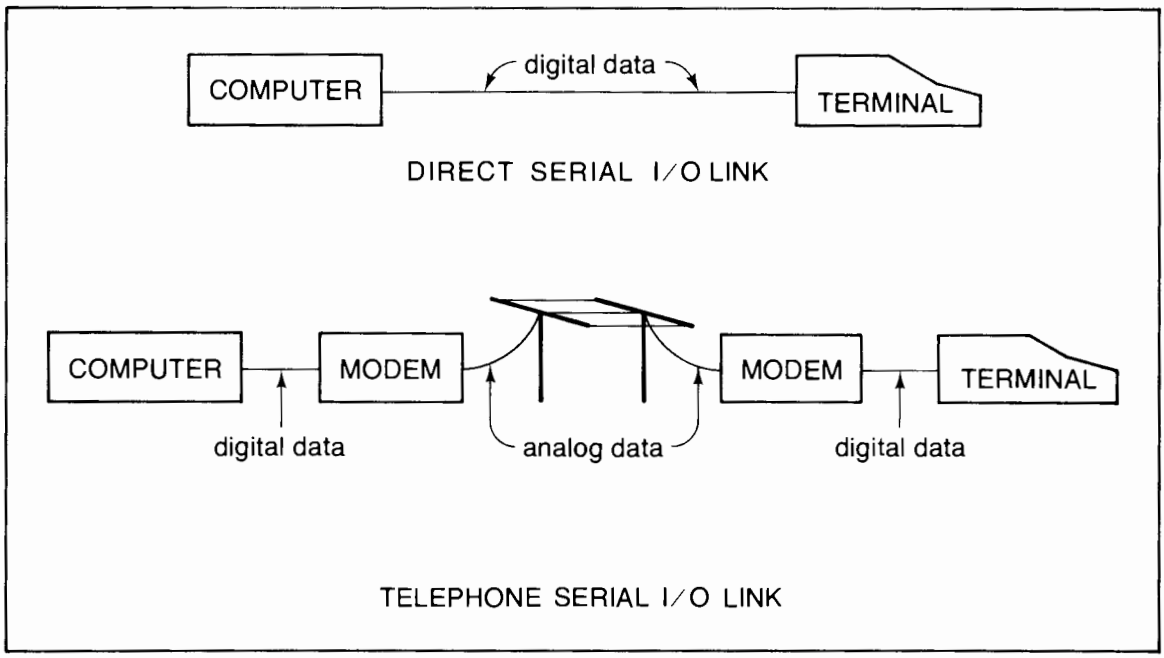


Figure 61

Although the interfacing of remote devices is the primary use of serial I/O, it is by no means restricted to this use. Many peripheral devices such as keyboards and printers are available which use a serial communications link to the computer, even though they may be physically located very near that computer. Because of the large number of manufacturers making modems and data terminal equipment, a need for some standard for compatibility was recognized leading to the RS-232-C standard for serial interfacing in the late 1960's. Since this was the most common standard available prior to the IEEE-488-1978 (see the previous section), many manufacturers of peripheral devices designed them with serial interfaces to take advantage of this compatibility.

Data Transmission Using Serial I/O

In this section, we will discuss in detail the method by which data is transmitted over a serial communications link, and introduce some of the terminology associated with serial I/O. The concepts involved are not difficult, but unless they are understood a great deal of confusion can result.

As with the other methods of interfacing, information is most commonly transmitted over the data line using two voltage levels to represent the two possible states of a binary digit or bit (1 or 0). We will see later that another convention called current loop is sometimes used in which current levels, rather than voltage levels, are used to represent this information. Figure 62 shows the voltage levels for these two states, and the meanings assigned to each.

State:	LOW	HIGH
Voltage range:	−3 to −25V	+3 to +25V
Binary state:	logic 1	logic 0
Level name:	mark	space

Figure 62

When data is not being transmitted, the line is held in the LOW state. Unlike the other methods of interfacing, the serial protocol does not use any type of handshake process. When the transmitting device has a byte of information ready to send, it merely puts the information on the data line, expecting the receiving device to be ready to take it. If the first bit of the data byte sent happens to be a logical 1 (LOW state), the receiver could not distinguish this bit from the quiet line, which is also a LOW state. Therefore, each byte of data is preceded by a start bit, which is defined to be in the HIGH state. This transition from the LOW state (idle line) to the HIGH state (start bit) lets the receiver know that a byte of data is being transmitted. As an example, let's look at how the transmitter would encode the ASCII character "E" to be sent over the data line.

Figure 63 shows the state changes that take place on the data line to send the ASCII "E." The transmitting device first pulls the data line HIGH (start bit) to tell the receiver that a data byte is coming. It holds the line high for an amount of time agreed upon between the transmitter and the receiver, called a bit time. Following the start bit, the bits of the data byte itself are placed on the data line. The least significant bit (bit 0) is sent first, and each bit is held on the line by the transmitter for one bit time. When the receiver senses the leading edge of the start bit, it waits for one half of a bit time in order to synchronize itself as closely as possible to the center of that start bit. Then, each bit time interval after that, it samples the state of the data line and reads a logic 1 or 0. These time intervals at which the receiver samples the data line are marked by ticks in Figure 63. After the last (most significant) data bit has been sent, the transmitter may also send a parity bit (marked P in Figure 63) which will be discussed later.

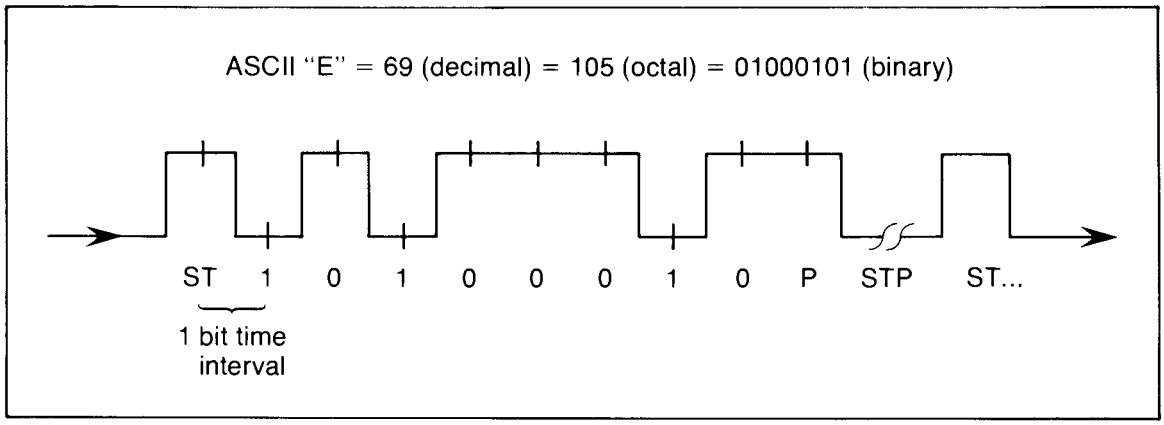


Figure 63

From this diagram, we see that the successful transmission of a data byte is highly dependent on precision timing. If the receiver is sampling the data line at a rate significantly faster or slower than the transmitter is setting that line, it is possible that the receiver will either miss a bit, or sample the same bit twice, resulting in erroneous data being received.

After the last data bit has been sent, the transmitter then allows the line to stay in the idle (low) state for some set minimum time interval before sending the next start bit to begin the next character transmission. This idle time is sometimes called a stop bit, although it does not actually represent a bit of real data. It merely allows the receiver time to process the data byte just received before the next one comes along. For some devices, one bit time may not be enough to process the previous character and be ready for the next one. In this case, the transmitter and receiver may agree that the transmitter will wait in the idle state for 1.5 or 2 bit times before sending the next start bit.

In the example used in Figure 63, we sent the character "E" using an 8-bit ASCII code. That is, eight of the bits sent represented the actual data. If we include the start bit, a parity bit, and one stop bit, we see that 11 bit times are actually required to send an 8-bit byte of data. As an example, lets assume that the overall transmission rate we are using is 10 characters per second. This data rate is very common among printing terminals such as the popular Teletype ASR Model 33. Figure 64 shows the timing characteristics for an example case.

Character length =	7 bit
Bits/character =	7 + Start + Parity + 2 Stop = 11
Data Rate =	10 characters/second
Bit rate =	(10 char/sec)(11 bits/char) = 110 bits/sec
Bit time =	1/110 bits/sec = 0.009091 sec = 9.1 msec

Figure 64

Thus we see that at 110 bits per second, each bit is held on the data line for approximately 9.1 milliseconds.

This bit rate of 110 bits per second is sometimes referred to as a baud rate, and we often speak of the data channel running at 110 baud. Strictly speaking, a binary data channel (i.e., one using low and high voltage levels) should only be described by the term bit rate, the word baud being reserved to characterize the data transmission rate of the analog signals sent between modems. Because of bit compression schemes used in some modems, the bit rate and the baud rate may not always have the same value. For our purposes, we will treat modems as transparent devices that convert digital information to analog and back to digital for long distance communication; and as such, we will only be concerned with bit rates.

The field of data communications and serial I/O probably has more terms associated with it than any other method of interfacing. It is probably also the area in which the terms are most commonly misunderstood and misused. In order to avoid some of this confusion, we will spend the remainder of this section discussing some of those terms and concepts that will be useful to understand when using the 98036A Serial I/O Interface.

Returning to Figure 63, we see that although the bits within the data byte must be sent at precise intervals, there is no restriction on the time between characters except that the required stop time be allowed. Indeed, it is this lack of a time restriction that makes the start bit necessary, so that the receiver will recognize the next character. This mode of transmission is called bit-synchronous, character-asynchronous or simply asynchronous transmission.

Up until now we have talked about serial transmission as though it took place over a single wire. Obviously, a common signal ground is also required so that both the transmitter and the receiver can measure the voltage levels on the data line with respect to the same reference point. Thus the simplest form of serial communication requires two wires for the data transmission.

If communication over the data line is always in one direction, the data channel is said to be operating in the "simplex" mode. For example, an RS-232-C printer would only receive information, while a device such as a tape reader would only transmit data. A terminal, however, may both send and receive data since it has both a keyboard (data transmitter) and a printer or a CRT (data receiver). In the previous section, we saw that the HP-IB allows bidirectional communications over a set of data lines. That is, the same set of data lines is used for sending information from device A to device B, and for sending information from device B to device A. A special HP-IB protocol (i.e., addressing talkers and listeners) is used to control the traffic on this set of data lines.

Since communication in one direction over an RS-232-C link uses only one wire, such protocols can be avoided by allocating separate transmit and receive data lines, with a common signal ground line used for reference. Figure 65 shows a schematic representation of such an RS-232-C data link.

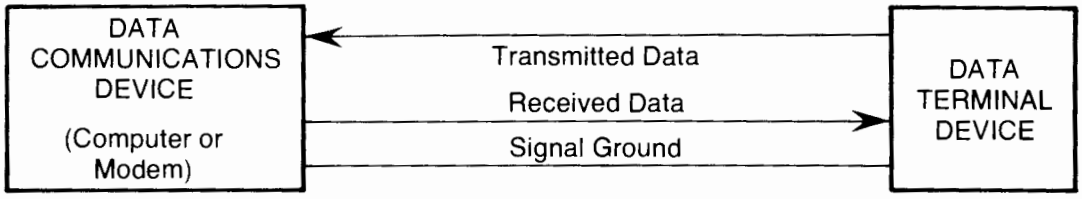


Figure 65

Typically, the devices represented in Figure 65 will be a computer (or a modem for remote communications) and a terminal. But this same diagram can also represent a link between any two RS-232-C devices. For this reason, the two devices are often referred to by the more general terms Data Communications Equipment (DCE) and Data Terminal Equipment (DTE). Also, the terms transmitted data and received data are defined relative to the terminal (DTE device).

An output-only device operating in the simplex mode would only implement the transmit and ground lines, not using the received data line. An input-only device would implement the received data and ground lines. If a device can both transmit and receive data, it would implement all three lines. Such a device is said to be operating in the "duplex" mode.

When two devices are directly connected over an RS-232-C link, they normally operate in what is called a full-duplex mode. This means that data may be transmitted and received simultaneously. Information may be carried from the DTE to the DCE on the transmitted data line at the same time that information is being sent from the DCE to the DTE on the received data line.

If these devices are connected over telephone lines using a pair of modems, only one data path (the phone line) interconnects the two modems. In this case, full duplex operation is still possible since many modems are capable of multiplexing the two signals representing the transmitted and the received data. As the transmission speed (baud rate) increases, however, the amount of information being sent by both transmitters simultaneously eventually exceeds the capacity of the telephone line. Thus, when using high-speed modems, a special protocol is used called half-duplex in which only one device at a time is allowed to transmit data to its modem for sending over the telephone line.

Figure 66 shows a schematic representation of all three of these modes of operation. Notice that while the computer is playing the part of a modem in the simplex and full-duplex modes, in the half-duplex mode it is operating as a terminal, as is shown by the labeling of the transmitted and received data lines.

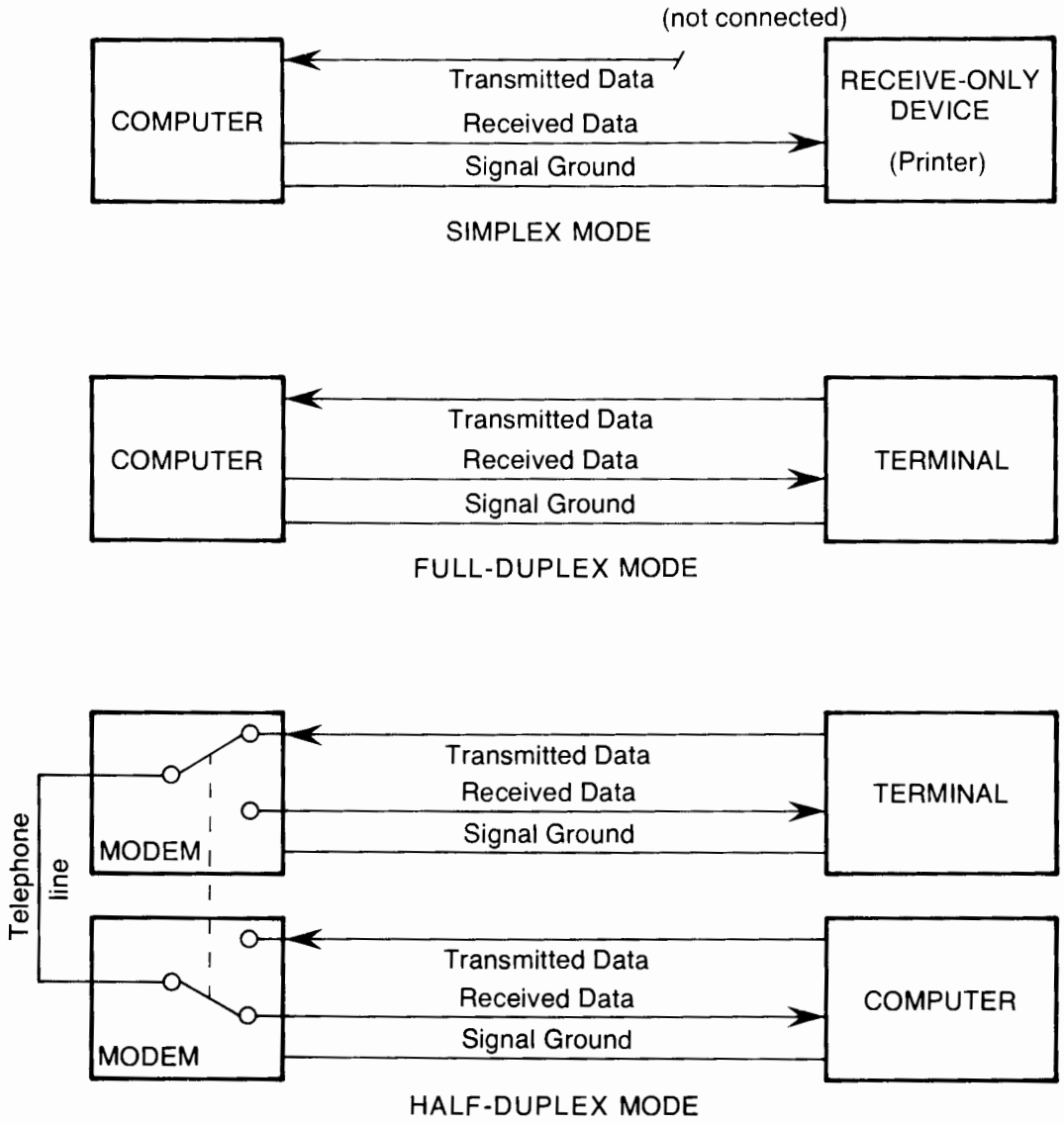


Figure 66

So far we have been using the terms half-duplex and full-duplex to describe the characteristics of the communications line itself. These terms are also applied to classify terminal types with similar, but not quite identical meanings. To clarify this, lets look at the characteristics of a terminal in operation.

Figure 67 shows a schematic representation for a half-duplex terminal. It consists of a keyboard for entering information to be sent to the computer, an output device such as a printer or a CRT for displaying information sent back by the computer, and some electronic hardware for selecting the transmit or the receive mode of operation on the half-duplex data line.

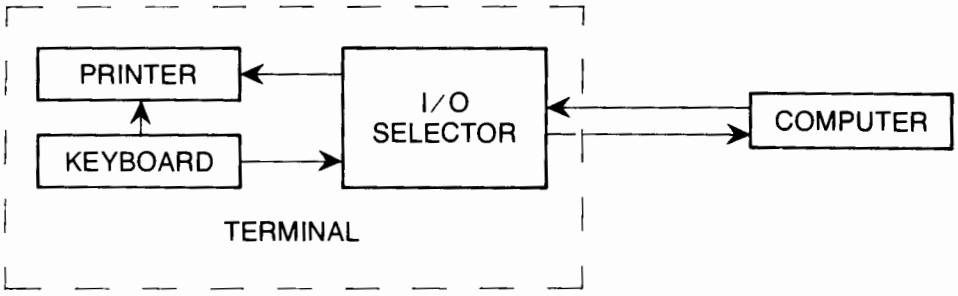


Figure 67

Information typed in on the keyboard is sent to the computer, and its responses are sent to the terminal's printer. Since it is very difficult to type on the keyboard without some visual feedback as to which keys have been pressed, the half-duplex terminal will also send its keystrokes to the printer as indicated by the arrow in the figure. Thus the printer shows a record of both the input from the keyboard as well as the output from the computer.

If, due to electrical noise on the data line, a bit is dropped (i.e., a transmitted 1 or 0 is received as a 0 or 1) the computer will not receive the same message as sent by the terminal. But since the information on the printer was generated by the keyboard, the message looks correct even though the computer responds with an error indicating that it did not understand the message received. This problem can be alleviated by operating the terminal in the full-duplex mode, and taking advantage of a capability offered by many timeshare computers called echo-back or simply echo. Figure 68 shows how a terminal would operate in this mode.

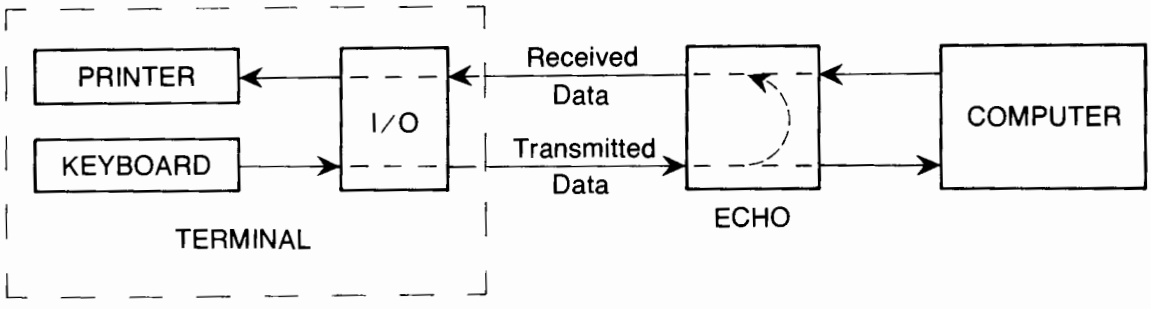


Figure 68

The keyboard on the terminal does not directly drive the printer. Instead, as each key is pressed, it is transmitted to the computer which receives it for processing, and also echoed back to the terminal to be output on the printer. In this mode, the operator at the terminal still gets visual feedback of what has been typed. But the characters printed are those received by the computer and echoed back. Now if there is a transmission error, the operator sees the incorrect character on the printer and can send a backspace character and retype the correct character.

Some terminals operate in only the half-duplex or the full-duplex mode while others will operate in either mode, usually selectable by a switch on the terminal itself. If a terminal is operating in the half-duplex mode with a computer which echoes characters back, each key-stroke will be printed twice — once from the keyboard and once from the echo-back. Thus typing the message "HELLO" would result in the printer showing "HHEELLLLOO." On the other hand, if a full-duplex terminal is communicating with a computer that has no echo-back capability (or has this feature turned off), neither the computer nor the keyboard is driving the printer during the typing of messages at the terminal, and the operator is "running blind." This close association between half and full duplex operation of a terminal, and having echo turned on or off, can lead to confusion unless this association is understood. When a selectable terminal is run in the half-duplex mode, the keyboard drives the printer and echo on the computer should be suppressed. In the full-duplex mode the keyboard does not drive the printer, and the echo-back feature of the computer should be on. If the particular computer being used cannot have its echo turned on or off, this will dictate the mode of operation of the terminal.

Control Lines and the RS-232-C Standard

Until now we have been concerned only with data transmission over serial I/O lines. This method provides a simple means of communication over a minimum number of wires, but does not allow for much flexibility. As data communications equipment became more sophisticated, the need for more control arose. For example, if a data terminal device offered the ability to perform more complex tasks (e.g., save a block of received data on a magnetic tape unit), it might require more than the provided stop-bit time between characters to perform these operations. With the advent of telephone communications and modem equipment, other new needs arose such as the ability to detect when a computer was trying to dial up a modem, and when it had dropped the line (i.e., hung up) at the end of the communication. In an attempt to prevent total confusion in this area with every manufacturer implementing these control features in whatever manner they chose, resulting in lack of compatibility between serial I/O devices, the Electronic Industries Association (EIA) tried to define a standard to guide designers of serial I/O equipment. After several proposals and earlier standards, the EIA RS-232-C standard was adopted in 1969 and is used today by a large number of manufacturers of data communications equipment.

Even though the RS-232-C standard is the most popular one in use today, several other standards exist which allow for more capabilities in certain areas of application. Some of these are very close to the RS-232-C in their definitions. And the user setting up a serial I/O system should be careful to recognize equipment which claims to be RS-232-C compatible but may have "slight" differences. In any given application, these differences may or may not be enough to prevent compatibility with a true RS-232-C device.

Data communications devices which are RS-232-C compatible use a standard EIA 25-pin connector, shown in Figure 69. The computer or modem (DCE) cable terminates with a female connector, and terminal devices (DTE) use a male connector. Although this polarity is the common one, some devices will be found which use the opposite type of connector. The problems that this can cause will be discussed in a later section.

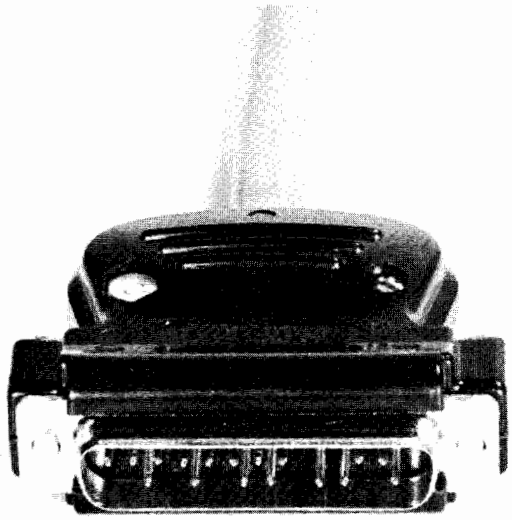


Figure 69

Figure 73 shows the various data and control lines that have been assigned to each of the connector pins by the RS-232-C standard. The arrows are used to show the direction of each line. That is, an arrow to the right signifies that the signal described is generated by the DCE device and received by the DTE; while an arrow to the left signifies a signal from the DTE to the DCE. Notice that the terms transmitted and received data (pins 2 and 3) are named relative to the data terminal device. In the following paragraphs, we will describe each of these lines; not in the order of their pin assignments, but collected into logical groups according to their functions.

Protective Ground (Pin 1)

This line is connected to the chassis ground of the device which is usually connected to the external ground of the power supply for safety reasons.

Transmitted Data (pin 2)
Received Data (pin 3)
Signal or Logic Ground (pin 7)

These three lines are used to obtain full-duplex data exchange and have already been discussed under "Data Transmission Using Serial I/O."

Request to Send (pin 4)
Clear to Send (pin 5)
Data Set Ready (pin 6)
Data Terminal Ready (pin 20)

These four lines perform status indication functions between the modem and the terminal, and indicate various go/no-go conditions. The Data Set Ready (DSR) and Data Terminal Ready (DTR) lines are similar to the PSTS line on the 98032A card. When they are on (high voltage level), they indicate that the device is operational. For example, the data set (modem) might turn off DSR if it were switched into the test or dial mode, or if it lost the carrier signal on the telephone lines. Similarly, the terminal would turn off DTR if it were switched from the on-line to the local mode of operation.

The Request to Send (RTS) and Clear to Send (CTS) lines perform different functions depending on the mode of operation. In the half-duplex mode, they are used to control the channel direction, or direction of communication flow on the data line.

Normally these lines are used by data communications equipment manufacturers to implement the various serial I/O protocols, and are not of concern in simple data exchanges between RS-232-C devices. The user should be aware, however, that some modems and terminals will not operate properly unless certain of these lines are set to the on state. We will discuss this further in the next section when we see how the 98036A card sets and clears these lines.

Ring Indicator (pin 22)
Carrier Detect (pin 8)
Signal Quality Detector (pin 21)
Data Signal Rate Selector (pin 23)

These four lines are used when the terminal is operating with a modem using telephone communications. The Ring Indicator indicates that the telephone ringing signal is being received on the communication channel. The Carrier Detect indicates that the acoustic signal or tone that is modulated to carry the data information is being received. Loss of this carrier

indicates that the communication channel is no longer established. Some modems can detect from the waveform of the carrier signal when there is a high probability of an error in the received data. This condition is indicated by the state of the Signal Quality Detect line. The Data Signal Rate Selector line is used by some modems with dual rate capability to switch between two data signaling rates.

When connecting RS-232-C devices to HP desktop computers using the 98036A Interface, these lines would not normally be used, although they are made available for setting and testing by the interface.

Transmitter Clock (pin 15)
Receiver Clock (pin 17)
Transmitter Clock (terminal source) (pin 24)

Normally, each device has its own internal clock signal used to send and receive data bit patterns at the correct bit-time intervals. If a device does not have its own clock, or if it wishes to use the other device's clock for special data rates or synchronization purposes, these lines are used to transmit those clock pulses.

Secondary Transmitted Data (pin 14)
Secondary Received Data (pin 16)
Secondary Request to Send (pin 19)
Secondary Clear to Send (pin 13)
Secondary Carrier Detect (pin 12)

These lines are assigned by the RS-232-C standard in order to allow for a second data communications channel. The 98036A does not support this secondary channel, although two of the control lines assigned for this channel (Secondary Request to Send and Secondary Carrier Detect) are made available to the computer and can be used for whatever purpose the user finds convenient. This assumes, of course, that the device being interfaced to does not use these lines for their assigned meanings.

As we will see in the next section, some of these lines are implemented by the 98036A Interface while others are not. For example, the transmitter and receiver clocks (pins 15 and 17) can be externally controlled on the 98036A, while the terminal source transmitter clock (pin 24) is not implemented. Some of the secondary channel lines are provided since they are sometimes used in implementing half-duplex protocols.

The 98036A Serial I/O Interface

In this chapter, we showed how a serial communications link is used to connect a computer to a remote terminal. Using the 98036A Interface card, HP desktop computers can participate in this communications link, acting as a substitute for either the computer, the terminal, or both. In the last case, the serial I/O link is used to allow two desktop computers to be connected together for program and data exchange. Figure 70 shows a schematic representation of these three methods of interfacing.

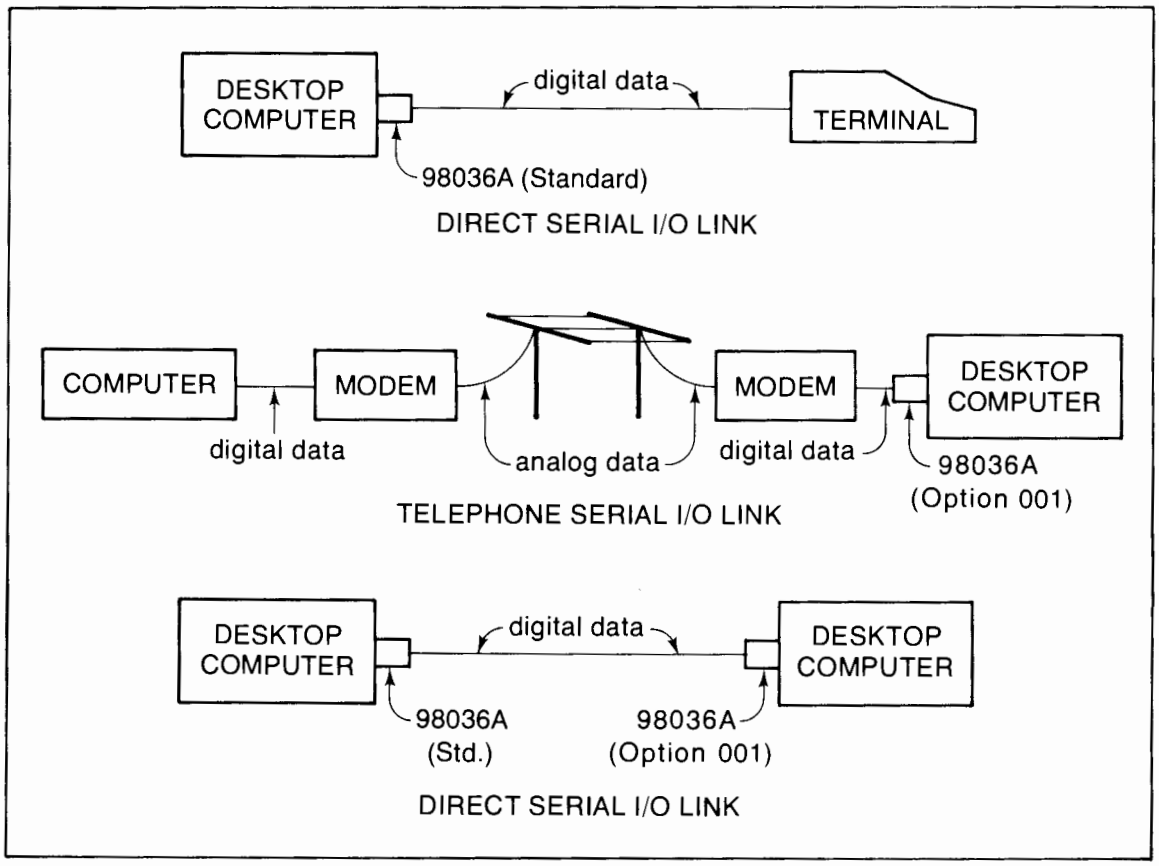


Figure 70

The desktop computer can be any one that uses the 98036A Interface, and we will be using the System 35 as a representative example. Depending on whether the System 35 is assuming the role of the computer or that of the terminal, the RS-232-C pin assignments will be slightly different. For example, as a terminal, the 98036A card should transmit its data on pin 2 and receive on pin 3 (see Figure 73). But the same 98036A card on the computer end of this link will receive data on pin 2 and transmit on pin 3. (Remember that the terms transmit and receive are named relative to the terminal.) Therefore, two versions of the 98036A Interface are required. The standard version makes the System 35 look like a computer or a modem, and is used to connect it to a terminal device. The option 1 version makes the System 35 look like a

terminal and allows it to be connected to a computer or a modem. When we say that the System 35 looks like a modem or a terminal, we mean only the way in which it handles the data communications channel. Additional software (i.e., a program in the System 35) is required to allow it to emulate the actual operation of a modem or a terminal. The only function of the interface card is to send and receive information over the data line, and to make the various RS-232-C control lines available for setting and testing. Any higher capability such as terminal emulation must be handled by a running program in the System 35.

Figure 71 shows the meanings that have been given to the interface registers on the 98036A in order to access the various data and control lines.

	IN	OUT
R4	DATA IN, R4E[1]	DATA OUT, R4C, R4D[1]
R5	STATUS	CONTROL
R6	LINE STATUS	LINE CONTROL
R7	(not used)	TRIGGER

Figure 71

Data input and output with the 98036A is handled in the same way as described for the 98032A Interface. That is, data bytes are sent and received through the R4 registers, and the R7 OUT register is used as a trigger. The same drivers presented in the section for the 98032A Interface are used to exchange data with the 98036A Interface. Using these drivers, the card is operated in the half-duplex mode only. Data may be sent or received, but not both at the same time. We will see later how the interrupt structure of the System 35 can be used to allow full-duplex operation.

Data exchange with the 98036A card is done as though it were an 8-bit parallel interface. An entire byte of data is sent to the card via the R4 OUT register. When the R7 OUT trigger is issued, the interface automatically breaks it down into a sequence of serial bits and supplies the required start and stop bit (plus a parity bit if parity is being used). It also takes care of the necessary timing requirements.

Most of the complex protocol for exchanging data over the RS-232-C channel is handled by a large-scale integrated circuit (Intel 8251) called a USART (Universal Synchronous/Asynchronous Receiver and Transmitter). This USART is the heart of the 98036A and implements most of the data, timing, and control requirements specified by the RS-232-C standard. The

1 These registers are also used for special status and control information.

remaining electronics of the 98036A provide an interface between this USART and the I/O backplane of the desktop computer. It should be mentioned that even though the USART is capable of supporting synchronous communications on the data channel, a complex driver would also be required in the desktop computer to implement one of the byte-oriented synchronous protocols (e.g., BISYNC), since these protocols are not provided by the USART itself. As a result, only the asynchronous mode of operation is supported on the System 35 using the 98036A Interface.

The status (R5 IN) and control (R5 OUT) registers on the 98036A are used for setting and testing various modes of operation of the interface card itself. Figure 72a shows the assignments that have been made for the bits in these registers.

R5 OUT Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Interface Interrupt Enable		Programmed Interface Reset			Interrupt Control 2 Receiver Control	Interrupt Control 2 Transmitter Control	R4 Control 0=Data IN/ OUT 1=Control/ Status

R5 IN Register

Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Peripheral Status 1 Mode	Interface Interrupt Enable Status	0	Interface I.D. 0	Interface I.D. 1	0	0	Control Status 2 Receiver	Control Status 1 Transmitter Mode

Figure 72a

Most of these bits are used for operating the card in the interrupt mode, and will be discussed in the next section on interrupt programming. The remaining bits will be discussed here.

Bits 4 and 5 of the status (R5 IN) byte contain the interface type identification bits (see Chapter 2). Bit 5 of the control byte (R5 OUT) is set to a 1 to cause the 98036A to return to its power-on state. The USART itself on the 98036A card makes use of two full bytes of control information, and provides one byte of its own status information. Since there are not enough bits in the R5 registers to contain all of this information, the 98036A card utilizes a multiplexing scheme to gain access to these USART registers. This scheme works in the following manner. If bit 0 of the control byte (R5 OUT) is set to a zero, the R4 registers have their normal meanings of data in and data out. If, on the other hand, this bit is set to a one, the R4 registers are now used to access the USART status, mode, and control bytes. Because of this mode of access, these USART registers have been given the names R4C, R4D, and R4E. Figure 72b shows the meanings given to the various bits in these registers.

R4C Mode Word

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Number of Stop Bits 00=not valid 01=1 bit 10=1.5 bits 11=2 bits		Parity Type 0=Odd 1=Even	Parity Enable 0=Disable 1=Enable	Character Length 00=5 bits 01=6 bits 10=7 bits 11=8 bits		Bit Rate Factor 00=not used 01=1 X bit rate clock 10=1/16 X bit rate clock 11=1/64 X bit rate clock	

R4D USART Control Word

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Always 0	USART Reset	Clear To Send Pin 5 (Standard) Request To Send Pin 4 (Option 001)	Reset Status Bits of USART Status Word	Send Break Character	Enable Data Receiver	Data Set Ready Pin 6 (Standard) Data Terminal Ready Pin 20 (Option 001)	Data Enable Transmitter

R4E USART Status Word

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	Always 0	Framing Error	Overrun Error	Parity Error	Transmitter Empty	Receiver Ready	Transmitter Ready

Figure 72b

Before discussing the meanings of these bits, let's look at how each of them is accessed. The USART Status Byte (R4E) would be obtained by setting bit 0 of the interface control register (R5 OUT) and as a precaution, bit 0 of the CONTROL MASK control byte to a one. A normal data input operation is then performed. Since this control bit is set to a one, this tells the interface to place the USART status byte in the input (R4 IN) register, rather than a normal data byte. When this sequence is finished, the card should be returned to the data mode by setting bit 0 of the control byte (R5 OUT) back to a zero. Thus, the sequence

```
10      WAIT WRITE 11,5;1
20      A=READBIN(11)
30      WAIT WRITE 11,5;0
```

would result in reading the USART status byte (R4E) and placing its decimal equivalent in the variable A. In a similar manner, the sequence

```
10    WAIT WRITE 11,5;1
20    WRITE BIN 11;32+4+1   ! Set RTS,Rec and Trans Enable
30    WAIT WRITE 11,5;0
```

would output the contents of the variable X to the USART control byte (R4D). The USART also uses a mode word (R4C) which is accessed through the following sequence.

```
10      WAIT WRITE 11,5;1
20      WRITE BIN 11;64,A,B
30      WAIT WRITE 11,5;0
```

The 98036A (set to select code 11) is set to the control mode and a decimal 64 (binary 01000000) is sent to the USART control byte (R4D) as in the example above. This sets bit 6 of the R4D register, which is a reset for the USART. In addition to its other reset functions, it also places the USART in the mode where the next two bytes output are sent to the R4C and the R4D registers respectively. In the example above, the value of variable A would be sent to R4C and variable B to R4D. Of course, bit 6 of the binary representation of B should not be a 1 or the USART will again be reset, nullifying the output to R4C and R4D.

The R-232-C standard specifies certain characteristics of the data line such as the voltage levels used, the start/stop protocol, etc. Other characteristics of the data transmission are left more flexible by the standard. These include the following: the bit rate (bits per second) at which the data is transmitted; the number of bits per character; the type of parity (even, odd, or none) to be used; and the number of stop bits. These characteristics can be chosen to suit the given application, so long as the sender and the receiver both agree on the particular set of characteristics to be used. Unless all four of these characteristics are the same for each end of the channel, the data transmitted will not be properly interpreted by the receiver. For example, if the transmitter is sending data at 300 bits per second (bps) and the receiver is operating at 600 bps, the data pattern 10010... transmitted will be received as 1100001100... since the receiver is sampling the data line twice as fast as the transmitter is setting it. When the receiver displays the characters it thinks it has received, they will appear totally unintelligible. Such received data is usually referred to as "garbage."

The 98036A allows for a wide range of flexibility in each of these four categories. The data rate may be set to all of the more common values in the range 75 to 9600 bps. The character length may be set to 5, 6, 7, or 8 bits per character. Most ASCII devices will use 7 or 8 bits per character, with the 5 and 6 bit options only used by special devices that use more limited character sets. The number of stop bits may be set for 1, 1.5, or 2. As mentioned before, these are not actual bits but rather the minimum time that the data line must be held in the quiet (low) state before the next start bit can be sent.

Because of the nature of serial I/O transmission, data on the line is very susceptible to "dropping bits," that is, having a bit sent as a 0 or a 1 being received as a 1 or a 0. In order to detect when this happens, a scheme called parity checking is often used. The transmitter will supply an extra bit that is not part of the data itself to be sent with each character . This parity bit is set in such a way that the total number of bits (both data and parity) set to a one is always even or always odd. Each character that the receiver gets is checked to make sure that the received data has the proper parity. For example, Figure 63 shows the bit pattern used for sending an ASCII "E" character. Since there is an odd number of 1's (three) in the ASCII representation of an "E," and the parity bit is a zero, this particular example is using odd parity. If even parity were being used, the parity bit would be set to a one to bring the total number of 1's to an even number (four).

It is important to note that if parity is not being used, the parity bit is not even transmitted. This can lead to some confusion because of the manner in which other methods of interfacing handle parity. For example, when 7-bit ASCII data is being sent over an 8-bit parallel interface (such as the HP-IB), the eighth bit is not being used for the 7-bit ASCII characters and is sometimes used to send a parity bit along with each data character. If parity is not being used, the eighth data line is still there and is usually always set to a zero. This sometimes leads to the conclusion that in serial I/O, if parity is not being used, the parity bit is always set to a zero. But in actuality, **if parity is not used the parity bit is not even sent.**

When connecting a 98036A card to another serial I/O device, the user must know the values for each of the four characteristics, bit rate, bits per character, parity and stop bits, and set the 98036A to match them. This information is usually contained in the operating manual for that device, or from a timeshare service if the user is going to go on-line to a timeshare computer. As an example of the difficulty that can arise, such a manual or a timeshare service might specify "8-bit data, even parity." After some trial-and-error evaluation, it becomes clear that the device is actually using 7 data bits plus parity, and in their specification they are considering the parity bit to be part of the data. Being aware of this lack of consistent terminology can sometimes save much time in determining the operating characteristics of a serial I/O data line.

All of these options for data line characteristics can be set on the 98036A card by the use of switches, (see the 98036A Installation and Service Manual). In addition, the character length, number of stop bits, and parity can be set through the R4C Mode Word, overriding the switch settings on the card.

The Mode Word also allows the setting of a value called the bit rate factor. Previously, we discussed how both the transmitter and the receiver must measure time intervals called bit times to determine when to set or sample the data line for the next bit. The more precisely these bit time intervals can be measured, the less likely it is that the time intervals of the transmitter and receiver will drift with respect to one another and cause an incorrect data exchange. Normally, an internal clock on the 98036A card is set to run at 64 times the bit rate being used. By dividing the bit time interval into 64 parts, the exact center of a bit on the data line (Figure 63) can be more precisely located. At bit rates greater than 2400 bps, however, the internal clock cannot run this fast. As a result, at 4800 and 9600 bps the bit time intervals are divided into 16 parts instead of 64, dropping the demands on the internal clock back into a range in which it can operate. This bit time interval divider and its associated restrictions are in the USART itself.

The R4D control byte is used to control various functions on the USART itself. We have already discussed how bit 6 is used to reset the USART and to address the R4C mode word. When the 98036A is reset (either by pressing CONTROL-STOP on the System 35 or by setting bit 5 of the interface control register, R5 OUT), a default value of 5 is set for R4D. This sets bits 0 and 2 which enable the USART for data transmission and reception. Normally, these bits are always left on and any output to the R4D register should include these bits. Two other bits, 1 and 5, are used to set or clear the two most commonly used RS-232-C control lines. These are Clear to Send and Data Set Ready when the 98036A is acting as a computer or modem interface; or Request to Send and Data Terminal Ready when the 98036A is acting as a terminal interface (option 001). As mentioned before, many terminals or modems will not operate unless they see one or both of these lines set.

When the data channel is operating in the half-duplex mode, the computer and the terminal follow an agreed upon set of rules that determine when each of them will transmit on the data line. If the computer is currently transmitting a large block of information, the terminal cannot transmit. If it would like to get the computer's attention (for example, to abort the data transfer) it would follow some agreed upon protocol for interrupting the transmission and turning around the communications link. In full-duplex operation, this is accomplished by sending what is called a break character. Strictly speaking, this is not a character in the sense of a transmitted data character. It merely holds the data line high for a period of time that is longer than one complete character time, typically about 200 milliseconds. The receiver of the trans-

mitting device detects this, and can act on it as it chooses. Most timeshare computers are set up to abort the current I/O sequence and return control to the terminal when a break character is detected. We will discuss the 98036A's response to receiving a break character. The break character is sent by setting bit 3 of the R4D register. This holds the transmitted data line high until this bit it again cleared to a zero. For example, the sequence

```
10      WAIT WRITE 11,5;1
20      WRITE BIN 11;47
30        WAIT 200
40      WRITE BIN 11;39
50      WAIT WRITE 11,5;0
```

would set bit 3 of R4D and then clear it after a 200 millisecond wait period. During both the setting and clearing of the break bit, bits 5, 2, 1, and 0 are specified (47 decimal = 00101111 binary) in the logic 1 state so that the transmitter and receiver remain enabled, and the two control lines (bits 1 and 5) remain set.

We will see that three of the bits in the USART Status Byte (R4E) are used as error indicators. Bit 4 of the R4D register is used to clear all three of these error indicators.

The R4E register returns a status byte from the USART itself containing information about various situations that can occur there. Bit 7 is used to monitor either the Request to Send line (standard card) or the Data Set Ready line (option 001 card). Normally these lines are only used when implementing special protocols.

Three of the bits in R4E indicate the status of the input (bit 1) and output (bits 2 and 0) buffers on the USART. During normal program operation, these indicators are of no interest. They are useful, however, in either debugging a program or in making sure that the data channel is properly set up. For example, if a terminal is connected to a System 35 using the 98036A, and data cannot be input from the terminal, checking bit 1 will tell whether data is being received and improperly handled by the program or not being received at all. This bit is set when a character is received by the USART and cleared when the computer takes that character.

The remaining three bits of the R4E register (5, 4, and 3) indicate a framing error, and overrun error, and a parity error respectively. Taken individually, these three errors have simple meanings. A framing error indicates that at the time the receiver was expecting to see the stop bits (low level), the data line was actually high. This could be caused by having the wrong bit rate set. For example, if the transmitter were sending at 300 bps and the receiver was set to 600 bps, the receiver would finish sampling for the data bits (doubly reading most of them!)

when the transmitter was only about half finished sending. The receiver would then look for the stop bit (or bits) in the data region of the character transmission. If the data line went high during this time, the framing error indicator would be set.

It is interesting to notice that the data being transmitted during the time that the receiver is looking for the stop bits could, by chance, be "1" bits (low level) and appear to the receiver to be correct stop bits. Thus, an incorrect character could be received without the framing error indicator being set. The probability of this situation (accidental matching of data bits with stop bits) decreases rapidly with the number of characters received. That is, if several characters can be received without the framing error being set, it is very unlikely that the bit rate selector is improperly set. With only one or two characters, it is difficult to be sure.

It should also be mentioned that once an error indicator is set, it can only be cleared by a reset operation; i.e., a card reset, a USART reset, or the specific error flags reset in bit 4 of R4D. For example, receiving a character without a framing error will not clear the framing error bit if it was set by a previous character that was improperly received. Otherwise, if the last character received were incorrect but accidentally matched the expected stop bits with data bits, the entire string would appear to have been properly received.

The overrun error indicator is set whenever a data character has been received but not taken by the computer before the next one came along. This error indicates that one or more data characters have been lost. The situation can be corrected for future transmissions by either slowing down the data rate, or by using a faster programming method to take the data as it comes in.

The parity error indicator is set when the 98036A is enabled to check parity on received data, and the expected parity bit is not correct.

Although the meanings of the three error conditions are straightforward, when combinations are considered the meaning can sometimes be confusing. Examples of this are apparent from the transmission of the ASCII "E" shown in Figure 63. Assume that the transmitter is sending the pattern as shown, but that the receiver is set for no parity. After reading the last data bit, since no parity bit is expected the receiver will expect the stop bit to immediately follow. Since the line is high at this time (transmitter is sending a parity bit of 0), a framing error is detected. Thus, even though the problem is caused by the fact that the transmitter is set for parity and the receiver is not, it is a framing error that is indicated by the error bits in the USART status word. This points up the necessity of knowing the data transmission characteristics of the device being interfaced with over the serial I/O channel. If these characteristics are not known, it can sometimes be a tricky bit of detective work to analyze the errors indicated and isolate the true cause of the problem.

When connecting an unknown terminal using the 98036A card, it is unwise to try running a complex applications program until simple read binary and write binary operations from the keyboard can be made to work. Otherwise, the user may waste considerable time trying to debug a correct program when the actual cause of the problem is that one or more of these data transmission characteristics is improperly set for that device.

Finally, when the bit rate has been properly set so that a framing error does not normally occur, the presence of a framing error indicates the reception of the break character. Since the line is held high for 200 milliseconds during a break, even at the slowest bit rate, the line will be high for longer than one character time and cause the expected stop bits to be missed. Depending on the number of bits per character set and the type of parity being used, the parity error may also be set during the reception of a break character.

The remainder of the RS-232-C control lines are only used for special applications, and are accessed through various bits in the R6 register. The specific bits that can be set (R6 OUT) or tested (R6 IN) depend on whether the 98036A is acting as a computer or modem (standard card), or as a terminal (option 001). These registers are accessed from the system using the read interface (WAIT READ) and write interface (WAIT WRITE) statements. Although these control lines are usually used for special applications only, they may control lines required by some devices in simple applications. Some terminals will not transmit data unless the Carrier Detect line (bit 0 of R6 OUT) is set, along with Data Set Ready and Clear to Send. Again, successful operation demands that the user know the requirements of the device being interfaced.

Figure 73 shows the RS-232-C pin assignments implemented by the 98036A Interface, along with the names and directions of these lines. On the left is shown the pin connector numbers on the standard 98036A card, and the interface registers used to access each of them. On the right is shown the same information for the Option 001 98036A card.

DCE (Standard)		RS232-C			DTE (Option 001)	
Register Access	**Standard Pin #**	**Direction ↔**	**Pin #**	**Signal Name**	**Option 001 Pin #**	**Register Access**
n.a.	1	↔	1	Protective Ground	1	n.a.
read	3	←	2	Transmitted Data	2	write
write	2	→	3	Received Data	3	read
R4E, bit 7	6	←	4	Request to Send	4	R4D, bit 5
R4D, bit 5	4	→	5	Clear to Send	5	(Note 1)
R4D, bit 1	17	→	6	Data Set Ready	6	R4E, bit 7
n.a.	7	→	7	Logic Ground	7	n.a.
R6 OUT, bit 0	16	→	8	Carrier Detect	8	R6 IN, bit 0
n.a.	—		9	(Reserved for test)	—	n.a.
n.a.	—		10	(Reserved for test)	—	n.a.
n.a.	—	↔	11	Data Rate Select (U.K.) (Note 2)	11	R6 OUT, bit 2
R6 OUT, bit 1	13	→	12	Second Carrier Detect	12	R6 IN, bit 2
n.a.	—	→	13	Second CTS	—	n.a.
n.a.	—	←	14	Second TXD	—	n.a.
n.a.	—	→	15	Transmitter Clock	15	(Note 3)
n.a.	—	→	16	Second RXD	—	n.a.
n.a.	—	→	17	Receiver Clock	14	(Note 3)
n.a.	—		18	—	—	n.a.
R6 IN, bit 0	8	←	19	Second RTS	16	R6 OUT, bit 0
(Notes 1,4)	5	←	20	Data Terminal Ready	17	R4D, bit 1
R6 OUT, bit 2	11	→	21	Signal Quality Detect	—	n.a.
R6 OUT, bit 3	10	→	22	Ring Indicator	9	R6 IN, bit 1
R6 IN, bit 1	9	←	23	Data Rate Select	13	R6 OUT, bit 1
n.a.	—	←	24	Transmit clock (term)	—	n.a.
n.a.	—		25	—	10	R6 OUT, bit 3

Note 1: this line cannot be read
Note 2: this line unassigned by RS-232-C
Note 3: switch selectable on 98036A
Note 4: can be set high by switch on 98036A

Figure 73

Programming with the 98036A Interface

For half-duplex operation, the 98036A is programmed in the same manner as the 98032A Bit-Parallel Interface. Output is done using the OUTPUT or WRITE BIN statements and input is done with ENTER or READ BIN operations of the I/O ROM.

When using the 98036A card in full duplex mode, the programming becomes more complex. One would assume that with the I/O structure of the 9835/45 it would be possible to utilize an interrupt process to handle one direction of data flow (interrupt transfers) and use the flag line that is dedicated to the interface for data flow in the other direction.

One example might be using an ENTER BINT statement for incoming data and using an OUTPUT statement for transmitting data. The limitation with this comes about with the overlap processing capabilities of the 9835/45. There is a select code queue that is associated with each select code in the 9835/45. If the ENTER BINT statement is executed, the select code queue associated with the ENTER BINT is then dedicated to an input process. This select code queue is then dedicated to incoming data until the ENTER BINT statement is completed. If at the same time an OUTPUT statement is executed, the processor looks at the select code queue associated with that output operation and sees that it is dedicated to an input operation. The processor then queues up the request for the output operation in its process queue and continually checks the select code queue that is associated with the OUTPUT statement to see if it has been released by the ENTER BINT statement. When the input process has been terminated the processor then releases the queue for that select code. When the processor again checks its process queue and sees that it has a request queued up, it checks the select code queue associated with the output operation. This time the queue is available and the processor then dedicates the select code queue to an output operation and begins the output transfer.

The way around this is with the TOPEN[1] statement. This statement utilizes the interrupt structure of the card for data input. When a character comes into the interface it signals the mainframe with an interrupt. The TOPEN statement puts each character from the card into a circular buffer called TBUF$. TOPEN accomplishes this interrupt transfer without using the select code queue of the processor. It therefore frees up the processor to do an output (which ties up the select code queue) while the TOPEN is handling input characters. The other consideration is that an interrupt type of output transfer is not allowed instead of the OUTPUT statement. This is because the TOPEN is already using the interrupt resources of the card for input. Therefore the only type of transfer that can take place for an output is one that uses the dedicated flag line of the interface card (i.e. OUTPUT). The other side of the coin would be trying to do an interrupt type of output transfer and a normal type of input transfer. There is no corresponding statement to TOPEN for an output operation. Therefore, for the same reasons that it is not possible todo interrupt type input transfers with normal type output transfers without TOPEN, it would not be possible to do interrupt type output transfers with normal type input transfers.

1 The TOPEN statement is not on the HP 9835 I/O ROM. Recognition of TOPEN can be put into the 9835 using a binary tape.

The following program gives an example of a simple program to output information to a computer connected to a System 45 via a 98036A Interface, and to print any information sent to the System 45 by the computer. The printing is done on a 9876A Printer set to select code 701, and the 98036A card is on select code 11.

```
10      DIM Input$[400],Send$[160]! THIS LINE DIMENSIONS STRINGS
20      CONTROL MASK 11;132        ! THIS SETS UP THE INTERRUPT MASK
30      WAIT READ 11,4;Dummy       ! COCKS THE CARD FOR A DUMMY READ
40      WRITE IO 11,7;0            ! TRIGGERS THE READ
50      CARD ENABLE 11             ! ENABLES THE CARD FOR INTERRUPT
60      ON KEY #0,3 GOSUB Send     ! ON KEY FOR SENDING DATA
70      TOPEN 11 GOSUB Receive     ! SETS UP THE TOPEN STATEMENT
80 Loop:  I=I+1                    ! SETS BACKGROUND LOOP
90        DISP I
100       GOTO Loop
110 Send:  INPUT "INPUT DATA THAT YOU WANT TO SEND",Send$
120   OUTPUT 11;Send$               ! OUTPUTS DATA TO OTHER MACHINE
130   RETURN
140 Receive:  Input$=Input$&TBUF$  ! DUMPS TBUF$ INTO INPUT STRING
160   IF NOT POS(Input$,CHR$(10)) THEN RETURN
161 ! IF THERE IS NOT A COMPLETE LINE THEN RETURN CHECKS THE
      POSITION OF THE FIRST LINE FEED
170 X=POS(Input$,CHR$(10))
180 PRINT Input$[1,X-1]    ! PRINTS THE LINE UP TO THAT POINT
190 Input$=Input$[X+1]     ! MOVES THE DATA TO THE BEGINNING OF THE
191                        ! STRING
200 GOTO 160               ! CHECK FOR ANOTHER LINE FEED
210 END
```

This program sets up two buffers called "Inputs$" and "Send$" to store received data and store data to be output, respectively.

The CONTROL MASK statement coupled with the CARD ENABLE statement enables the R5 register for an interrupt on a received character.

The WAIT READ statement sets the 98036A card busy so that an interrupt is not generated immediately. The WRITE IO statement then triggers the READ statement.

The ON KEY statement directs the program to a subroutine which requests input. The input is stored into the buffer "Send$" and then output to the 98036A card which in turn outputs it on the serial line.

When a character is received an end-of-line branch is logged in the system. When the current executing line is finished the machine checks to see if the software priority associated with TOPEN is greater than the current system software priority. If it is, the machine branches to the Receive subroutine. If it is not, then the machine holds off the end-of-line branch until the system software priority is lower than the priority of TOPEN. During this time TOPEN continues to take

incoming characters from the 98036A and put them into TBUF$. TBUF$ is a temporary 320 character circular buffer in machine memory. If the program cannot dump TBUF$ into a user buffer before 320 characters are received then for each character received above 320 one character at the front of the buffer is overwritten.

The Receive subroutine takes whatever is in TBUF$ and puts it into "Input$". (The character(s) must be put into a user buffer because any direct operation on TBUF$ clears that buffer.) The subroutine then checks for a linefeed. If no linefeed has been received the program returns to its background loop. If a linefeed has been received the line is printed up to the first linefeed, the data in "Input$" is then moved to the first character position of "Input$" and the program branches back to line 160 to check for another linefeed. If no other linefeed is present then the program returns to the background loop. If another linefeed is present then the line is processed as was done before.

Lines 80 to 100 are simply a background loop to represent a processing section of the program.

With a suitable BASIC program, it is possible for the System 45 to emulate an RS-232-C terminal. Such a program would cause the System 45 to mimic many of the operations of a terminal such as transmitting information entered through the keyboard to the computer and printing information received from the computer. It should be remembered, however, that HP desktop computers are designed to be stand-alone computational and controlling devices, and not primarily as terminal replacements. Also, it is not necessary to provide (via a high-level program) a complete terminal emulator in order to exchange data with another computer over an RS-232-C communications link. Within an applications program, data may be exchanged with a remote computer without any operator intervention (other than establishing the communications link if a modem is involved).

RS-232-C vs. Current Loop Operation

In the previous sections we have discussed two methods of interfacing for which standards exist that specify certain electrical, mechanical, and functional parameters. Most manufacturers of devices which are compatible with the IEEE-488-1978 (HP-IB) standard adhere closely to its definitions. And as a result most of these devices are plug-to-plug compatible. Unfortunately, this is not always true of devices that claim to be RS-232-C compatible. In particular, the fact that a serial I/O device uses an EIA 25-pin connector does not automatically make it an RS-232-C device.

Even when a device is RS-232-C compatible, variations from the expected configurations may still exist. We mentioned earlier that the common convention used for connectors is that the data terminal equipment (DTE) will use the male connector while the data communications equipment (DCE) will use the female connector. Indeed, the 98036A Interface is available in two configurations (see Figure 70) so that the desktop computer can play the part of either the DCE or the DTE. As an example of the difficulties that can arise, consider a terminal device (DTE) which terminates in a female 25-pin connector. Since the desktop computer is to act as a modem (DCE) in this system, the standard 98036A card will be used. But since both devices have female connectors, they cannot be plugged together. In this case, we will have to obtain a special female-to-female adapter in order to make the connection. But the fact that this adapter is necessary at all should alert the user to the possibility that all of the pin assignments for the terminal's connector may not be as expected. In particular, the transmitted data line (pin 2) and the received data line (pin 3) often need to be interchanged on one of the connectors or cross-wired in the adapter.

In general, when two RS-232-C devices are connected and do not appear to operate properly, there are two areas which should be thoroughly checked out before suspecting either a programming error or a hardware malfunction. The first is to insure that all of the pin assignments in both devices are as expected. For the 98036A Interface these pin assignments are found in Figure 73, while the assignments for the device being connected should be contained in the operating manual for that device. If these pin assignments are all correct, the user should then check to see if any control lines (Clear to Send, Data Set Ready, Carrier Detect, etc.) required by the device being interfaced are not set. The 98036A is capable of setting any of these control lines; but is the responsibility of the user to determine which lines his particular device requires and include the instructions for setting them within his program. For the 98036A card itself, these requirements are as follows: The standard 98036A (acting as a DCE device) requires the Data Terminal Ready (DTE, pin 20) signal to be true before it will transmit. Most terminals will automatically set this line when they are switched into the remote or on-line configuration. The Option 001 98036A (acting as a DTE device) requires the Clear to Send (CTS, pin 5) signal to be true for proper operation. If the DCE device being interfaced cannot supply this signal, it may be set on the 98036A card itself (see 98036A Installation and Service Manual).

The problem of connecting serial I/O devices is further complicated by the fact that a second data transmission convention known as "current loop" exists. As the name implies, this method of transmission uses the presence or absence of a current flow in the data line to represent the two logic states (1 or 0) rather than two different voltage levels as specified by the RS-232-C standard. It should be noted that a device operating in the current loop mode is not an RS-232-C device, even though it uses many of the same conventions such as start and stop bits, parity, etc., of the RS-232-C data format.

Historically, the first serial I/O devices operated in this current loop mode. With the advent of the RS-232-C standard, most manufacturers of serial I/O devices switched over to the use of voltage levels on the data lines. Even so, many current loop devices are still available. One reason for this is that while the practical operating distance for a direct (not using modems) RS-232-C data link is about 50 feet (15 meters), much longer distances are obtainable in the current loop mode.

Figure 74 shows the three basic elements that make up a current loop. The element labeled SOURCE is an active current supply. Typically, there is one and only one active source in any current loop, with all other elements acting passively.

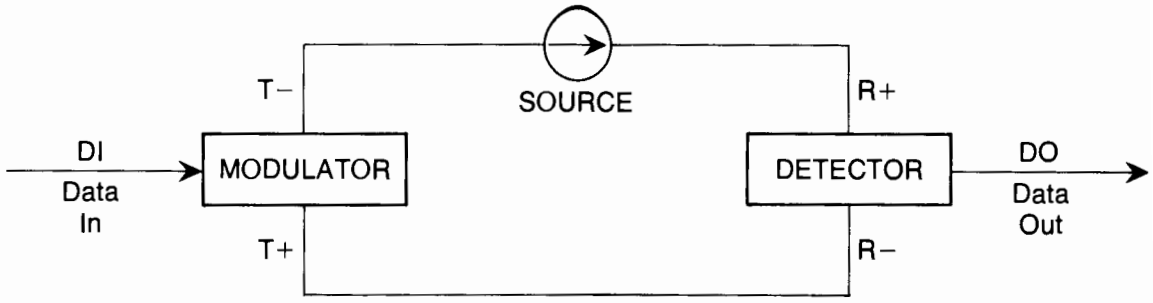


Figure 74

In the quiet (no data transmission) state, this source maintains a continuous current flow through the loop. When data is presented to the transmitter at the data-in line, the modulator converts this digital information (ones and zeros) into a matching sequence of current-on and current-off states by either allowing the current in the loop to flow, or blocking it. Since the

same current pulses flowing through the modulator also flow through the detector, the detector can sense these current on/off states and translate them back into digital information in a form (usually voltage levels) that can be recognized by the receiving device. The order in which each of these devices appear in the loop is not important.

In simplex operation, the source and modulator are located in the transmitting device, while the detector is located in the receiving device. For full-duplex operation, each device contains a transmitter (source plus modulator) and a receiver (detector). The transmitter of each device is connected to the receiver of the other device, thus creating two complete, independent current loops.

Most data terminals that operate in the current loop mode are passive devices; that is, they have a modulator and a detector, but not a current source. They depend instead on the DCE device to provide the current source. The 98036A card, when operating in the current loop mode, is capable of supplying the required current (20ma). Figure 75 shows how such a passive terminal should be connected to a 98036A for half-duplex current loop operation.

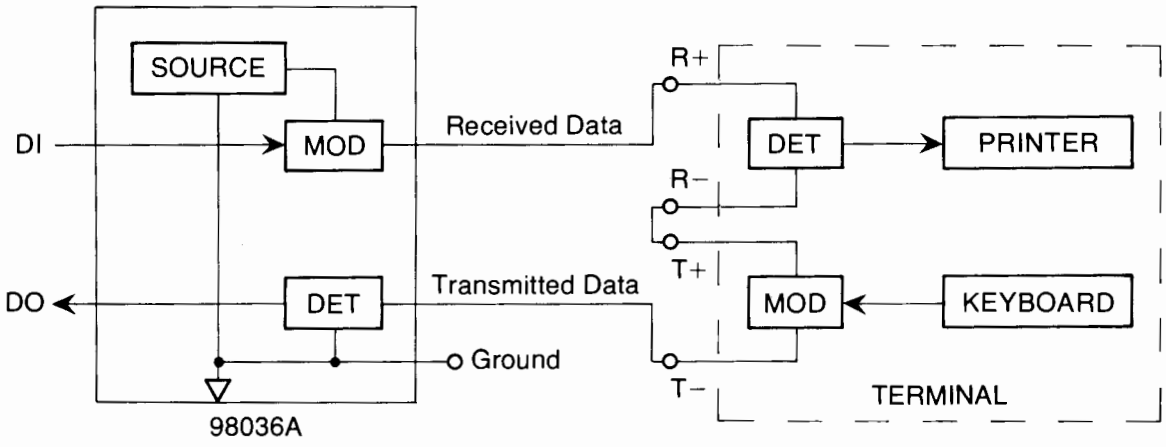


Figure 75

Several points should be noted about this connection. It is necessary to jumper (connect together) the R− and the T+ leads at the terminal in order to complete the loop. Also, in half-duplex current loop mode the ground lead from the 98036A is not connected. If the terminal also has an active current source, Figure 75 could be modified for full-duplex operation by disconnecting the R−/T+ jumper and reconnecting R− to ground and T+ to the terminal's current source.

In the half-duplex arrangement shown, there are actually five elements in the loop: a source, two modulators, and two detectors. Only one modulator at a time is allowed to modify the current flow (i.e., encode information to be sent around the loop), which is why this is a half-duplex arrangement. Both detectors, however, can be active at the same time. This also means that since the terminal's detector receives the information sent by the terminal's modulator, no special circuitry is required to get the effect of an echo-back.

In the figures above we have labeled the current loop connections at T+, T−, R+, and R−. Various manufacturers may use different designations to label these connections, such as T5, T6, T7, and T8. The exact labeling for a given device can be obtained from the operating manual for that device. Figure 76 shows three typical receiver (detector) circuits and may be helpful in recognizing the R+/R− connections from the schematic diagrams in the operating manual for the terminal being connected.

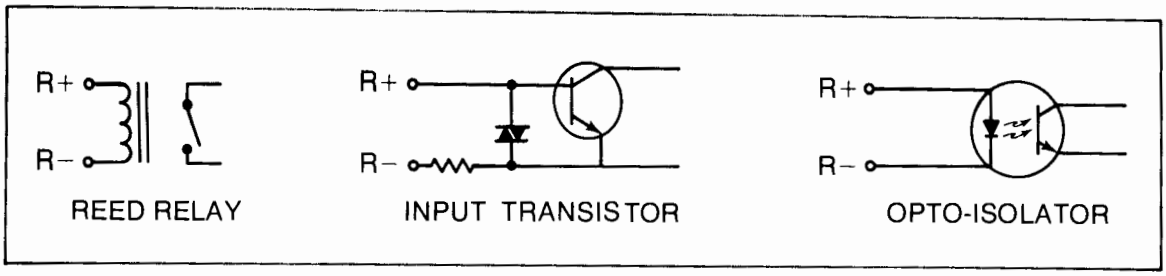


Figure 76

Up to now we have discussed current-loop operation in terms of presence or absence of current in the loop, without saying anything about the amount of current flowing. Most current sources for use in current loop operation provide either 20 milliamps (ma) or 60ma. The 98036A Interface card can supply (acting as a source) 20ma. Its receiver is capable of sinking (acting as a detector) either 20ma or 60ma with no modifications or switch settings required on the card to select 20ma or 60ma operation.

The current source in the 98036A operates from a 12 volt power supply, and the detector in the 98036A requires 6 volts to operate properly. This means that the other passive elements in the loop (terminal modulator and detectors) must not drop the voltage of the current passing through them more than 6 volts. Converting this voltage drop to the equivalent resistance at 20ma we have

$$R = V/I = (6v)/(20ma) = 300 \text{ ohms}$$

Thus, when the combined resistance of the external elements in the loop exceeds 300 ohms, the detector in the 98036A will no longer be able to reliably distinguish logical ones from zeros. This external resistance is made up not only of the elements in the loop, but also of the wire in the loop itself. And since the resistance of a wire is proportional to its length, this limit on external resistance is what ultimately determines the maximum operating distance of the serial I/O link in the current loop mode.

Appendix

ASCII Character Codes

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

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

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

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

HP-IB Universal Commands (ATN true)

Primary Command Group (PCG)

Decimal Value	ASCII Character	Interface Message	Description
0	NUL		
1	SOH	GTL	Go To Local
2	STX		
3	ETX		
4	EOT	SDC	Selected Device Clear
5	ENQ	PPC	Parallel Poll Configure
6	ACK		
7	BEL		
8	BS	GET	Group Execute Trigger
9	HT	TCT	Take Control
10	LF		
11	VT		
12	FF		
13	CR		
14	SO		
15	SI		
16	DLE		
17	DC1	LLO	Local Lockout
18	DC2		
19	DC3		
20	DC4	DCL	Device Clear
21	NAK	PPU	Parallel Poll Unconfigure
22	SYN		
23	ETB		
24	CAN	SPE	Serial Poll Enable
25	EM	SPD	Serial Poll Disable
26	SUB		
27	ESC		
28	FS		
29	GS		
30	RS		
31	US		
32-62	SP to > (Numbers, special char)	LAG	Listen Address Group
63	?	UNL	Unlisten
64-94	@ to ↑ (Upper case ASCII)	TAG	Talk Address Group
95	—	UNT	Untalk
96-126	(lowercase ASCII)	SCG	Secondary Command Group
127	DEL		

98032A Interface

Register Map

	IN	OUT
R4	DATA IN	DATA OUT
R5	STATUS	CONTROL
R6	HIGH BYTE DATA	HIGH BYTE DATA
R7	(not used)	TRIGGER

R4-IN: Read 16 bits (lower 8 bits if jumper B is not installed) of data from the input data latches. Sets I/O line to input.

R4-OUT: Write 16 bits (lower 8 bits if jumper F is not installed) of data to the output data latches. Sets I/O line to output.

R5-IN: Read 98032A card status byte.

R5 IN

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INT	DMA	1	0	IID	IOD	STI1	STI0

R5-OUT: Write 98032A card control byte.

R5 OUT

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INT	DMA	RESET	AH	—	—	CTL1	CTL0

R6-IN: Read 16 bits (upper 8 bits if jumper B is not installed) of data from the input data latches. Does not affect I/O line.

R6-OUT: Write 16 bits (upper 8 bits if jumper F is not installed) of data to the output data latches. Does not affect I/O line.

R7-OUT: Sets PCTL to initiate an input/output handshake, depending on the state of the I/O line from the last R4 access.

Jumper Options

Jumper	Function (when installed)
1	Indicates input data lines are positive-true.
2	Indicates output data lines are positive-true.
3	Inverts PCTL to high = set, low = clear.
4	Inverts PFLG to high = ready, low = busy.
5	Inverts PSTS to high = not OK, low = OK.
6	Set for pulse-mode handshake.
7	Required for DMA transfers.
[1] 8	Clock high input byte when PFLG goes from ready to busy.
[1] 9	Clock high input byte when PFLG goes from busy to ready.
[1] A	Clock high input byte on R6-IN operation.
B	Select words (16 bit) input mode.
[1] C	Clock low input byte on R4-IN operation.
[1] D	Clock low input byte when PFLG goes from busy to ready.
[1] E	Clock low input byte when PFLG goes from ready to busy.
F	Select words (16 bit) output mode.

1 Select only one of these three.

98033A Interface

Register Map

	IN	OUT
R4	DATA IN	(not used)
R5	STATUS	CONTROL
R6	(not used)	(not used)
R7	(not used)	TRIGGER

R4-IN: Read one 8-bit ASCII character from the 98033A BCD-to-ASCII translator.

R5-IN: Read 98033A card status byte.

R5 IN

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INT	0	1	0	0	0	0	0

R5-OUT: Write 98033A card control byte.

R5 OUT

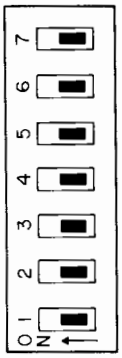
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INT	—	RESET	—	—	—	—	—

R7-OUT: An output to R7 (actual value output is a "don't care") causes the 98033A to place the next ASCII character in the sequence representing the reading into the R4-IN register. After 16 characters have been so placed, the next R7-OUT causes a new reading to be taken (i.e., the card sets CTLA and CTLB to start a data handshake with the BCD device) and places the first character of that reading in the R4-IN register.

Switch Configurations

Switch set to "ON", will:

- Invert DFLGA
- Invert DFLGB
- Select CTLA-2
- Select CTLB-2
- Invert CTLA
- Invert CTLB
- Select Optional Format



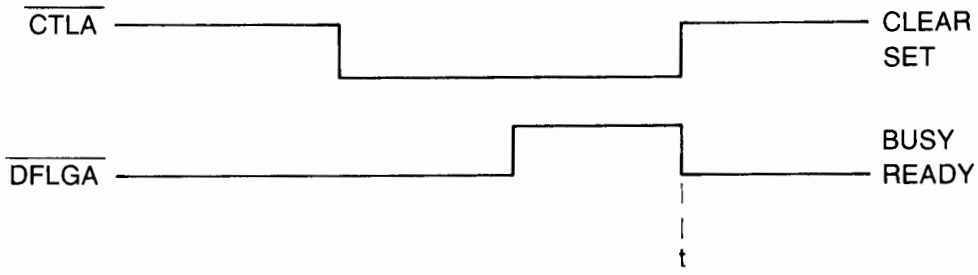
S2

- Invert SGN2
- Invert SGN1
- Invert OVLD
- Invert Data

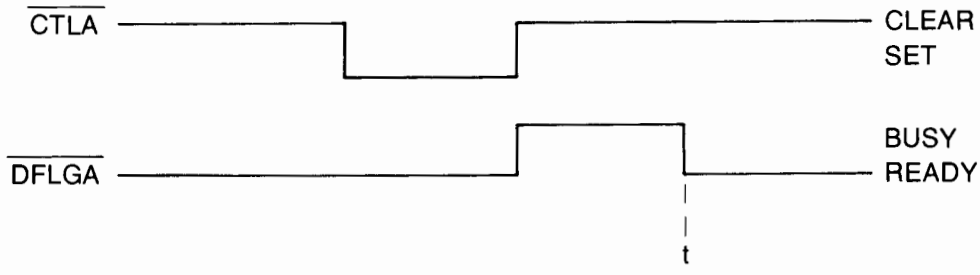
S3

Handshake Diagram

CTLA-1 MODE (Select CTLA-2 switch off).



CTLA-2 MODE (Select CTLA-2 switch on).



At time "t" the data on the BCD input lines is valid and the BCD-to-ASCII translation process begins. CTLB and DFLGB operate in a similar manner.

98034A Interface

Register Map

	ON	OUT
R4	DATA IN	DATA OUT
R5	STATUS	CONTROL
R6	STATUS/DATA	COMMANDS
R7	PARALLEL POLL	DIRECT BUS CONTROL

R4-IN: Initiates a data byte input sequence.

R4-OUT: Transfers one byte of data to the bus.

R5-IN: Initiates a status read sequence.

R5-OUT: Outputs a control byte to enable the 98034A for various interrupt conditions.

R5 OUT

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SRQ	ACT	TLK	LST	IRF	ORE	OTHER	EOI

R6-IN: Completes a data byte input sequence. Clears ATN.
Delivers 98034A status bytes.
Completes a parallel poll input sequence.

R6-OUT: Sets the ATN line true and outputs a byte of command or addressing information.

R7-IN: Initiates a parallel poll byte request.

R7-OUT: Direct[1] bus control.

1 After executing this R7-OUT instruction, the 98034A will clear the STS line if an illegal operation (e.g., specifying ATN if the 98034A is not active controller) is indicated.

R7 OUT, Bit 7 = 1

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	0	EOI	IFC	ATN	REN	SRQ

Service Request control and serial-poll response byte.

R7 OUT, Bit 7 = 0

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	SRQ	X	X	X	X	X	X

X = user definable.

98034A Interface

Operational Sequences

General

Any operation with the 98034A should be preceded by testing the flag (FLG) line and waiting for it to indicate ready. Otherwise, erroneous operation can result.

After a sequence of operations, the status (STS) line should be tested. It will be cleared if an illegal operation was specified, otherwise it will remain set.

Controller Talker Addressing (TAD)

This sequence addresses the 98034A as a talker, and one or more bus devices as listeners. When used in other operational sequences, it will be abbreviated as TAD.

1. R6-OUT Send 98034A talk address.
2. R6-OUT Send unlisten (63) command.
3. R6-OUT Send device listen address.
4. R6-OUT Send device secondary address if specified.
5. Repeat 3 and 4 for any multiple listeners.

Controller Listener Addressing (LAD)

1. R6-OUT Send unlisten (63) command.
2. R6-OUT Send 98034A listen address.
3. R6-OUT Send device talk address.
4. R6-OUT Send device secondary address if specified.
5. R6-OUT Send device listen address for multiple listener.
6. R6-OUT Send device secondary address if specified.
7. Repeat 5 and 6 for any other multiple listeners.

Data Output

1. TAD Address the bus.
2. R4-OUT Send the first data byte.
3. Repeat 2 for each data byte.

Data Output Using EOI

1. TAD Address the bus.
2. R4-OUT Send the first data byte.
3. Repeat 2 for each data byte.
4. R7-OUT Send a 144 to R7 to set EOI with REN false, or 146 to set EOI with REN true.
5. R4-OUT Send the last data byte. The 98034A will automatically clear EOI after the handshake is completed.

Data Input

1. LAD Address the bus.
2. R4-IN Start acceptor handshake (set NRFD false).
3. R6-IN Take in the received data byte.
4. Repeat 2 and 3 for each data byte.

By setting bit 0 of R5-OUT, the 98034A is enabled to clear STS if EOI is set. In this case the STS line would be tested after step 3.

Read Status

1. R5-IN: Initiate status read sequence. In the byte received, bits 4 and 5 are ones, indicating an HP-IB card type. No other bits are meaningful.
2. R6-IN: Get status byte 1.

STATUS BYTE 1

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	DCL	0	ERROR

3. R6-IN: Get status byte 2.

STATUS BYTE 2

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	1	0	A_5	A_4	A_3	A_2	A_1

4. R6-IN: Get status byte 3.

STATUS BYTE 3

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
EOI	REN	SRQ	ATN	IFC	NDAC	NRFD	DAV

5. R6-IN: Get status byte 4.

STATUS BYTE 4

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SRQ	ACT	TLK	LST	SAC	1	SPL	EOR

The 98034A is not monitoring the bus during this sequence. Thus, if the 98034A is not the controller, this sequence must be completed within 100 microseconds to satisfy IEEE-488 timing specifications. This sequence also resets the status (STS) line if it had been cleared by a previous illegal operation.

Serial Poll

1. LAD Address the bus.
2. R6-OUT Send SPE (24) command.
3. R4-IN Initiate a data input handshake.
4. R6-IN Take in the serial poll byte.
5. R6-OUT Send SPD (25) command.
6. R6-IN Optional dummy operation to clear ATN.

Parallel Poll

1. R7-OUT Send 148 to R7 to set ATN and EOI.
2. R7-IN Initiate parallel poll byte request.
3. R6-IN Take in the parallel poll byte.
4. R7-OUT Send 128 (or 130 if REN should be set) to R7 to clear ATN and EOI.

Passing Control

1. R6-OUT Send unlisten (63) command.
2. R6-OUT Send device talk address.
3. R6-OUT Send TCT (9) command.
4. R6-IN Clear ATN line to complete transfer of control.

98036A Interface

Register Map

	IN	OUT
R4	DATA IN, R4E	DATA OUT, R4C, R4D
R5	STATUS	CONTROL
R6	LINE STATUS	LINE CONTROL
R7	(not used)	TRIGGER

R4C Mode Word

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Number of Stop Bits 00=not valid 01=1 bit 10=1.5 bits 11=2 bits		Parity Type 0=Odd 1=Even	Parity Enable 0=Disable 1=Enable	Character Length 00=5 bits 01=6 bits 10=7 bits 11=8 bits		Bit Rate Factor 00=not used 01=1 X bit rate clock 10=1/16 X bit rate clock 11=1/64 X bit rate clock	

R4D USART Control Word

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Always 0	USART Reset	Clear To Send Pin 5 (Standard) Request To Send Pin 4 (Option 001)	Reset Status Bits of USART Status Word	Send Break Character	Enable Data Receiver	Data Set Ready Pin 6 (Standard) Data Terminal Ready Pin 20 (Option 001)	Data Enable Transmitter

R4E USART Status Word

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	Always 0	Framing Error	Overrun Error	Parity Error	Transmitter Empty	Receiver Ready	Transmitter Ready

R5 OUT Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Interface Interrupt Enable		Programmed Interface Reset			Interrupt Control 2 Receiver Control	Interrupt Control 2 Transmitter Control	R4 Control 0=Data IN/ OUT 1=Control/ Status

R5 IN Register

Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Peripheral Status 1 Mode	Interface Interrupt Enable Status	0	Interface I.D. 0	Interface I.D. 1	0	0	Control Status 2 Receiver	Control Status 1 Transmitter Mode

R6 OUT Register (standard cable)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
			Half/Full Speed Control (Interface)	Ring Indicator Pin 22	Signal Quality Detect Pin 21	Secondary Line Signal Detect Pin 12	Line Signal Detect Pin 8

R6 IN Register (standard cable)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Always 1	Always 1	Always 1	Always 1	Always 1	Always 0	Data Signal Rate Select Pin 23	Secondary Request To Send Pin 19

R6 OUT Register (Option 001 cable)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
			Half/Full Speed Control	Special Purpose Pin 25	Data Signal Rate Select (U.K.) Pin 11	Data Signal Rate Select Pin 23	Secondary Request To Send Pin 19

R6 IN Register (Option 001 cable)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Always 1	Always 1	Always 1	Always 1	Always 1	Secondary Line Signal Detect Pin 12	Ring Indicator Pin 22	Line Signal Detect Pin 8

9835 Keyboard/Display/Printer Registers

When the peripheral address is set to zero, the keyboard, display, and printer are addressed. In this case, the I/O registers have the following meanings.

	IN	OUT
R4	KEYBOARD KEYCODE	DISPLAY DATA
R5	STATUS	CONTROL
R6	(not used)	PRINTER DATA
R7	(not used)	CHARACTER SET SELECT

R4-IN: Returns an 8-bit keycode from the keyboard (bits 0-6) plus bit 7 indicating 1 = shift, 0 = no shift.

R4-OUT: Output one character to the rightmost position of the 32-character shift buffer for the display. Bit 7 indicates cursor on (= 1) or off (= 0) for this character (9835B only).

R5-IN:
bit 0: Always 1.
bit 1: Printer out-of-paper (= 1) indicator.
bit 2: Printer busy (= 1) or ready (= 0) indicator.
bit 3: Control key (= 0) if Control Key is pressed.
bit 4: Key flag (= 1) if a key has been pressed; cleared by R4 IN.

R5-OUT:
bit 0: Dump printer buffer to printer.
bit 1: Dump display buffer to single line display.
bit 2: Trigger beeper.
bit 3: Set busy light off.
bit 4: Set busy light on.
bit 5: Select insert cursor for display.
bit 6: Select replace cursor for display.
} 9835B only

R6-OUT: Output one character to the rightmost position of the printer shift buffer.

R7-OUT: bit 0: 0 = Select standard character set.
1 = Select alternate character set (if available).

9845 Keyboard and Printer Registers

When the peripheral address is set to zero, the keyboard, display, and printer are addressed. In this case, the I/O registers have the following meanings.

	IN	OUT
R4	(not used)	PRINTER DATA
R5	PRINTER STATUS	PRINTER CONTROL
R6	KEYBOARD KEYCODE	(not used)
R7	KEYBOARD STATUS	KEYBOARD CONTROL

R4-OUT: Outputs one 8-bit character to the next left justified position of the printer's 80 character buffer.

R5-IN:
- bit 0: Printer interrupt (=1).
- bit 1: Print head fault (=1).
- bit 5: Printer present (=1).
- bit 7: Printer interrupt mode is enabled (=1) or disabled (=0).

R5-OUT:
- bit 5: Immediate printer reset (=1).
- bit 7: Enable (=1) or disable (=0) printer interrupt mode.

R6-IN: Returns a 7-bit keycode from the keyboard (bits 0-6).

R7-IN:
- bit 0: Keyboard interrupted (=1).
- bit 12: PRINTALL key down (=1).
- bit 13: AUTOST key down (=1).
- bit 14: CONTROL key down (=1).
- bit 15: SHIFT key down (=1).
- Note: Bits 12-15 are updated only when the keyboard interrupts.

R7-OUT: Clear interrupt request (acknowledge interrupt) and BEEP if bit 15=1.

Bibliography

General

- System 35 I/O ROM Programming (HP #09835-90060)
- System 45 I/O ROM Programming (HP #09845-91060)
- Calculator Users Guide and Dictionary, Charles J. Sippl (Champaign, Illinois: Matrix Publishers, Inc.). A survey of calculator and desktop computer products, plus a glossary of hundreds of commonly used computer terms and concepts.

HP-IB

- IEEE Standard Digital Interface for Programmable Instrumentation. IEEE Std. 488-1978. (Institute of Electrical and Electronics Engineers, Inc., 345 E. 47th Street, New York, N.Y., 10017, USA). This is the complete and formalized description of the HP-IB, intended for use by an engineer designing a bus compatible instrument. It is not a good starting point for learning about the bus.
- Condensed Description of the Hewlett-Packard Interface Bus (HP #59401-90030). A reference guide to the HP-IB extracting the operational aspects from the IEEE Std. 488-1978.
- HP-IB Improving Measurements in Engineering and Manufacturing (HP #59300-90005). Operating characteristics for nineteen popular HP-IB instruments, along with sample 9825 programs for each instrument.

Serial I/O

- Guidebook to Data Communications (HP #5955-1715). An extensive survey of terms, concepts, and equipment used in data communications.

Notes

Notes

Subject Index

a

b

c

d

e

f

g

h

i

j

l

m

n

o

p

r

s

t

u

v

w

x

BASIC Language Interfacing Concepts

Part No. 09835-90600
Microfiche No. 09835-99600

Printed in U.S.A.
Third Edition (June 1981)