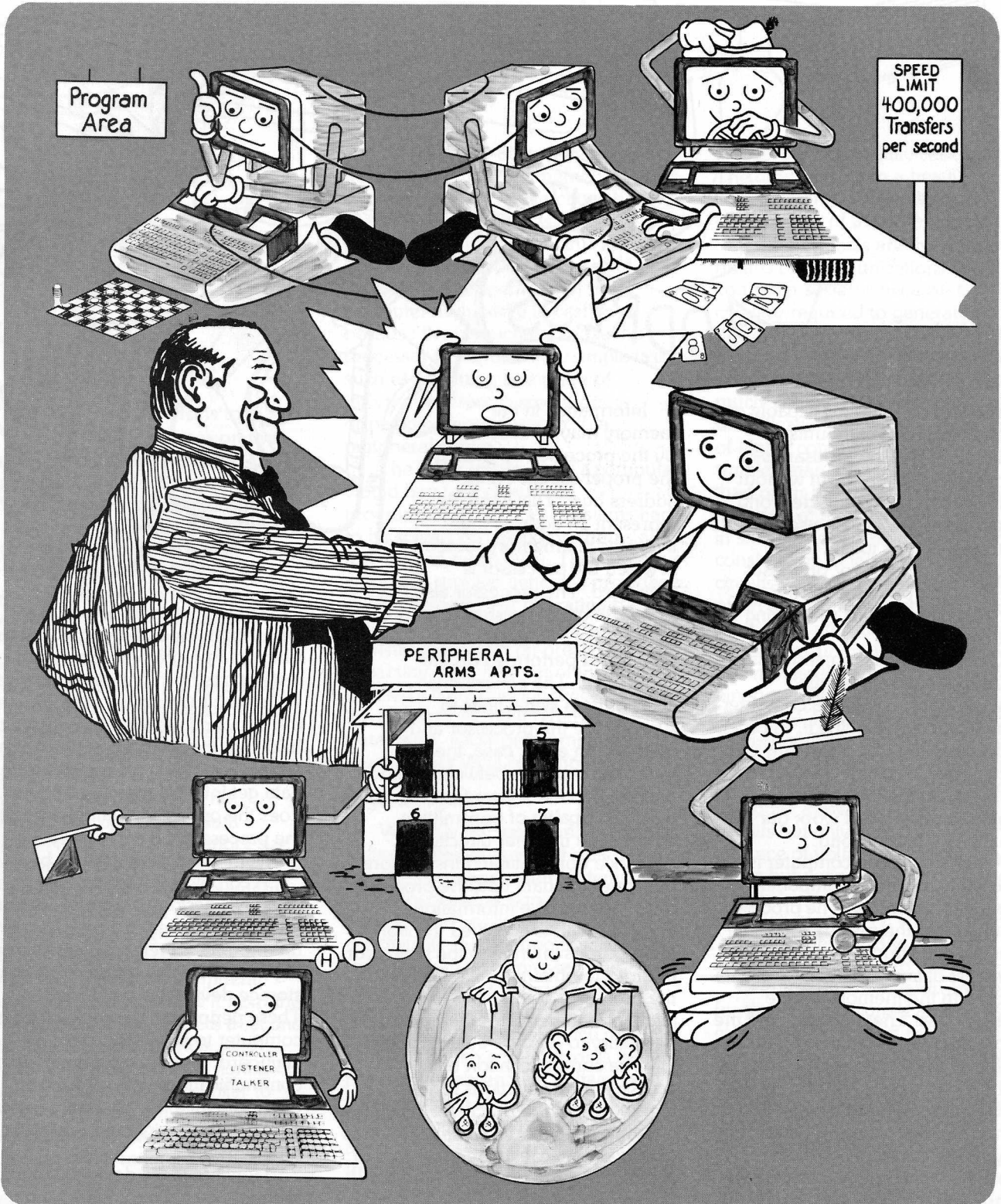
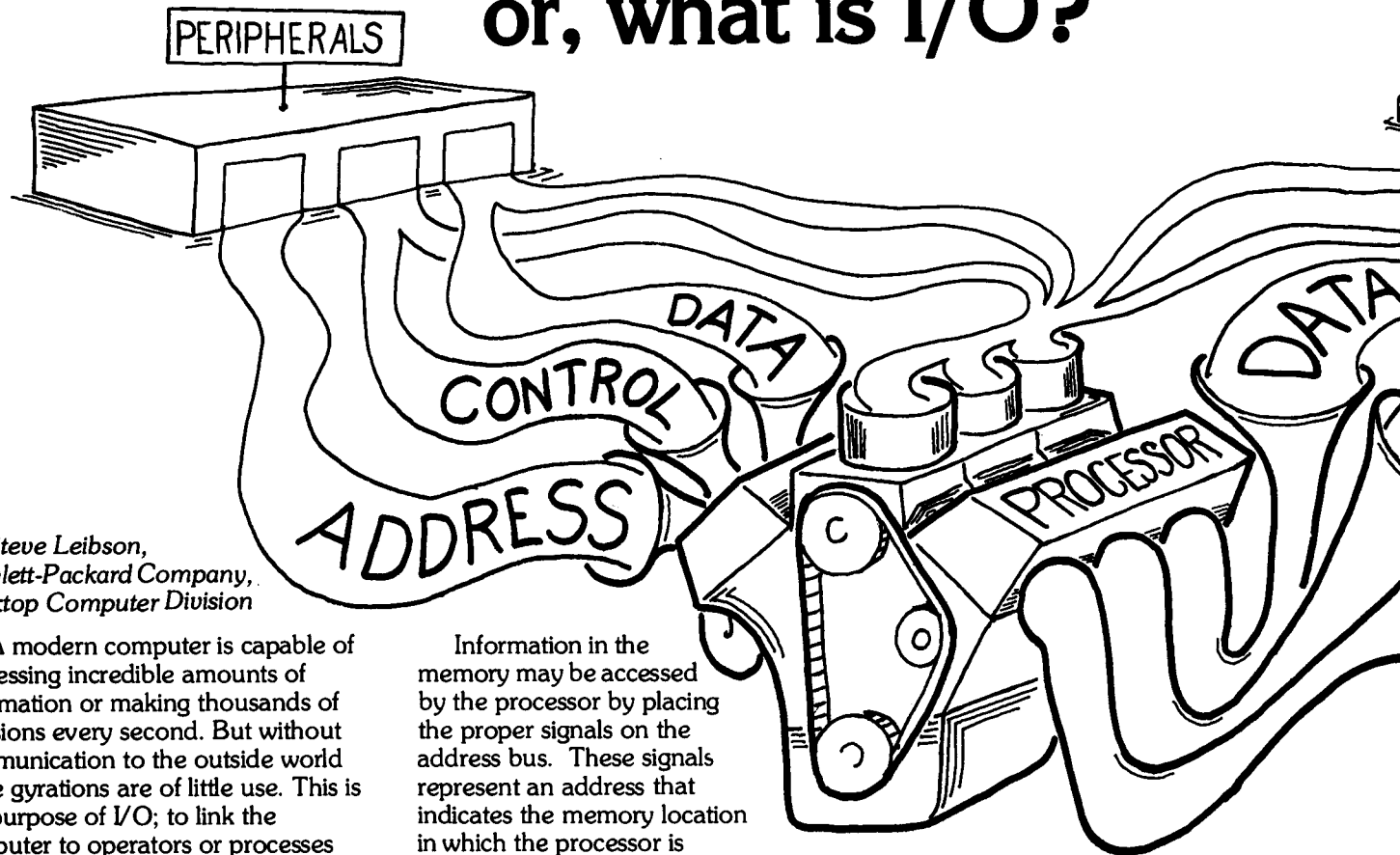


# Leibson on I/O

**A Publication of Hewlett-Packard Desktop Computer Division**



# How do computers communicate, or, what is I/O?



by Steve Leibson,  
Hewlett-Packard Company,  
Desktop Computer Division

A modern computer is capable of processing incredible amounts of information or making thousands of decisions every second. But without communication to the outside world these gyrations are of little use. This is the purpose of I/O; to link the computer to operators or processes that require the problem-solving power provided by data processing equipment.

I/O is an abbreviation, it stands for input/output and represents communications between a computer and the world surrounding it. In order to understand the various means used to effect these communications, we are going to start at the core of the system, the computer itself, and work our way out to the rest of the world.

A general purpose computer is composed of two main components: a processor and memory. The processor is the engine of the system, following sequences of instructions which cause it to process data. Instructions and data are stored in the memory for the processor's use. The processor and the memory are linked together by three sets of lines called busses; the address bus, the data bus and the control bus. The computer memory is organized into thousands of locations, each having its own unique address and capable of storing one piece of data or one instruction in a sequence. It is the processor's job to differentiate between instructions and data.

Information in the memory may be accessed by the processor by placing the proper signals on the address bus. These signals represent an address that indicates the memory location in which the processor is interested. The processor also must signify whether it wishes to extract information into this location. This signaling is performed on the control bus. The control bus also contains signal lines to synchronize the processor and memory. In either case, the information passes between memory and the processor over the data bus, which is capable of transmitting information in either direction.

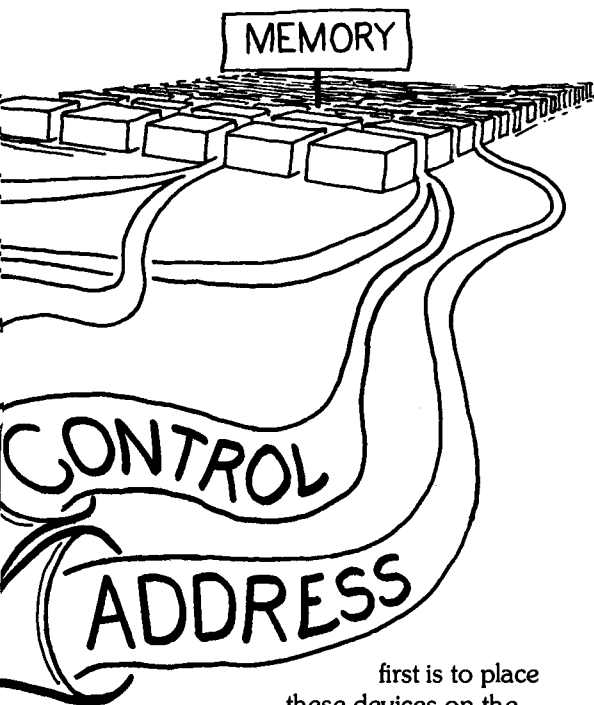
Since both data and instructions pass over the data bus, the processor must interpret the information correctly. This is achieved through timing cycles internal to the processor. In order to obtain its next instruction, the processor performs an *instruction fetch*. It then performs the operations necessary to execute the instruction.

The current location being accessed for instructions is held in a register within the processor called the *program counter*. The instruction thus obtained may cause the processor to again access memory, this time to obtain or to place data in the memory. Such operations are caused by executing *memory reference instructions*.

The computer is now able to perform the operations involved in running a program: it can obtain instructions from the memory, it can access the memory for data, process the data, and place this processed data back into the memory. A problem now arises; how do the program and the data get into the memory and how does the operator obtain the results of the processing? It is precisely this problem which is addressed by I/O.

A complete computer system, such as an HP desktop computer, is not merely composed of a processor and a memory. Peripheral devices such as a keyboard, display, printer and tape storage device are also included. These peripheral devices connect the computer to the outside world. The keyboard, display and printer allow communications with a human operator while the tape storage device provides for storing and retrieving programs.

How are these devices connected to the processor-memory combination residing inside of the computer? There are currently two methods in use. The



first is to place these devices on the memory bus already discussed. Thus the peripheral devices "appear" to the processor as memory locations. Data can be sent to or obtained from the peripherals using memory reference instructions. This configuration is called *memory-mapped I/O* because some portion of the computer memory has been allocated to peripheral devices.

The advantage of this system is that the existing processor instructions now serve the dual purpose of interfacing to memory and to I/O devices. The disadvantage is that the full range of the memory is not available for program and data storage. The maximum memory size of the computer has been reduced.

A second method of implementing I/O in a computer is to create a new bus, the I/O bus. The I/O bus is very similar to the memory bus. There is an address bus, called the peripheral address bus to differentiate it from the memory address bus, there is a second set of data lines and there is a peripheral control bus. The signals on these I/O busses may be similar to those of the memory bus or they may be very different. This system has the advantage of full memory capability at the expense of creating a new set of instructions for the processor called *I/O instructions*.

Let us briefly discuss instructions before going on. The memory reference and I/O instructions belong to a class of instructions called processor or machine instructions. This class of instructions is used for controlling the operation of the computer at the very lowest level because the instructions cause the processor to perform very simple tasks such as obtaining one piece of information from memory or dispatching one character to a peripheral device.

The typical operator of a computer would have a tremendous programming task if all problems had to be solved by writing programs at this level of complexity. Therefore the computer supplier generally provides a systems program or operating system which in effect implements a new set of instructions with far greater capability. This new set of instructions is called a *high-level language* because the instructions, now referred to as statements, allow programming on a much higher level of complexity.

### Digital Signals

We have discussed briefly the sets of lines called busses and stated that the processor and other systems components send signals along these busses. That implies that these busses are metallic carriers upon which voltages may be impressed and currents caused to flow, which is correct.

The simplest signal which might be sent along such a conductor is the presence or absence of voltage or current flow. This would then be a binary signal because it may only assume two states: present or absent. In the case of a voltage related signal, the voltage is either there or isn't: the voltage is either X volts or zero volts. Voltages are measured with reference

to a zero point, usually called ground. The ground is often a heavy conductor interconnecting all components of the computer system.

Binary signals are the primary means of communications in computer systems because the circuitry required to generate and detect mere signal presence or absence is much simpler to construct than circuits concerned with "how much" signal is present. This circuit simplification allows the construction of highly complex processors because simple binary circuits require less room than other types and therefore large numbers of them may be constructed in small spaces. This is the key to the construction of large scale integrated circuits which incorporate thousands of circuits on a small chip of silicon.

Busses are simply sets of parallel conductors upon which binary signals are impressed. The most common binary signal at present is the "TTL" level set. "TTL," which stands for Transistor-Transistor Logic, is the name of a family of integrated circuits that are used as the building blocks of computer systems. These digital circuits not only define the presence or absence of voltage as proper binary levels but define regions of voltage as proper levels. These regions are:

High region = 2 volts to 5 volts

Undefined = .8 to 2 volts

Low region = 0 to .8 volts

Thus we have a hardware system for transmitting signals as long as the circuits that send and receive the signals agree on the levels to be used. As we shall see later in this series, one of the tasks of I/O is to convert levels used by one portion of a system to those used in another portion. Unfortunately, not all peripheral devices use "TTL" levels. The computer busses that we discuss will all use these levels, however.

## Data Representations

Now that the signal levels have been established, an agreement must be made on what they represent. For instance what is the digital representation of an "A" or how is the number 123 represented? The alphabet from which the "A" was obtained can assume any of 26 values: "A" through "Z". Numbers may assume an infinite number of values. How can all of these values be represented with only two levels: on and off?

The answer is to use more than one signal line, to create a bus. If we were to use eight lines with each line able to assume one of two levels, then two to the eighth power or 256 values could be represented. This is sufficient to represent all of the characters in the alphabet, both upper and lower case, plus the other printed characters and punctuation marks on the typewriter, along with a few other characters.


With eight lines, values need only be assigned to each symbol to be represented, and as long as both the sender and receiver agree on what each value represents, communication can occur. Thus the second task of I/O is to assure agreement between sender and receiver, or at least to convert from one set of values to the other.

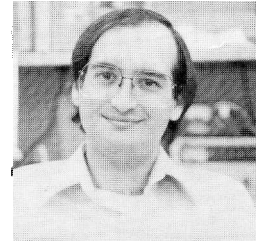
In addition, not all devices communicate on the same number of lines. Some use a single wire (plus ground) and send one binary "bit" of information at a time. The receiver assembles these sequential bits of information back into parallel representation. Some devices need only to send numerals, which may be represented with ten values requiring only four digital signal wires. Other forms of representation may require 16, 24, 32 or 64 lines making universal interconnection very difficult. The interfacing among these devices must

somehow adapt one system of representation to another for communication to be accomplished.

## Summary

It would simplify matters greatly if all devices could agree on data representation, format, signal levels, timings or even the number of wires to be used for interconnection. Attempts at such standardization have been made but due to the swift pace of technological development some standards are obsoleted before they are published. In addition, compromises always need to be made and different systems require different compromises. Older equipment also needs to be interconnected since the replacement of a computer should not mandate a complete system replacement.

Fortunately, present technology can reach backward as well as forward. The representational adaptations may be made by the computer itself; computers excel at changing one value into another. The hardware incompatibility can be overcome with interface circuitry which links the computer's memory or I/O bus to the I/O of the peripheral device. The techniques for accomplishing this interconnection are the topics to be covered in future I/O articles. 



Steve Leibson, lead engineer, received his BSEE from Case Western Reserve University in 1975. He has been with the Desktop Computer Division of HP for 3.5 years. His efforts include work on the 9878A I/O Expander, 98036A Serial Interface Card, 98224A Systems Programming ROM, and the System 45.

# The I/O bus

by Steve Leibson

Picture a void.  
Totally empty. Black.

Now place a computer processor into this void.

But without memory to hold program instructions and data, the processor is useless. So we will provide a memory and some wires to connect the processor to its memory. Our creation floats in the void quietly running its program, performing its assigned task in the scheme of things. Suddenly, it arrives at *THE* answer — but alas, we have given it no voice, no I/O with which it can announce the conclusion.

In the first installment of this series, we discussed several basic concepts relating to computer systems and I/O (Input/Output). We are now ready to provide the computer with a voice, some means of supplying the answers to the questions asked of it by some programmer.

## Bus is a set of conductors

The first order of business is to create an I/O bus leading from the processor to the outside. As we discussed last time, the I/O bus is a set of conductors carrying signals that represent the information which the computer is trying to transmit from the processor to the interface.

In addition, several conductors carry control signals that make it possible for the computer to signal the recipient at the other end of the bus when the data on the bus is valid and should be accepted. The recipient must also have some signals to communicate to the processor its readiness to accept data and its operational status. Finally, a signal is

needed to dictate the direction of data flow on the I/O bus since we want the computer to receive as well as transmit data.

Our figure of the data bus shows that it has a number of connections. The topmost connection represents a group of 16 data lines, and is shown with arrowheads at both ends. This is the peripheral data bus. It is capable of carrying data in either direction, depending on the immediate need. Under the data lines is a single wire called "strobe". This wire is a synchronizer and is used by the computer to signify that data is available.

## I/O wire is the traffic cop

The next line is called I/O and controls the direction of data flow. It is the traffic cop of the I/O bus, allowing bidirectional data flow, but only in one direction at a time. The recipient signals the computer on two wires called "status" and "flag". Status is a very simple signal, used to represent the presence or absence of the recipient. It is impossible to communicate with a device that isn't there.

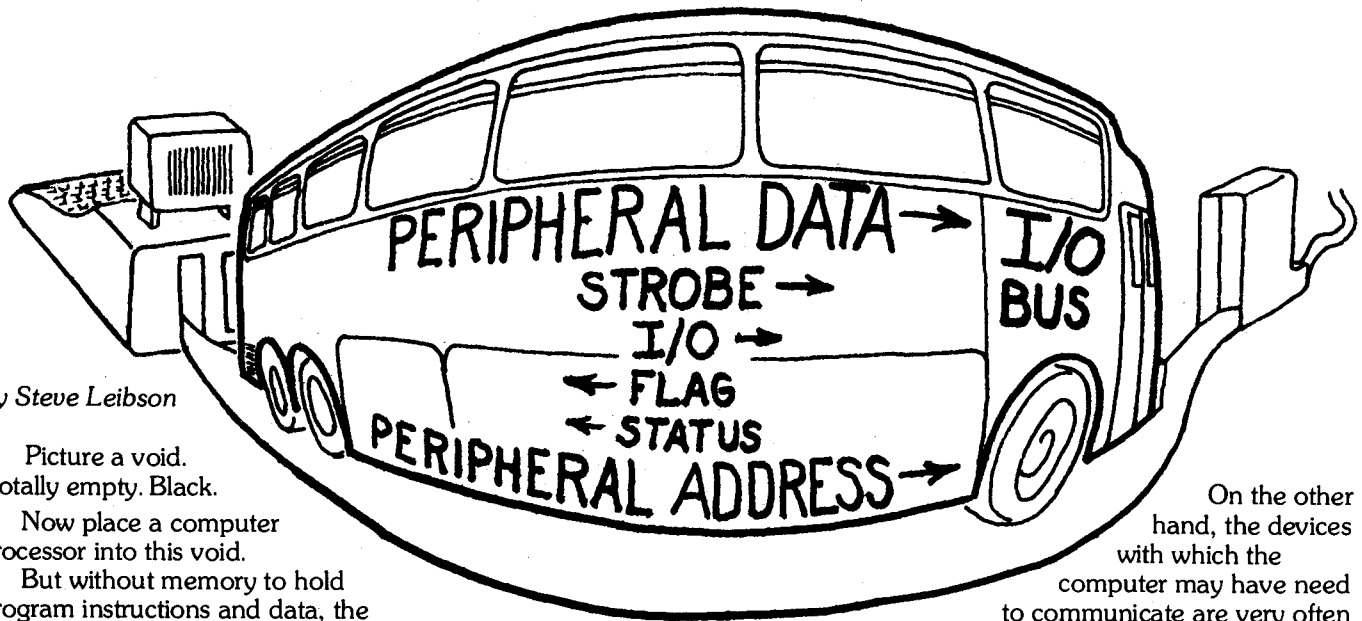
"Flag" is a more complex signal and to explain it requires a brief study of speed. In the scheme of things electronic, computer processors are very fast. The only moving parts inside a processor are the speedy electrons carrying the digital signals.

On the other hand, the devices with which the computer may have need to communicate are very often mechanical in nature. Disc and tape mechanisms, printers and plotters all have moving parts which take relatively long periods of time to perform their assigned tasks. Taking a printer as an example, let us examine the interchange between the computer and this peripheral device. The computer first addresses the printer using the last set of wires on the I/O bus, the "peripheral address" bus. If there is a device at the selected address, it will respond on the status line and inform the computer of its presence.

The computer will place the information it wishes to transmit on the data lines, set the I/O line to "output" (data direction is always from the computer's perspective) and finally cause the strobe line to clock the information into the printer. If the printer is operational, it will accept the information and print it.

A serial printer, similar in operation to a typewriter, would have to move to the next character position, select the proper character and mechanically fire some mechanism to strike the paper and leave a mark.

All these proceedings may take .01 of a second or so. That may not seem like too much time but processors generally execute an instruction in .000001 second. From the processor's perspective, the printer is taking forever. Fortunately, computers are





patient and will wait if told to do so. Our example computer will courteously wait for a signal on the flag line, which informs it that the printer has finished the task assigned.

### Processor interrupts

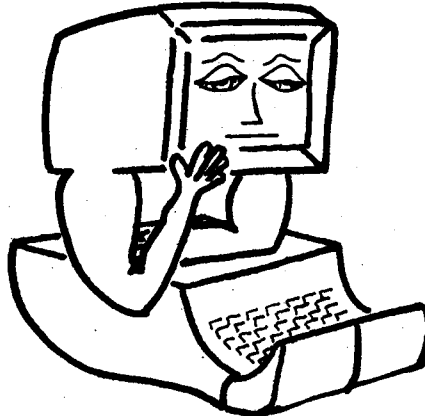
If it seems wasteful that such an expensive tool as a computer should have to wait most of the time on slower equipment, you are definitely thinking ahead. In a future article, we will explore a different communication mechanism known as interrupt. This feature allows the processor to go on about its business after transmitting a piece of information, later to be called back when the printer is ready for more.

We have already covered the peripheral address lines, though only briefly. Computers generally communicate to several peripheral devices. There are two ways to accomplish this. The first is to have a complete I/O bus for every peripheral device connected to the computer. Such a scheme would rapidly create a rat's nest of wires resulting in a totally unproduceable system.

Our I/O bus has a set of lines called peripheral address lines that are used to specify the device in which the computer is currently interested. This greatly simplifies wiring the system together and results in major cost savings. It does limit the computer to communicating with only one peripheral device at a time, but for most computer processors that is the limit anyway.

### Peripheral lines for multiplexing

The peripheral address lines allow the I/O bus lines to be shared or "multiplexed" by many devices. Each device must have its own unique address or conflicts will arise when two devices try to use the information at the same time. For instance plotters are very useful for graphing data but



Our example computer will courteously wait for a signal on the flag line.

are terrible program storage devices. Therefore, when accessing the disc storage device, the computer would prefer that the plotter ignored the transactions on the I/O bus.

Clearly each peripheral device must have a unique address. But it is more advantageous for each to have several unique addresses. Think of the peripheral device as an apartment building having a unique street address. Apartment one gets the daily newspaper, mostly general information, while apartment two gets the *Wall Street Journal* which reports information relevant to the economic running of the country. Both apartments receive information, but of differing types.

A peripheral device must also receive varying types of information. A printer not only has characters to print, but also information relating to the running of the printer such as: line spacing, number of characters to print per line and print font to name a few. We therefore create subaddresses within the peripheral device so that information of different types can be directed to the relevant section of the peripheral.

The peripheral address lines are split into a "select code" to specify the

peripheral's address and a "register code" to specify the subaddress. A "register" is a location which will temporarily hold information until it can be used by the peripheral.

### Setting up subaddresses

For our purposes, it may be sufficient to create four subaddresses within each select code. And being obstinate as all computer designers are, we will call these addresses 4,5,6 and 7 because nobody likes to start at zero. These are four registers which serve as portals to the peripheral device.

Four registers require two lines to specify the register address, since two lines can take on four states. These states are:

State of line #1	State of line #2	Register Addressed
0	0	4
1	0	5
0	1	6
1	1	7

(Remember that digital signal lines can take on only two states: on and off or "1" and "0".)

Although we have four registers, we still have a complication. The data lines which carry information to and from these registers are bidirectional and therefore the registers must be also. Actually, what we must have is a total of eight registers.

Four input registers contain information for the computer to input and four output registers receive information from the computer. We will select between these two sets of registers based on the state of the I/O signal line. Remember that the I/O line was used to specify the direction of data flow over the data lines and is ideal for selecting between the input and output register sets.

We have now completed construction of a simple I/O bus which can be used to convey information

between the computer and external devices. It isn't the most advanced I/O bus but it will satisfy our present needs. We will be upgrading this "bunch of wires" in the future but first there is a more pressing problem to solve.

### Introduction to interfaces

Our I/O bus happens to be a subset of the I/O bus used in the Hewlett-Packard Desktop Computers 9825A, System 35 and System 45. It would be most convenient if all peripherals were available with circuitry installed in them that directly interfaced to our I/O bus.

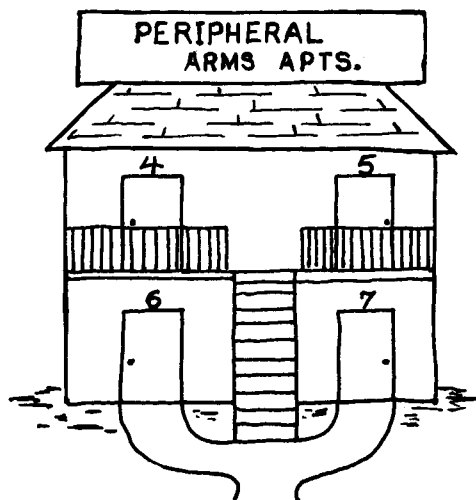
Unfortunately, the reality of the situation is far from the ideal. Our bus is parallel-oriented, meaning every binary digit (bit) of a piece of information such as a character is available simultaneously on the 16 data lines.

Not all peripherals use 16 bits of parallel data, some do not have parallel data lines at all but send and receive one bit of data at a time in a serial fashion. No peripherals use the 8 register scheme exactly as discussed above and some do not even use the same voltage levels to represent 0 and 1. This can present quite a problem to the person attempting to interface a computer to a peripheral.

### Interfaces act as translators

It is necessary to interpose some specialized circuitry between the I/O bus and the peripheral device to adapt the signals from one to those of the other. This specialized circuitry is called an interface. The interface is the actual recipient of the I/O bus. The interface acts as an intermediary, translating between the two interfaced devices.

If every peripheral manufactured in the last decade required a different interface, it would be impossible for a computer system to communicate with even a small fraction of the range of



Think of the peripheral device as an apartment building having four unique subaddresses.

devices available. Fortunately, a large number of devices can be interfaced using only four basic types of interfaces: parallel I/O, serial I/O, HP-IB and BCD.

The parallel interface connects to the peripheral with a set of wires very similar to those in our I/O bus, less the address lines. This interface is the most common among current peripherals. Major variations involve the physical connector used, and the sense of the control and data lines (Does zero volts mean a 0 or a 1?). A flexible parallel interface is available with several connectors as well as with an unterminated cable so that a custom connector may be installed. It is adjustable as to logic senses used and even logic levels used.

The serial interface takes the data from the I/O bus and serializes it into a stream of bits. Incoming serial data is converted to parallel data and sent to the computer. One type of connector is usually encountered, though not always. Many specialized control lines exist in this type of interface because serial I/O is found in the specialized data communications environment

where special channels are used for long distance communications.

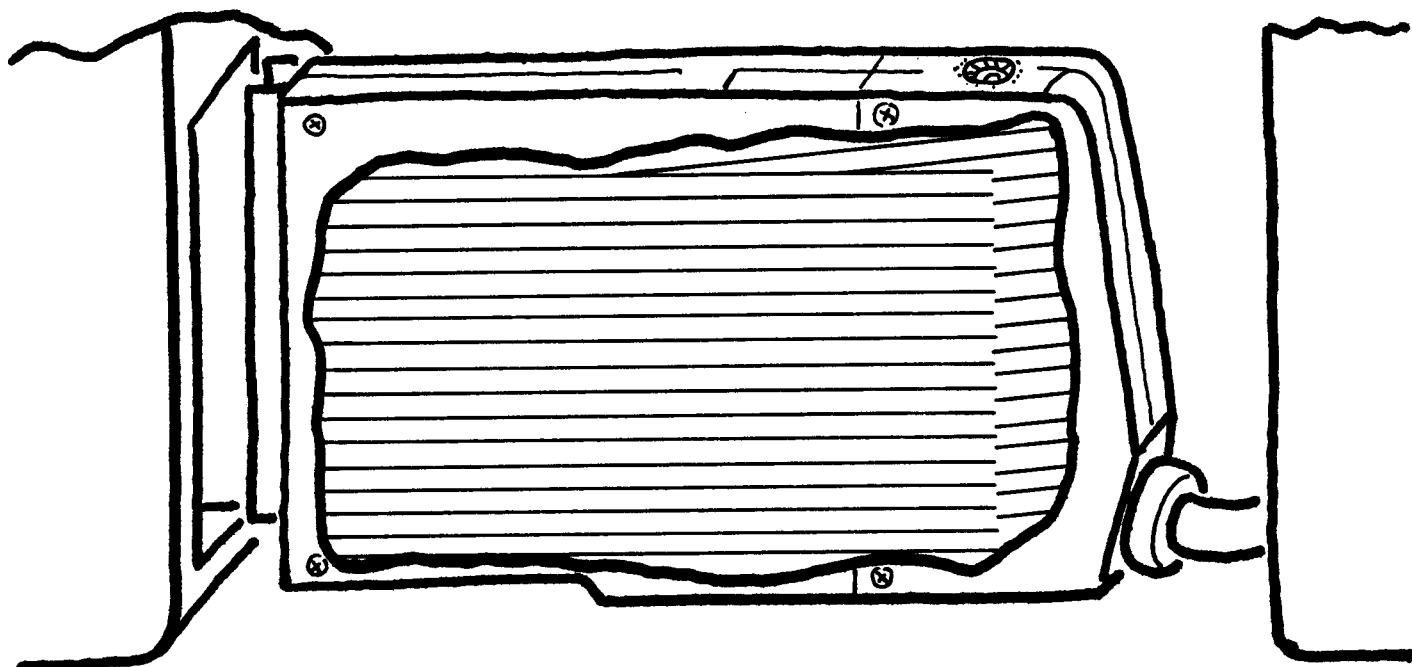
HP-IB is a relatively new interface standard. It is formally known as IEEE 488-1975 and the signals, connector, logic levels and logic senses are well defined. This interface allows connection to a large number of devices in a very simple manner; the cable connector is simply bolted to the connector on the peripheral device.

The hardware connection is not in question and the software for communication can be directly addressed. As a bonus, one interface can be used to service up to 14 peripheral devices. The HP-IB (Hewlett-Packard Interface Bus) is also known as the General Purpose Interface Bus (GPIB).

Older instruments use a different type of interface known as BCD. Data is dealt with four bits at a time to form numerals. This interface is used predominantly by instrumentation where the data to be transferred is only numeric.

In future articles, we will be examining these various interfaces in depth. We will find that in order to interface many devices to our I/O bus, we will be fighting problems of lack of information, faulty information and worst of all — faulty assumptions. In the process of this examination, we will develop the means for overcoming these problems and finally provide that "computer in a void" with a voice. **K**

# The parallel interface



by Steve Leibson, Hewlett-Packard Company, Desktop Computer Division

Computers are information processing machines and thus require paths for raw data to enter and for processed information to exit. In modern computer design, a very common technique is to create one universal path that leads both into and out of the processor. That path is the I/O bus.

This concept simplifies computer design but brings a complication: whatever the design of the I/O bus, the computer will be incompatible with a large number of peripheral devices. Some will be too old and use different signal levels, some will have varying data formats and most will be slow enough to seriously degrade the computer's performance if it must wait on every data transaction.

The complication is solved through the use of interfaces which act as transformers of voltage levels, data formats and transaction speeds, thus allowing a computer to communicate with a vast array of peripheral devices.

## Data lines in parallel

A very simple peripheral will often have interface requirements which are

very similar to the I/O BUS. Data is transferred over a set of data lines using a signal line to indicate when the next chunk of information is ready. The peripheral indicates its readiness to accept another piece of data on another signal line.

This type of interface is a parallel interface, so-named because the data lines are in parallel and data is transferred several bits at a time. The HP 98032A is a parallel interface designed for the 9825A, System 35A/B, and System 45A/B desktop computers. The I/O bus described in the Mar-Apr, 1979 issue of *Keyboard* is the I/O bus of the above-named computers, so we will be using the 98032A 16-bit Interface as the model for our discussion of parallel interfaces.

The I/O bus has 16 bidirectional data lines. Data is handled in 16-bit chunks and flows over these lines either into or out of the computer, but not in both directions at the same time.

The 98032A splits the I/O bus into two sets of data lines: 16 output lines and 16 input lines. The configuration is more compatible with unidirectional peripherals. Unneeded lines are left unconnected. Out of 32 data lines only eight might be used by a unidirectional, eight-bit peripheral.

As mentioned above, interfaces are sometimes used to transform

voltage levels used on a computer I/O bus to those required by a peripheral. Our I/O bus uses "TTL" levels meaning that a low level is represented by a voltage between 0 and 0.7 volts and a high level is between 2.0 and 5.5 volts. Since the input lines of the 98032A parallel interface are built using TTL integrated circuit logic, these voltages are required on the data inputs of the interface to represent high or low logic levels.

The data outputs have been built with transistors, however, and can withstand 30 volts for a high level. The low level is still around zero volts. Remember when discussing logic signal lines that only two signal levels are allowed, one designated high and the other low. If the high level corresponds to a logic one, the signals are said to be positive-true logic. If a low level signal corresponds to a logic one, it is called negative-true logic. Logic zero would correspond to low or high levels respectively.

We have now established:

1. The I/O bus data lines, which are the conductors used to transfer data between the computer and the interface.
2. The interface input and output lines, which are the conductors used to transfer data between the interface and the peripheral.



*Handshake lines serve to synchronize the interface and the peripheral. The meaning of each line depends on the direction in which data is flowing.*

## Register architecture

In the previous article, we established that each interface would have a unique address on the I/O bus and would be selected via the peripheral address lines of the I/O bus. Each interface was further subdivided into registers which could be individually addressed by a register code. The register model contains eight registers. Four of the registers are output registers, receiving data from the computer. The remaining four registers are input registers, supplying data to the computer. Each register has a special function which is defined by the interface. The 98032A Interface makes the following definitions:

### Input registers

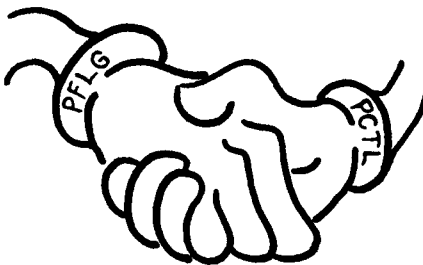
Register code	Function
R4	Data input
R5	Interface status
R6	High byte data input
R7	(Not used)

### Output registers

Register code	Function
R4	Data output
R5	Interface control
R6	High byte data output
R7	Data transfer trigger

The R4 registers are the primary means of data transfer in the interface. The R4 OUT register is directly connected to the interface output data lines and the R4 IN register is connected to the interface input data lines. When the computer places information into the R4 OUT register of a 98032 interface, the data pattern appears on the data output lines. A reading of the R4 IN register provides an image of what is in the interface's R4 input register which may or may not represent the current state of the interface's input data lines.

Note that the R7 OUT register is called the data transfer trigger. When used in conjunction with the R4



The peripheral handshake

registers, the R7 OUT register forms a handshake mechanism which effects data transfer between the fast computer and the slower peripheral.

Before discussing data handshake however, four more registers need to be discussed. The R5 IN register contains several pieces of important information. Only the lower eight bits of this register have been implemented. The meanings of these numbered bits are as follows:

Interface status (R5 IN) register							
7	6	5	4	3	2	1	0
INT	DMA	1	0	IID	IOD	STI1	STI0

The INT and DMA bits are used for I/O operations called interrupt and direct memory access respectively. These are advanced I/O techniques and will be discussed later in this series.

Bits five and four are interface identification bits. The 10 pattern identifies the 98032A interface as a type two interface, 10 being two in binary notation. Software in the computer uses the interface identity to decide how to communicate with the interface.

The IID and IOD bits are also used by the computer software. IID stands for invert input data while IOD stands for invert output data. These bits allow the computer-interface combination to communicate with peripherals using either positive-true or negative-true logic on the data lines.

It is important to note that the data inversion occurs in the computer and

not in the interface, and that the computer may choose to ignore these bits in certain classes of I/O operations.

## Another two bits

The two remaining bits of the interface status register, STI1 and STI0 are directly connected to two input lines. These two lines are general purpose and can be used for any user-defined function.

Interface control is effected through the R5 OUT register. The bit pattern for this register is as follows:

### Interface control (R5 OUT) register

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

The INT and DMA bits are used in the interrupt and direct memory access modes mentioned earlier. The RESET bit is used to place the interface in the initial, power-on state. In addition, when the RESET bit is set, a reset signal is sent out on one of the 98032A peripheral lines.

The AHS bit is the auto handshake bit. When this bit is set, the R7 DATA TRANSFER TRIGGER is not needed. This mode is useful for higher speed operations and is usually used by the internal software only. Bits three and two are unused.

The CTL1 and CTL0 bits are directly connected to output lines. These lines are separate from the output data lines and may be used to control the peripheral device. Such an application might be to latch the door of a printer while it is printing.

The R6 registers are data registers similar to the R4 registers. When the 98032 is set to the "byte" mode, the upper eight of the 16 data lines, in both directions, are allocated to the R6 registers instead of the R4 registers. This splitting of the two sets of 16 data

lines into four sets of eight data lines is useful for some peripheral devices with unusual protocols.

### Peripheral handshaking

Placing data on the output lines or reading the levels of the input lines connecting the interface to the peripheral device is not sufficient for smooth data flow. A set of signals signifying "new data ready" and "ready for new data" is needed.

These two lines are called handshake lines and serve to synchronize the interface and the peripheral. Each controls one line, but the meaning of the line depends on the direction in which data is flowing. We will call the line controlled by the interface PCTL for peripheral control and the line controlled by the peripheral PFLG for peripheral flag. We now have enough connections between the interface and the peripheral to discuss handshaking.

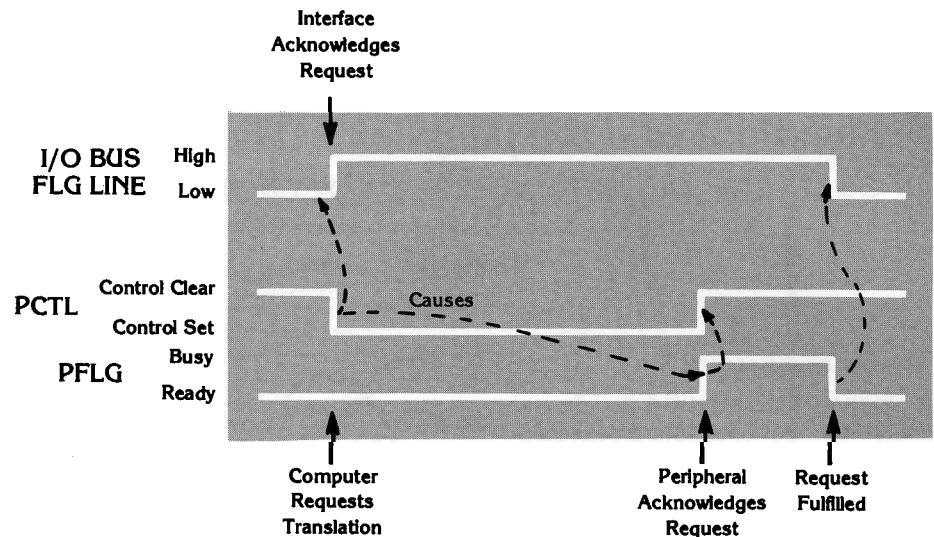
### Data output

Output is the simpler of the two data transactions. As mentioned before, the computer may place information in the R4 OUT register, setting the interface data output lines. It may then perform an R7 OUT operation, starting the handshake mechanism.

The interface recognizes the R7 OUT operation and causes the PCTL signal line to change from the clear state to the set state. The transition is a signal that "new data is ready" and that the peripheral should accept this new data.

The peripheral responds by changing the PFLG line from ready to busy, signifying that the data has been recognized and is being processed by the peripheral.

From the time that the computer performs the R7 OUT operation to the time that the peripheral returns to the ready state after processing the



Handshaking between the computer and the interface, and between the interface and the peripheral involves the same basic sequence of events. The diagram shows related changes on three key lines.

information, the interface is busy transferring the information placed in its R4 OUT register.

It is extremely important that the computer not access the R4 OUT register before the transaction has been completed. For this reason, the interface and the computer have a handshake mechanism also. While the interface is busy with a transaction, it will indicate this situation to the computer on the interface flag line.

### Data input

Data input from a peripheral is a slightly more complex operation because it is a three-step transaction. The computer first performs an R4 IN operation, reading the interface's R4 IN register. This operation is a dummy, and the information obtained is discarded because the interface did not have time to request a piece of information from the peripheral. The initial R4 IN serves to place the interface into the input mode and to set the peripheral I/O line to input.

The computer then performs an R7 OUT operation. As in the data output transaction, this causes the interface to set the PCTL line, signifying to the peripheral device that a piece of data has been requested.

The peripheral signals that it is busy and performs whatever operations are necessary to obtain the requested data. It then places the information on the data input lines and sets the PFLG line to ready.


When the peripheral returns to the ready state, the transition on the PFLG line holds, or "latches" the states of

the data input lines into the R4 IN register and causes the interface to signal ready on the flag line. The computer, which has been waiting for the interface to signal ready, now performs a second R4 IN operation. The transaction is complete.

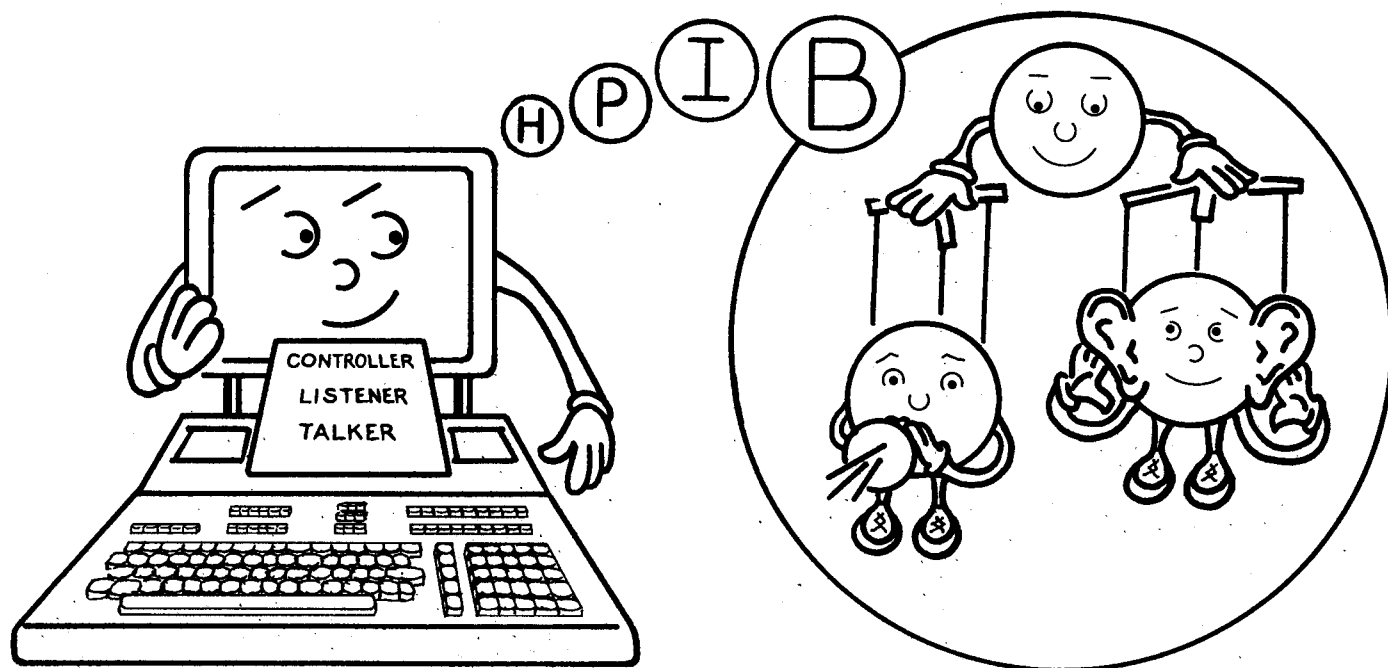
### "Latched" lines

There may be times when peripherals are so simple that they are incapable of performing the two-wire handshake. In these instances the PFLG and PCTL lines may be tied together so that the interface handshakes with itself. This results in 16 "latched" output lines and 16 input lines which may be read on demand.

Most peripherals using the parallel interface use only eight bits. This allows two raised to the eighth power or 256 combinations. If these combinations are treated as character codes, then numerals, upper and lower case letters, punctuation marks and other characters can be represented.

There are notable exceptions however. The 9885 Flexible Disc Drive uses the full 16 bits because Hewlett-Packard desktop computers are organized as 16-bit machines. The 9862A Plotter uses 12-bit instructions. Many analog-to-digital converters come in 10-, 12- or 16-bit sizes. All of these applications are served by the single 98032 16-bit Interface. 

# The standard interface



by Steve Leibson,  
Hewlett-Packard Company,  
Desktop Computer Division

Computer designers constantly strive to implement the latest parts and the fastest logic in new and different configurations. This characteristic has created a volatile situation in the computer industry. Additionally, designers of peripherals to be connected to the computers are creating entirely new classes of devices.

The end result is a multitude of interfaces, all of which have been optimized for the instrument to which they belong. But very few of them are compatible with each other.

## Adopting a standard

The situation is similar to the American railroads of the early 1800s. Dozens of track gages existed and cars of one railroad could not travel on the tracks of another. Just as the railroads

quickly standardized the track gages, the computer industry agreed on an interfacing standard that the Institute of Electrical and Electronic Engineers (IEEE) subsequently published in 1975.

It was the first comprehensive, nearly universal interfacing standard for computers and instrumentation. That first version, IEEE Standard Digital Interface for Programmable Instrumentation, or IEEE Std 488-1975, was revised in 1978, and now is called IEEE Std 488-1978.

This standard defines a general-purpose interface, designed for instrumentation systems requiring limited-distance communications. The intent of IEEE 488-1978 is to pin down as many variables in an interface as possible without defining the actual use of the interface.

In addition, the interface is defined without reference to the hardware circuitry required to implement the interface. This allows newer products to take advantage of newer

technologies, permitting faster, less expensive construction of devices and systems.

In the previous article in this series, we discussed the parallel interface. That interface had 16 input and 16 output lines, to make possible interfacing to as many different devices as possible. A very popular version of the parallel interface has no connector at the end of the cable. The system builder must add the appropriate connector for his peripheral, since there is no standard for either the connectors or how the pins in the connector are to be used.

Connector and pin usage are precisely specified in IEEE standard 488, as are signal levels, both voltage and current, and signal timings. Thus a system becomes a "remove-from-the-box-and-plug-together" operation. The hardware of interconnection is defined so that two interconnected instruments can communicate, although understanding is not guaranteed by the standard.

## HP's enhanced version

Hewlett-Packard has its own implementation of the IEEE standard, called the Hewlett-Packard Interface Bus, or HP-IB. HP-IB is a combination of the hardware specified by the IEEE standard plus a communication technique that makes it possible for instruments to understand each other, and for the designer to understand what is happening in the system.

The standard is so general that almost any instrument or peripheral can be purchased in an HP-IB version. Voltmeters, power supplies, printers and plotters are only a few of the available devices. All may be connected together on the same bus.

Unlike the parallel interface, which connects a single device with an HP desktop computer, the HP-IB interface makes it possible to connect to as many as 15 devices. HP-IB is indeed a bus, similar in concept to the I/O bus that the interface cards plug into on the computer.

## Controlling, talking, and listening

On the I/O bus, only two entities reside; the computer and the interface. The computer always controls the I/O bus, and the interfaces react to commands from the computer.

Three types of devices exist on the HP-IB; controllers, talkers and listeners. These entities are actually attributes, and may exist alone or in combination within any given device. For example, the HP-IB interface allows a desktop computer to be a talker, a listener and a controller. A voltmeter might be only a talker, only able to supply data to the system, while a printer may be only a listener, only able to accept data from the system. Additionally, these functions may be active or not at any given time.

Figure 1 illustrates how an HP-IB system might work. The lines on the right of the figure represent the signal

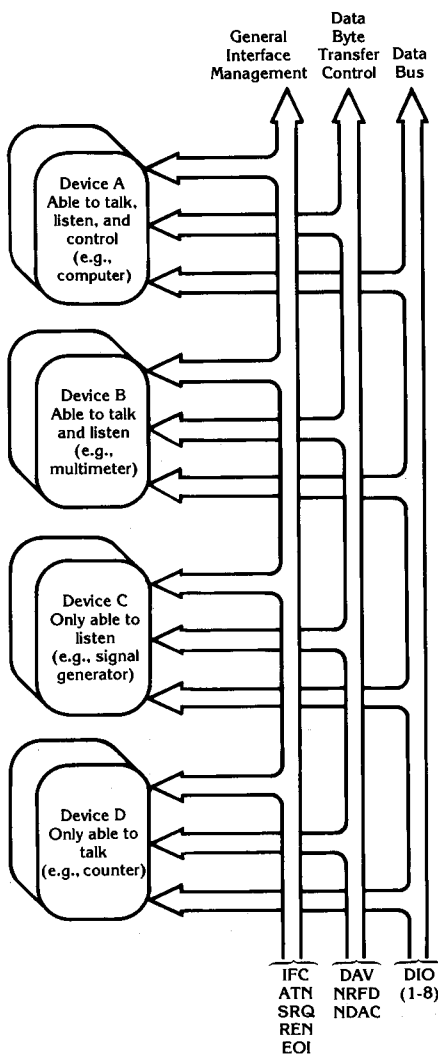


Figure 1

lines of the HP-IB. There are a total of sixteen signal lines, divided into three groups. The first group is composed of eight data lines, forming the data bus. These lines are bidirectional and carry information and messages between devices.

The data byte transfer control group is composed of three lines called DAV (dava valid), NRFD (not ready for data) and NDAC (not data accepted). As the name of each of

these lines implies, this group is used to control the flow of information over the data lines. The five remaining lines form the general interface management group. These lines are used for control and status information pertaining to the HP-IB devices attached to the bus.

## Assigning roles

Four devices are shown attached to the HP-IB in figure 1. Device A is a controller, a talker and a listener. As a controller, it may assign the role of the active talker to any device on the bus capable of being a talker, including itself. As a talker, it can supply information to other instruments on the bus, and, as a listener, it can receive information from a talker.

Although device A is shown as the only controller in this example, more can be accommodated in an HP-IB system. Only one controller can be active at a time, however, to prevent conflicts. A controller that is designated system controller becomes active at power-up. All others must remain passive until control is passed to them. The IEEE standard specifies the signals and timings necessary to do this.

Device B is both a talker and a listener. It can be addressed by the controller and made an active talker. The active talker has control of the DAV control line in the data byte transfer control group. Device C can only be a listener, and can be addressed by the controller and made an active listener. Device D is only a talker and can be made an active talker by the controller.

Active listeners have control of the NRFD and NDAC lines in the data byte transfer control group. The active talker drives the data lines and the active listeners read the information. To avoid conflicts, only one talker is allowed to be active at a time, but several listeners may be active simultaneously.

*HP-IB is a combination of the hardware specified by the IEEE standard plus a communication technique that creates an enhanced version of IEEE 488.*

### Transferring information

The potential existence of several active listeners receiving data simultaneously presents a problem. The active listeners may not be capable of accepting data at the same rates. The speed of information transfer must be controlled by the slowest active listener in order for data not to be lost. The data rate sequencing is controlled by an electronic voting system called "open collector." The transfer of information takes place as follows:

1. All active listeners indicate on the NRFD line their state of readiness to accept a new piece of information. The line is usually connected to a voltage through a resistor. This resistor causes the line to assume the same voltage on both sides of the resistor if there are no loads on the line. If an active listener is not ready, it turns on a transistor connected between the NRFD line and ground. The turned-on transistor acts like a short, pulling the voltage of the NRFD to ground. The active talker will not transmit the next data byte until the voltage on the NRFD line reaches its high voltage condition, when all of the active listeners have become ready and have released the NRFD control line.
2. The active talker, having put valid data on the data lines at least two microseconds (0.000002 seconds) ago, asserts the DAV line by pulling it low. Two microseconds is a settling time to allow the data to reach valid logic voltage levels. Assertion of the DAV line is a signal for the active listener(s) to read the information on the data bus.
3. During the previous portions of the data transfer, the NDAC line

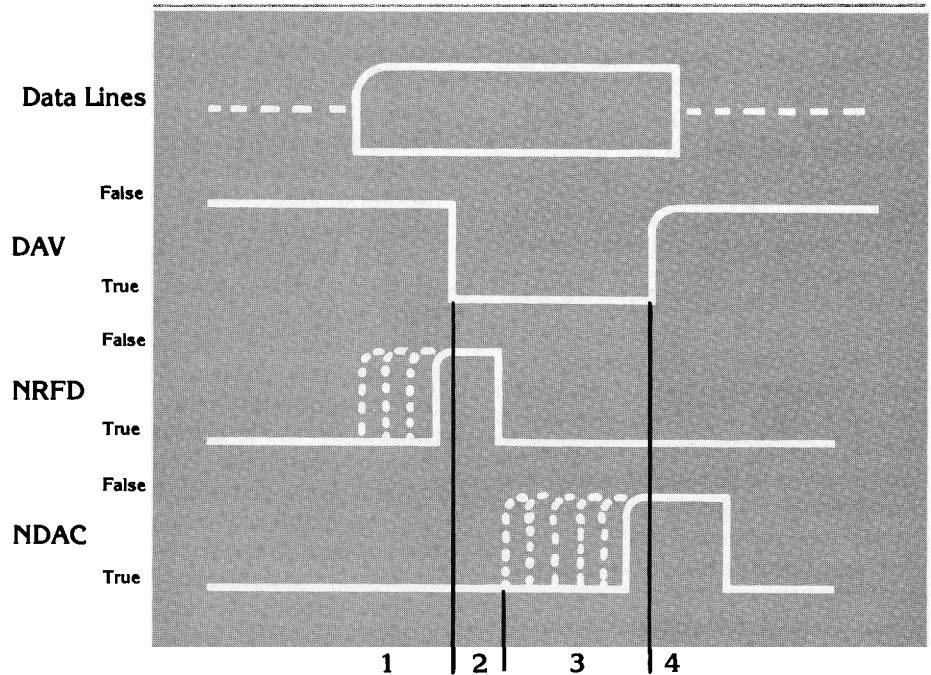


Figure 2

has been held in the low state by the active listeners. When DAV is asserted and the active listeners accept the data, they will release their hold on the NDAC line. When the last active listener releases its hold, the line will be pulled high by a resistor.

4. The active talker waits until it observes the NDAC line in the high state, signifying that all active listeners have accepted the byte. It then releases the DAV line. The release of the DAV line is the cue for the active listeners to again pull down on the NDAC line in preparation for the next data transfer.

A timing diagram of the complete handshake is shown in figure 2. Note that control of data transfer is effected by the active talker and active listener(s). Once the controller has configured the bus, it takes no part in the data transfer.

### Configuring it out

Now that we have examined how data is transferred on the HP-IB, let us consider the operation of configuration. One of the general interface management lines is called ATN (attention). This line, run by the active controller, signifies whether the information on the data lines is for control or data transfer.

When the controller wishes to configure the HP-IB, it asserts the ATN line. This causes any active talker to relinquish the DAV line. The transmission of control information is similar to that for data, but the active controller takes the place of the active talker and both talkers and listeners accept the information.

The active talker and active listeners may be designated during the transmission of this control information. The information is on the data lines. The following table shows what these values are:

*One of the best features of the IEEE standard  
is that a system user need not know any of this information.*

Bit number	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

Note that bit 7 is not used (DIO 8). Bits 6 and 5 are used to designate one of four classes of control information. A bus command (bit 5 = 0, bit 6 = 0) is used to directly control the devices on the bus. Such functions as triggering of devices and passing control require bus commands. Listen addresses (bit 5 = 1, bit 6 = 0) are used to activate listeners. A listener that observes its listen address on the bus when attention has been asserted becomes an active listener. The state of other listeners remains unchanged.

### Unlistening

Thirty-one listen addresses are possible, from 0100000 to 0111110. The last code in the listen address class, 0111111, is the unlisten command. All active listeners become inactive when an unlisten is sent. Talk addresses (bit 5 = 0, bit 6 = 1), are similar to listen addresses except that the definition of an active talk address causes any other active talker to become inactive, since only one active talker at a time is allowed.

The 1011111 pattern is the untalk command, leaving no active talkers on the bus. Secondary addresses (bit 5 = 1, bit 6 = 1) are used to address subunits within a device. Some devices may provide more than one simultaneous function and require more extensive addressing than the talk and listen addresses provide.

The remaining four lines in the general interface management group are used to control the interface sections of the HP-IB devices. IFC (interface clear), is used by the active controller to override all bus activity and put the HP-IB into a known state. Such an action is abortive to any data

transfers in progress and is used when something has gone wrong.

REN (remote enable) is a line that allows the HP-IB to control a device. The active controller indicates whether an addressed listener will use programming information sent to it by a talker by using REN.

EOI (end or identify) is used in two ways. It may be asserted by the active talker to designate the last byte in a data transmission, and it is used during a parallel poll, discussed later. SRQ (service request) is a line which a device may use to get the attention of the active controller. Note that this is a request, not a demand, and may be ignored by the active controller until there is time to service the request.

When the controller decides to acknowledge the service request, it has to discover which device on the bus issued the request. Since all devices on the bus share the SRQ line, all service requests look alike.

### Polling along

There are two ways the active controller can determine the address of the requesting device. Both methods are polls. A poll is a request for status information. The active controller may request the status of a device individually, by addressing the device as a talker, and sending the device a serial poll enable command, one of the bus commands sent while ATN is asserted. The active controller can then obtain eight bits of status information about that device. Serial poll disable must then be sent to return the device to the data mode.

For faster decisions, a parallel poll may be made. The active controller asserts ATN and EOI, thus requesting a parallel poll. Up to eight devices may respond, each one using a different data line (DIO1 to DIO8). If a device is requesting service, it will pull down on its data line, signifying that condition.

### You don't have to know

One of the best features of the IEEE standard is that a system user need not know any of the preceeding information. It is built into the definition of the interface and is supposed to work correctly for any device built to the IEEE specifications. What, then, does the sytem user need to know?

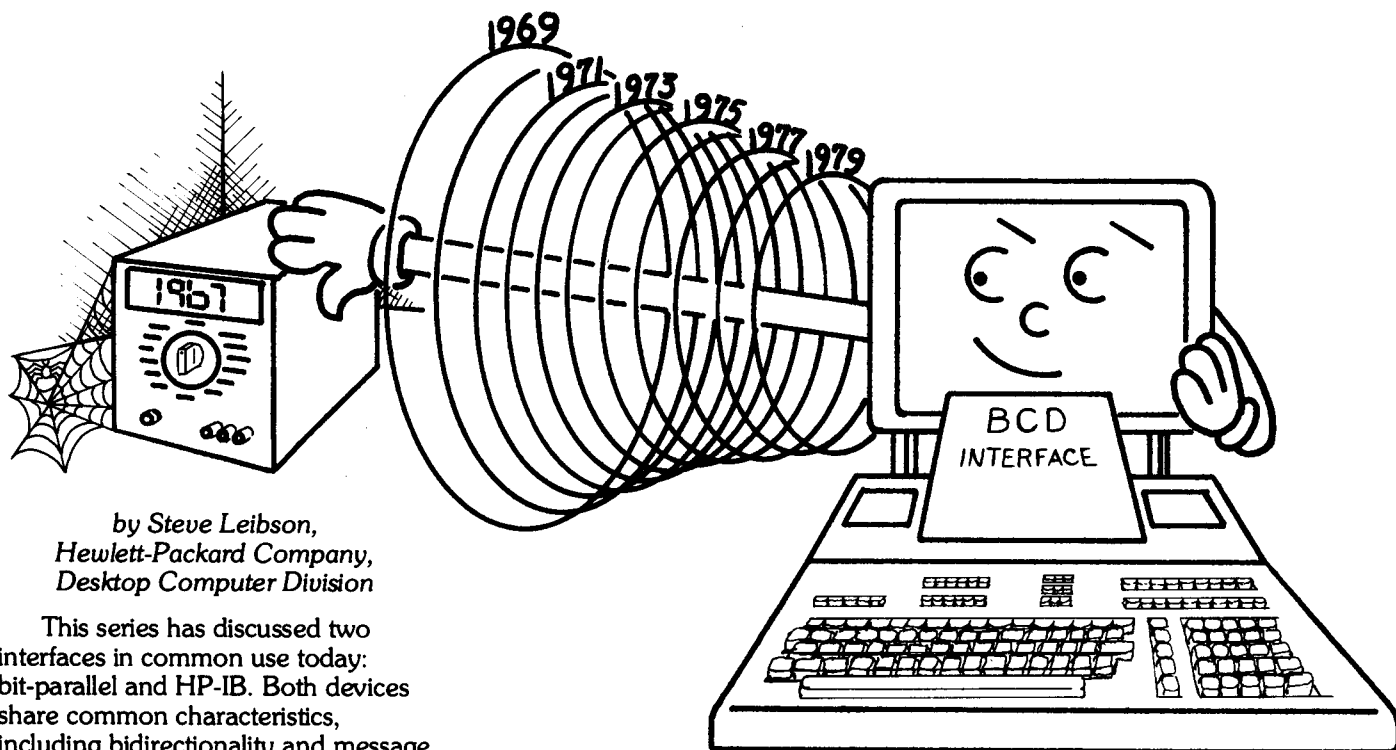
The actual messages and data formats sent are not specified. They are service dependent. For example, a voltmeter wishes to inform the desktop computer that it is reading +1.433 volts at its input. When it becomes the active talker, what should it send? Since most computers recognize the ASCII character set, it would be nice to send ASCII. The only decisions left are the format and the order of the digits, least-significant to most-significant, or the opposite.

Most computers prefer the most-to-least-significant order, and the voltmeter would send +,1.,,4,3,3,CR,LF. The CR and LF characters stand for carriage return and line feed, two characters used to terminate a transmission.

The definition of messages and message formats leaves the area of the IEEE standard and enters the realm of HP-IB, which is the HP implementation of the standard. And that removes yet another level of interfacing problems from the shoulders of the system user. ☒



# The BCD interface



by Steve Leibson,  
Hewlett-Packard Company,  
Desktop Computer Division

This series has discussed two interfaces in common use today: bit-parallel and HP-IB. Both devices share common characteristics, including bidirectionality and message handling in portions (characters).

When instruments were first connected to computers, instruments by themselves did not have the electrical sophistication in their circuitry for either of these interfacing techniques. A different method of connection, called binary coded decimal (BCD), was used.

## Computer bears burden

This method allows the burden of the intelligence for the interface to reside with the computer, which somehow has to accept all of the information in parallel. But interface designers created the required circuits, and the BCD interface remains popular in instrumentation. It provides a link to older instruments that have been reliably turning out data for years. BCD also is generally simple to design into a current instrument.

Equipment which uses the BCD interface usually measures some physical parameter such as voltage, current or weight. These instruments send information to the computer, but do not receive information from it.

Thus, the BCD interface is unidirectional. Information flows only from the instrument to the computer. Control lines may be available from the computer to actuate ranges or control other aspects of the readings, but they are not used for messages.

## Using 10 binary codes

BCD is simply a coding method which takes the ten decimal numerals 0 through 9 and encodes these into 10 binary codes. The encoding is the binary sequence 0 through 9 as follows:

Binary code				Numeral
Bit 3	Bit 2	Bit 1	Bit 0	
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

Note that the encoding is simple binary and that four binary digits (bits) are required to represent 10 numerals. Also note that codes 1010 through 1111 are not used. Because of this, BCD coding is not as efficient as pure binary coding. This inefficiency allows simple decoding for display to human operators.

An analogous situation is that of a calendar. Each page contains five weeks, which is more than enough to hold any month. The extra spaces for days are left blank, which is inefficient. But because people block the days of the year into months, the convenience of separating the months into different pages more than makes up for the blank spaces.

Each digit of a reading therefore requires four signal wires to transmit the binary values associated with that digit. Since all digits are available on the I/O connector simultaneously, the connector may have as many as 40 or 50 signal pins on it for a high-resolution instrument.

*At its inception, BCD interfacing was a big success. The tedious job of taking data was greatly simplified.*

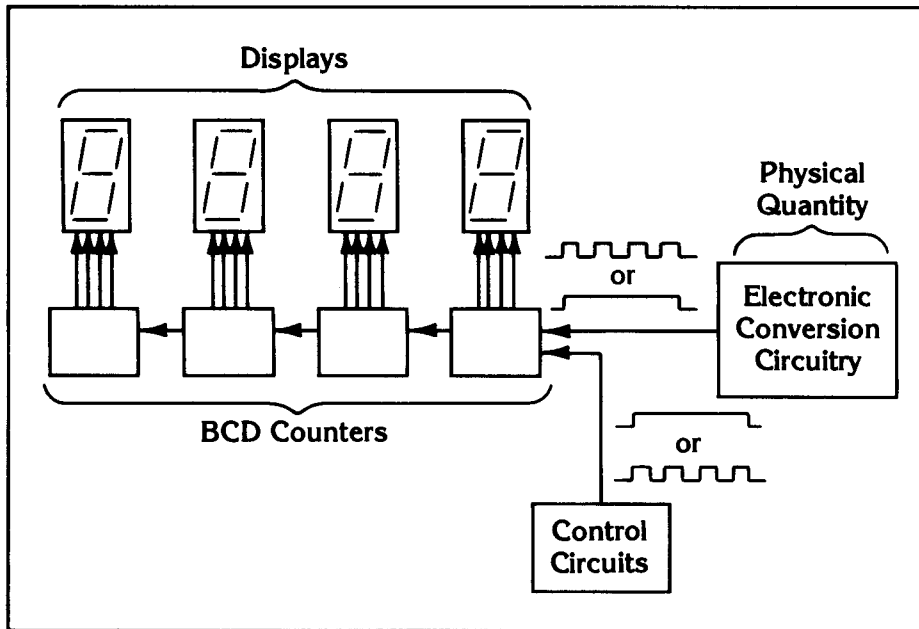


Figure 1

### Implementing BCD

Why is the BCD interfacing technique easy to implement in these measuring instruments? The answer lies in the circuitry used to actually do the measuring. A diagram of a typical measuring instrument is shown in figure 1.

Input to the instrument, such as voltage, current or weight, is fed to an electronic circuit that converts the physical parameter to a signal that represents that parameter. This signal may be either a square wave with a frequency that is controlled by the physical parameter, or a pulse with a duration that is controlled by the parameter.

If the signal is a frequency, it is fed to the input of a counter which counts for a precisely-controlled length of time. If the signal is a pulse, it controls a counter which counts the oscillations of a precise square wave.

In either case, a count is obtained which is proportional to the input parameter. A small input generates a small count, and a large input

generates a large count. The count drives the digits in a display. If the count is binary, it is difficult to read because people don't think in binary, computers do. Fortunately, it is possible to build a BCD counter.

If several BCD counters are cascaded, they are able to produce outputs that are easily displayed. This works in much the same manner as the odometer of a car. Each wheel of the odometer has the numerals 0 through 9 printed on it. Each time a wheel on the odometer makes a full turn, it advances the wheel to the left of it by one count.

In a chain of BCD counters, each time a counter advances from 9 to 0, it advances by one the next counter on the chain. Each BCD counter has four signal lines that represent the state of the counter. The signal lines are used to drive one digit of a display. When the digits from all of the counters are combined, they form the complete reading of the instrument.

The preceding explanation is true for a wide range of measuring

instruments that have digital displays. Most use a counting technique to convert a physical quantity into a digital display.

### Adding a printer

The first accessory instrument designers added to digital instrumentation was a printer. This made it possible for an unattended instrument to log its own readings. Signal lines are brought out from the counters in the instrument to drive the print wheels in the printer. Each digit has its own wheel in the printer. Signals from the BCD counter control the position of the print wheel when it hits the paper.

Extra signals are only required for this interface to tell the printer when the data on the BCD lines is valid (print command), and to allow the printer or other external device to control the rate at which readings are made (external trigger). These two wires form a handshake mechanism between instrument and printer.

### Automating experiments

At its inception, BCD interfacing was a big success. Experiments which had required an attendant to write down the readings could now be automated. Printed logs could be obtained for production testing. The tedious job of taking data was greatly simplified.

Now if a printer could do all that, just think what could be done by replacing the printer with a computer. Data would no longer have to be punched on cards or entered by hand. The eyes and ears of a computerized process control loop were about to come into being.

### BCD at HP

Let's look at a BCD interface for a Hewlett-Packard desktop computer

(figure 2) and see how it works. The 98033A BCD Interface is used with the 9825A/S, System 35A/B and the System 45A/B. It has several inputs that connect to the instrument. There are enough signal wires for an eight-digit mantissa with a sign bit and a single-digit exponent, with a sign.

In addition, there is a bit to represent overload or overflow and four bits for a function code. When the computer takes a reading, the interface card scans its input lines and transforms the BCD digits, all available in parallel, into a string of ASCII characters which the computer reads one at a time.

Sixteen characters form one reading as shown in figure 2. In this way, the reading may have as many as eight digits in the mantissa, and a single-digit exponent. This is usually more than sufficient to handle a BCD instrument. Unused digits can be wired to always read zero.

### Interfacing flexibility

Since there is no standard for BCD interfacing, the 98033A provides flexibility in the interpretation of the signal wires. It can be configured for either positive true logic, where logic 0=0 volts and logic 1=+5 volts, or negative true logic, where logic 1=0 volts and logic 0=+5 volts. Note that the voltage levels 0 and +5 are TTL standard, a logic family introduced in the late 1960s and currently dominating logic design.

Character numbers 10, 13 and 16 ("E", ",", and "LF") are generated within the 98033A Interface. They aid the computer in deciphering the meaning of individual digits coming from the instrument. The "E" is a prefix that indicates an exponent will follow. The comma separates the reading from the overload bit and the function code. "LF" is a line feed character that terminates the message.

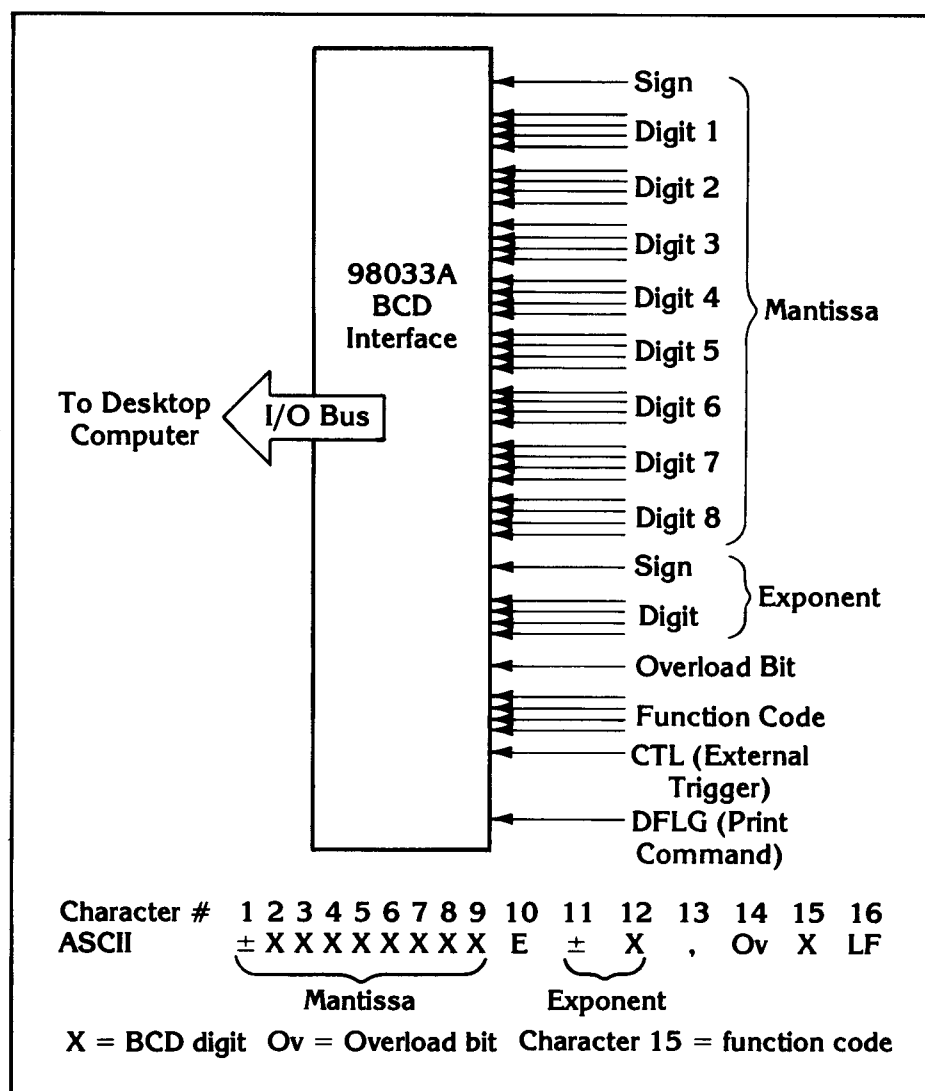
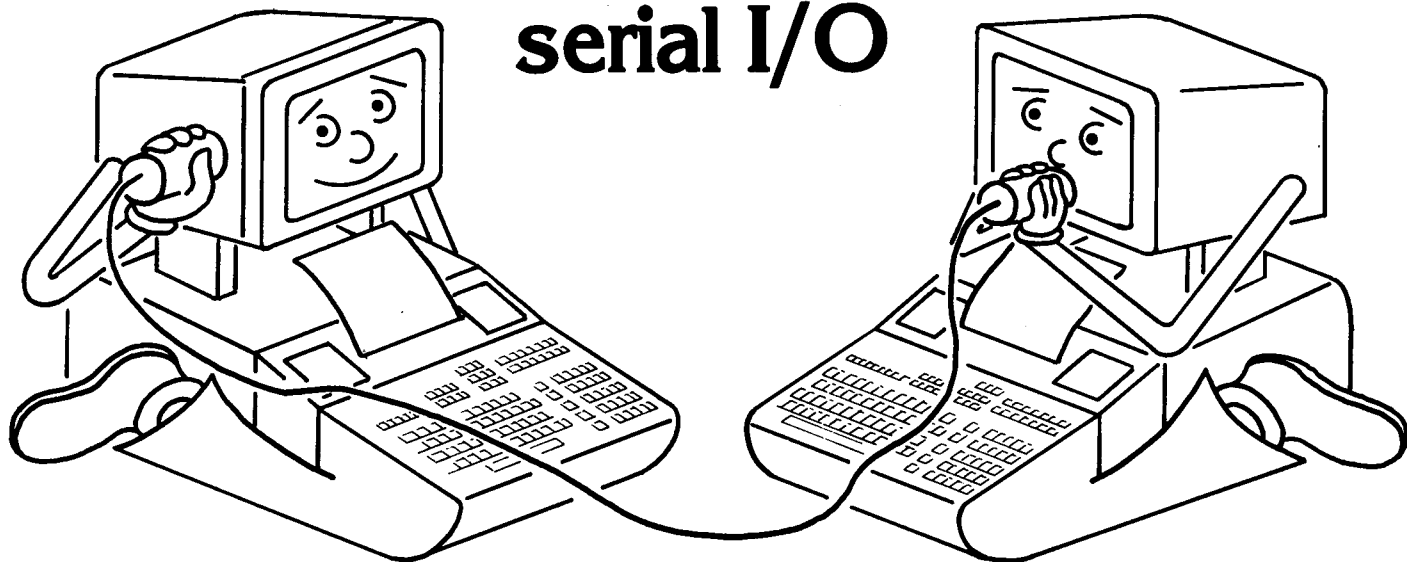


Figure 2

Intelligence in the 98033A replaces that which might otherwise be required in the instrument's interface. This interface serves as an electrical link to the past. It provides a data path between today's computers and earlier interfacing instruments. It also eases the burden for instrument designers who may not be able to justify development costs for a more complex interface. ☒

# The unstandard interface: serial I/O



by Steve Leibson  
Hewlett-Packard Company  
Desktop Computer Division

The fact that the computer represents a relatively new technology may lead you to believe that I/O does also. The first electronic computers appeared in the 1940s, and serious work in computer data communications did not start until the next decade.

But when engineers did begin to connect computers to other devices, they used a technology that originated in the previous century — serial I/O.

## Encoding for machines

The first electrical device used extensively for communications was the telegraph. Samuel Morse improved the telegraph mechanically, but more importantly, he devised the Morse Code. This was the first really practical encoding of the symbols humans use for communications into a machine-transmittable form.

Symbols are represented in the code by a series of dots and dashes, each character having its own unique representation. The dots and dashes may be considered the predecessors to the ones and zeros of the modern

character codes we use in computer data communications.

Improvements of the telegraph led to printing telegraphs that required no human operator to decipher the codes. New codes and more advanced machines were devised, culminating in the teletypewriter.

By the time of the teletypewriter, dots and dashes had become ones and zeros. Morse code was discarded in favor of codes that assigned the same number of bits to each character. This made it much easier for machines to decode. By the time electronic computers were invented, there already existed a wealth of technology for electronic data communications.

At first, teletypewriters served as I/O devices between humans and computers. The keyboard and printer of the teletypewriter provided a low-cost data entry and display mechanism. As technology progressed, terminals and faster printers replaced the teletypewriters, but retained serial I/O interfacing.

## Transmitting over one wire

The basis for serial data communications is the transmission of information over a single wire.

Interfacing techniques previously discussed in this series have relied on several parallel wires to carry information between devices. Each wire carries a single bit of a character composed of multiple bits.

When long distances are involved, the cost of running several wires in parallel becomes prohibitive. Serial interfacing is the solution.

Time sharing was born when computers became sufficiently powerful to handle several tasks simultaneously. Since computers were still very expensive, it was necessary to spread their cost over many users.

The problem then was how to connect users at several locations to the central computer facility. Stringing wires to each location was too expensive. Fortunately, such communication lines already existed. They belonged to the phone system.

## Building a standard

Unfortunately, these connections were not necessarily wires. They could just as easily have been satellite or land-based microwave links, since these also made up the phone system. All were designed to carry voice signals, not computer data.

In addition, phone companies were extremely unhappy at the prospect of finding all kinds of strange signals in their networks. Because teletypewriters did not have standardized interfacing requirements the voltages involved could range anywhere from 6 to 140 volts. A standard was required.

The Electronic Industries Association (EIA) standard RS-232C resulted. This standard was specifically developed to do one thing. It defines the electrical characteristics for an interface between a piece of data terminal equipment (DTE) and a piece of data communications equipment (DCE). DTE is the terminal for the timeshare user, while DCE is a modulator-demodulator (modem) which encodes the computer data into voice-like signals that are permissible on the phone system.

Figure 1 is a picture of the wires associated with the RS-232C standard. Note that there is actually more than one wire involved. Pins 2 and 3 are the data-carrying wires, called transmitted and received data. Pin 7 is a signal ground serving as a signal current return path. These three wires are sufficient for communications between DCE and DTE.

What are all the other wires for, then? They serve as control wires between the DTE and DCE and are there merely for establishing and maintaining communications with the computer. Let's ignore them for now.

### Examining data wires

Let us examine the data wires more closely. Information is sent out on the transmitted-data line while signals are received on the received-data line.

Note that this is not the same as the HP-IB interfacing standard discussed previously. On that interface, one set of data lines carried information in both directions. Technology had not

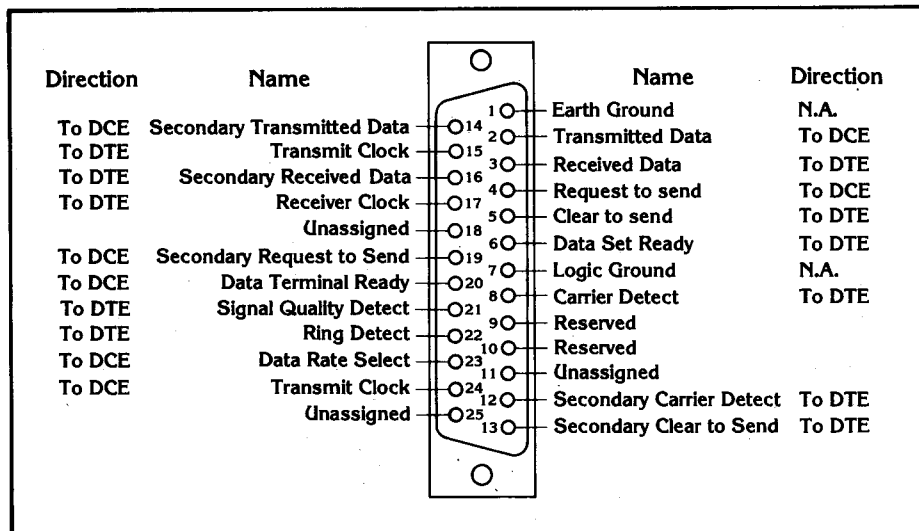


Figure 1

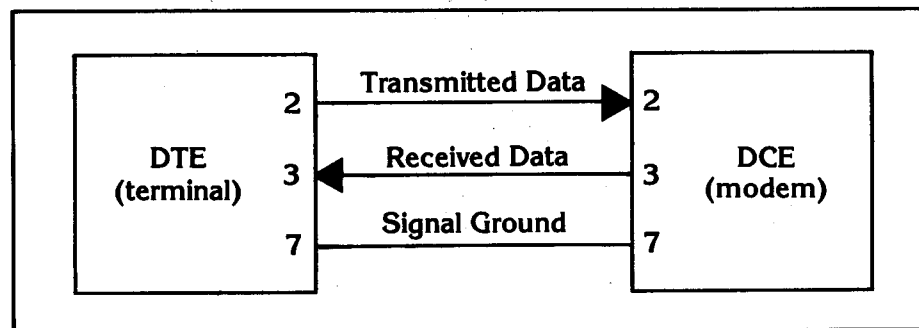


Figure 2

progressed sufficiently for RS-232C to have bidirectional signal lines.

Who transmits on the transmitted data line? All of the names for the RS-232C signals are from the perspective of the DTE. So the DTE transmits on the transmitted data line and the DCE receives on it.

Similarly, the DTE receives data on the received data line and the DCE transmits on it. Figure 2 should clear up any confusion about this.

Now let's get confused again. We have a computer and a printer. We are going to connect them using their "standard" RS-232C cables.

But which of them is the DTE and which is the DCE? More specifically,

which is going to transmit on pin 2 of the connector and which on pin 3? Neither the computer nor the printer is a terminal or a modem.

Manufacturers of these instruments may offer cables to allow their equipment to look like either DTE or DCE. More often however, the RS-232C connector is installed on the rear panel of the instrument and no choice is possible. In the case of two instruments of the same type, a crosswire cable may have to be assembled to get signals on the correct wires. Usually this task falls to the user.

We have just covered one source of incompatibility between pieces of RS-232C equipment: plug-to-plug. In

order to discuss other potential problems we must understand the data signal.

First, the data signal levels are not like those of the interfaces we have discussed previously. Those interfaces were based on TTL integrated circuits. That logic family is based on 0- and 5-volt signals.

RS-232C was a standard long before TTL and so uses different voltage levels. A positive voltage between 5 and 25 volts is used to represent a logic 0 while a negative voltage between -5 and -25 volts is used to represent a logic 1.

These levels are only for the data lines which use negative true logic. The control lines all use positive true logic and so a positive voltage represents a logic 1 and a negative voltage represents a logic 0.

Because the bits of a character are separated by time, a waveform is produced when transmitting a character. Such a waveform for transmitting the character "E" is shown in figure 3. The ASCII code for "E" is 1000101 in binary and is transmitted least significant bit first. The data line idles in the "1" state.

### Waiting for the start bit

A start bit is always sent first to mark the beginning of the character. Then data bits are sent in order from least significant to most significant, each bit remaining on the line for a precisely-arranged length of time called a bit time.

The receiver, alerted to the incoming character by the start bit, times the incoming signal and samples the state of the data line as close to the center of the bit as it can. Naturally, both the transmitter and receiver must agree on the length of time each bit will be given on the line or the transmission will be garbled by samplings taken at the wrong times.

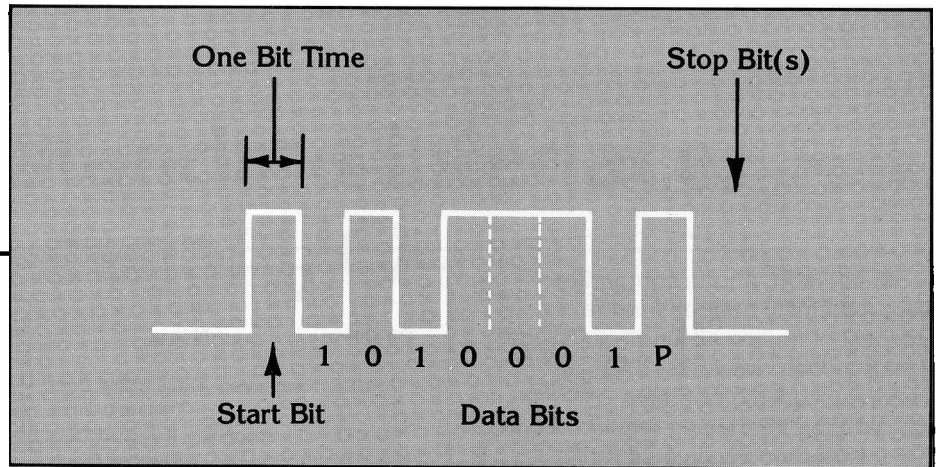


Figure 3

This bit time determines the maximum rate at which bits may be transmitted and thus defines the "bit rate" at which this particular serial interface is running. Standard bit rates are 50, 75, 110, 134.5, 150, 300, 600, 1200, 1800, 2400, 3600, 4800 and 9600 bits per second.

Following the data bits in the transmission, there may be a parity bit, used for error detection. If a noise pulse should affect the data line at the wrong time, a bit in the transmission could be misread.

If the transmitter keeps track of the number of 1s in the character being transmitted, it could set the parity bit so that the total was either always even (even parity), or always odd (odd parity). The receiver can also keep track and use the parity bit to determine whether the transmission was received in error.

The last bits to be transmitted are the stop bits. These bits carry no information but allow the receiver time to prepare for the next character. There may be 1, 1.5 or 2 bits. Since this period is really just a resting time, fractional bits are allowed.

### Picking up the pieces

Now, what might go wrong? There are several parameters on which the transmitter and the receiver must agree. The bit rate has already been mentioned. In addition, the parity (odd, even or none) and the number of stop bits both have to be the same in the transmitter and the receiver.

Character codes also have to be considered. Our example used the

ASCII code that is the most commonly used character code today. ASCII is a seven-bit code. Other codes do exist however, and may include 5 bits (Baudot and Murray), 6 bits (IBM Correspondence Code) and 8 bits (EBCDIC). The RS-232C standard makes no mention of what character code should be used.

Now that we have our computer and printer connected so that they use the proper wires and they agree on bit rate, parity, number of stop bits and character codes, you may feel that the problem has been solved. — Not so. The printer uses the ASCII character set, has odd parity, and one stop bit. In addition, it has a switch for setting the bit rate up to 9600 baud.

Well, that's fine. We set the switch to its fastest setting so that our equipment doesn't waste a lot of time sending characters. We also set the computer to the same settings so that both the transmitter and the receiver agree on all of the parameters.

### Finding the problem

We then write a program on the computer to send one line of print to the printer and it works! Finally, we list out the program on the printer so that we have a copy of our triumph. Unfortunately, several characters are lost in the transmission. We try again with equally dismal results. We run the program once more, and now that line prints perfectly. What is going wrong?

First, let's consider the data rate at which the computer is sending information. We are using ASCII (7 bits) plus a parity bit, a stop bit, and a



## *Surely RS-232C must have a handshake mechanism also — but no, this is not always true.*

start bit for a total of 10 bits. We are transmitting these 10-bit pieces of information at 9600 bits per second, which translates into 960 characters per second.

The printer manual specifies that the printer can print 175 characters per second. We are sending information to the printer at more than five times the rate it can print them! The printer does have an internal buffer for 127 characters. After that, the transmitter must wait to allow the buffer to partially empty.

When we ran the program, fewer than 127 characters were sent to the printer and the printer's buffer could handle the data rate. Listing the program sent more than 127 characters to the printer and the buffer overflowed, causing some characters to be lost.

In previous articles, we discussed interfaces that had handshake mechanisms which prevented transmitters of information from going too fast for their receivers. Surely RS-232C must have a handshake mechanism also — but no, this is not always true.

Returning to figure 1, we will now consider the other RS-232C signals. We are looking in particular for two sets of handshake lines, one for the transmitted data line and the other for the received data line.

Aha! Pins 4 and 5, request-to-send and clear-to-send, look like prime contenders. And many printer manufacturers have fallen into this trap. According to the strict RS-232C definition, the DTE asserts the request-to-send line when it has some data to transmit. It then waits for the DCE to assert the clear-to-send line before transmitting.

That is one half of a traditional handshake. The problem is that the DCE is not allowed to drop the clear-to-send line until DTE has dropped the request-to-send line.

### **Take a long drink**

The situation is similar to taking a drink from a garden hose with a friend controlling the spigot. It's easy to start the flow, but you had better be prepared to take either a long drink or a short shower.

The DTE and DCE signals were intended as a handshake between the terminal and the modem to allow the terminal to request control of the communications link from the modem. It also makes it possible for the modem to tell the terminal when control has been acquired.

Some manufacturers have ignored this strict definition and used the clear-to-send line as a handshake line anyway. Others avoid the definition conflict by using the data-terminal-ready or data-set-ready line (depending on whether they are emulating a terminal or modem). None of these lines was intended for the purpose of handshaking characters, however, and use by one instrument does not guarantee recognition by the other.

Let us consider the possibilities of using the clear-to-send line. If the printer drops the clear-to-send signal in the middle of a character, what should the computer do?

If it stops immediately in the middle of a character, the character is certain to be garbled. If it waits till the end of the current character to stop, it may overrun the receiver's buffer. Because this possibility is not covered in the standard, the results are not predictable without carefully reading manuals for both instruments.

Finally, consider the device that started this discussion, the teletypewriter. The RS-232C standard at least defines signal levels and a pinout on a connector. There are no standards for teletypewriters. The serial transmission concepts are the same with start and stop bits, data bits

and parity bits, but the signal interface is called current loop.

Instead of positive and negative voltage levels to represent logic 0 and 1 levels, current loop uses the absence or presence of current. Presence may be either 20 or 60 milliamps depending on the teletypewriter model. There is no standard connector or pinout for teletypewriters.

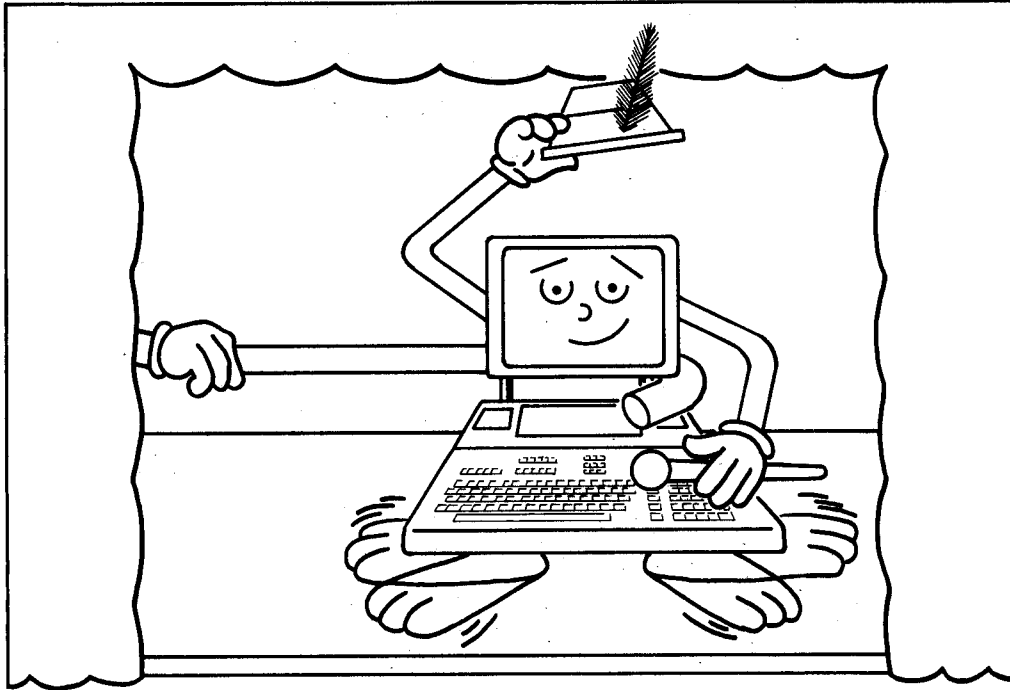
Despite all of these problems, designers of serial interfaces for computers strive to include current loop capabilities in their designs. Where RS-232C is limited to 50 feet for a direct connection, current loop can be run much farther. In addition, the teletypewriter interface has been around for many years and several instruments still useful to connect to computers use it. Teletypewriters remain a cost-effective solution as a combination printer and terminal.

### **Handle with care**

As you can see, serial interfacing needs to be approached more carefully than other forms of hardware interfacing techniques. There are several factors that are far from being standardized.

As technology progresses, serial interfaces become more adept at covering an ever-widening range of hardware. Unfortunately, this still doesn't guarantee an efficient interface in every application, because serial I/O remains the unstandard interface.

# Interrupt I/O: getting the attention of the processor



by Steve Leibson,  
Hewlett-Packard Company,  
Desktop Computer Division

What do you think is the most important part of the telephone? The dial? The receiver? The cord?

I submit that it is the bell. If the telephone had no way to summon you when a call came in, you would have to check it periodically to see if there was someone on the line.

The inconvenience of lifting the receiver every few seconds would quickly make the instrument seem very irritating. Fortunately, telephones do have bells, which interrupt you when someone calls.

## Waiting for peripherals

Early in this series we discussed the relative speeds of computer processors and peripheral devices. The mismatch in speeds necessitated the creation of handshake lines that the processor could check to see if the peripheral was busy. Without these lines, the speedy processor would inundate the poor peripheral with data.

The use of these handshake lines is the simplest form of I/O. The computer spends much of its time

patiently waiting for the peripheral to get ready for the next transaction.

## Interrupting

The above situation is quite satisfactory if there is nothing else for the computer to do. Frequently, however, there are many other things the computer could be doing, and the use of handshake I/O is inefficient. Fortunately, an alternative exists in most modern computers. It is interrupt I/O.

First, let's decide what it is that we will be interrupting. The computer is continuously executing a program in its memory. If there is no user program currently running, then at least the operating system is executing.

Thus, we have two levels of programs in the computer. The highest level is the user program, usually written in a high-level language such as BASIC.

Microprocessors currently cannot run a BASIC program directly, and so have a second, lower-level program which interprets the BASIC statements. This lower-level program is written in machine code, instructions that can be directly executed by the processor. This program is called an interpreter.

Interrupts are hardware mechanisms for forcing the processor to leave the part of the program it was executing just before the interrupt and start execution at a different location in memory. This interruption takes place at the machine-code level. It is a very useful mechanism for synchronizing external events with the computer program, but it must be used carefully. Let's take an example.

Suppose that a user program asked the computer to compute the value of  $2.5 + 2.5$ , print the answer on a teletypewriter and then compute the value of  $3 + 3$ . The computer would first execute the routine in the BASIC interpreter that performs floating point addition to produce the result: 5.00.

This creates a total of six characters to print: 5, ., 0, 0, carriage return and line feed. We can assume that the addition takes two milliseconds. Teletypewriters print ten characters per second, so the printing of six characters will take approximately 600 milliseconds (actually a little longer because the carriage return requires extra time).

Handshake I/O requires the computer to wait out the full 600 milliseconds before performing the second addition. The alternative

## *BASIC programs do not require an interrupt service routine for HP desktop computers, because the routine is in the interpreter.*

offered by interrupt I/O is that the characters to be printed can be placed in memory somewhere, in an area designated as the I/O buffer.

### **Interrupting machine code**

The first character to be printed then is sent to the teletypewriter, causing the interface to the peripheral to "go busy," transferring the character to the printer. Now the computer can proceed to the next BASIC statement, confident that when the teletypewriter has finally printed the first character it was given, it will become ready for the next one. At that time the interface will interrupt the processor and ask for another character.

Note that it is the machine code interpreter that is interrupted and not the BASIC program. The flow of execution of the BASIC statements is not changed, but the interpretation of the program into machine code is stopped while the computer outputs another character. This illustrates the use of interrupt for buffered I/O.

The writer of the BASIC program does not have to write an interrupt service routine for Hewlett-Packard desktop computers because the routine has been provided in the interpreter. This is quite convenient because many factors must be carefully handled in such a routine. The interrupt forces a branching in the machine code program to a different location.

If the processor does not remember where it was before the interrupt, it cannot return and will be "lost," unable to continue operating properly. Most processors automatically save the address of the location being executed before the interrupt, and a return from the interrupt is sufficient to restore that address.

If the interrupt service routine uses any of the internal registers in the

processor, it must first carefully save the contents of these registers and then restore them at the end of the interrupt service routine. This must be done, because it is difficult to tell what information in these registers was important to the program that was interrupted. By saving and restoring the registers, the processor is left as it was found and the interrupted program will not be affected.

### **Interrupting BASIC**

Sometimes, the buffered I/O routines are not sufficient for handling the problem. Some problems require more complex action from the computer than the transfer of a piece of information. In these instances, it is necessary to interrupt the BASIC program itself and branch to an interrupt service routine written in BASIC.

Interrupting the BASIC program is considerably more complex than interrupting the machine code program. BASIC statements can affect large portions of memory such as those used to store the values of variables. If a variable is being changed just as the interrupt comes in, and the BASIC interrupt service routine also uses that variable, the wrong value or a garbled value may be used in the interrupt service routine.


### **Waiting for the end of the line**

To prevent such problems from arising, Hewlett-Packard desktop computers force BASIC-level interrupt service routines to wait until the end of the current line has been reached before the actual branching occurs. This is called end-of-line branching. The interrupt can be logged in at any time during the execution of a BASIC statement, but the granting of the interrupt is withheld until the end of the line.

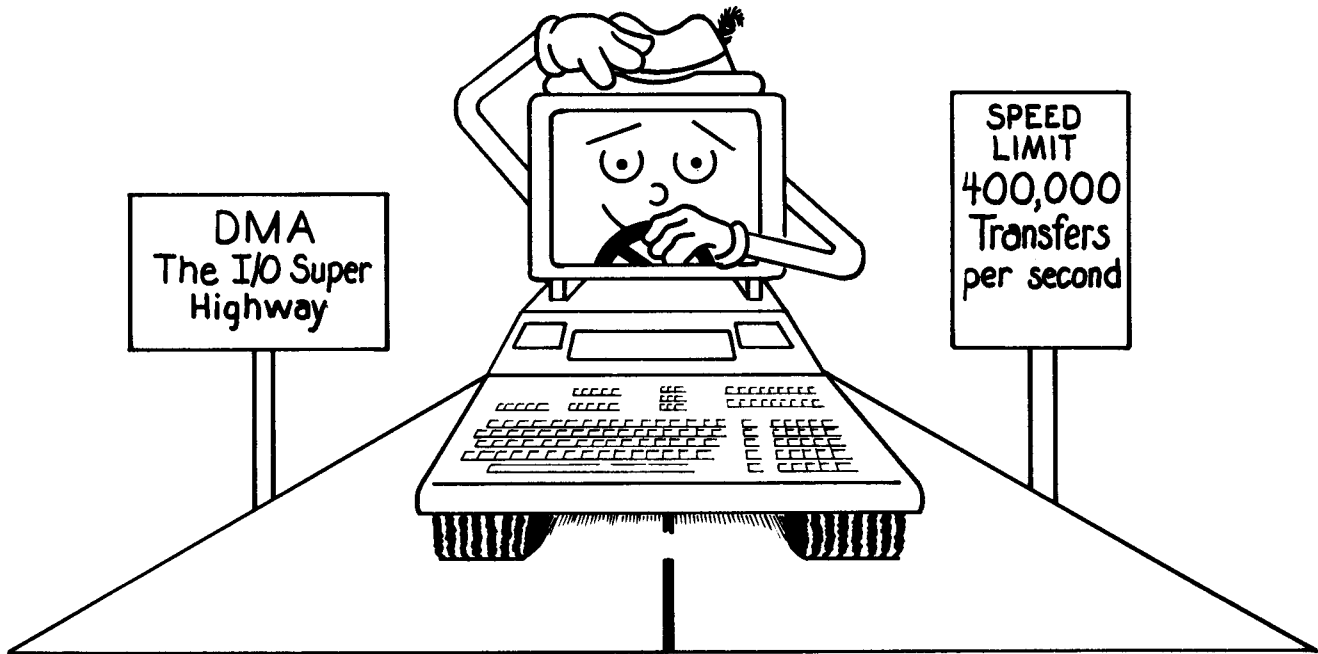
Machine code, or low-level interrupts, are generally called hardware interrupts because the processor hardware performs the interrupt request granting and the subsequent branching. Interrupts of the BASIC, or high-level program, are called software interrupts because several instructions in the operating system are required to log in the low-level interrupt, request the end-of-line branch and then take control of program flow at the end of the line.

Finally, let's consider what is actually meant by the interrupt. A classic example of misunderstanding interrupt occurs whenever a first-time writer of interrupt service routines tries to use interrupt for input. The typical programmer will enable the interface to interrupt and expect that when the interrupt comes, the interface will have the desired piece of data.

Unfortunately, the interface actually interrupts whenever it is not busy. Since the interrupt service routine did not make the interface go busy by requesting acquisition of the data before enabling the interrupt, the interface interrupts immediately, as it had nothing to do.

The interrupt service routine then ends up with no data or old data. The key is that to properly use interrupts, the first data transfer is performed before enabling interrupts, and subsequent transactions are performed under interrupt. 

# DMA: the I/O superhighway



by Steve Leibson  
Hewlett-Packard Company  
Desktop Computer Division

The articles in this series have described the hardware and circuitry necessary to interface peripheral devices with computers.

All the discussions thus far have assumed that the computer processor is in control of the data transfer process. This is true for many of the devices interfaced. The processor is usually fast enough that the peripheral device determines the data transfer rate.

Some devices, however, are too fast for processor-controlled I/O. These devices are capable of data rates approaching the speed of the computer memory and require a different I/O technique. The technique for interfacing such fast peripherals is called direct memory access (DMA).

In the previous article, we discussed interrupt I/O, which is used for interfacing with devices so slow that it is very inefficient to have the

processor wait for the completion of each I/O transfer.

Instead, the processor initiates a transfer and then continues with other processing. When the peripheral device is ready for the next transaction, it interrupts the processor and reminds it of the previous I/O commitment.

The interrupt I/O technique is used as a software transformer to match the slow peripheral with the fast processor. If the peripheral device is faster, the computer processor may only be able to execute the few machine instructions necessary to perform the I/O transfer before the peripheral is ready for another. Here there is a good match between the I/O software and peripheral speeds, and programmed I/O is sufficient for the task.

## Speedy peripherals

Ultimately, there is a class of peripherals too fast for even the few instructions needed to perform programmed I/O. As long as these

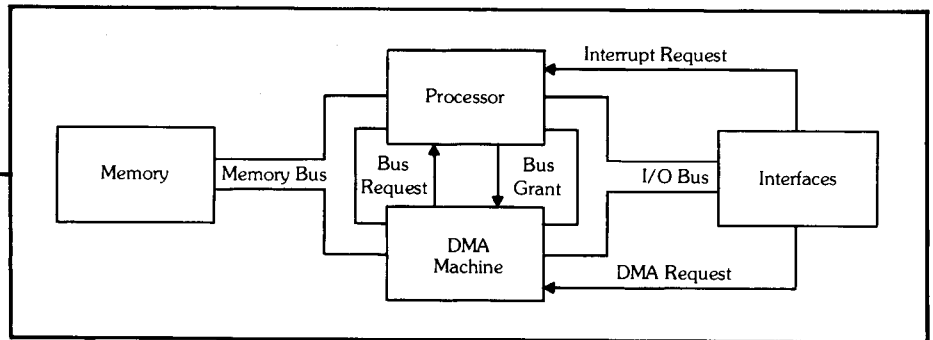
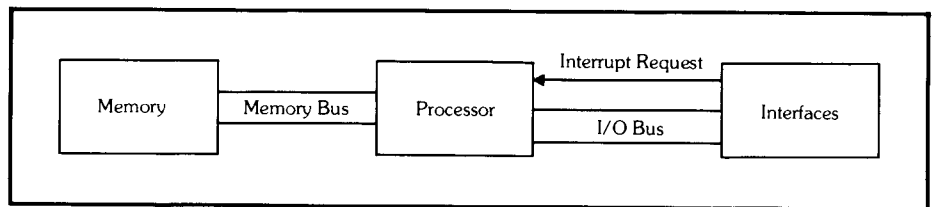
peripherals are not faster than the computer's basic memory cycle, there should be a method for performing the required I/O. There is and it is called direct memory access (DMA).

In order to discuss DMA and how it works, we must return to the model of the processor-memory-I/O system discussed in the first article in this series. Recall that the processor is linked to the memory via a set of lines called a memory bus and to the I/O interfaces via an I/O bus.

Both busses require the processor to generate address signals and control signals to synchronize the flow of data over these busses. Generally, I/O consists of taking information from the interfaces through the I/O bus and transmitting this information to the memory using the memory bus or vice versa.

## Inefficient throughput

During this transfer the processor is also using the memory and memory bus to supply machine instructions so that it knows how to effect the data



Diagrams above illustrate the differences between a system that does not include a direct memory access machine, top, and one that does, bottom.

transactions. If we assume that it takes only nine machine instructions to perform one data transaction, we can see that the effective I/O throughput is only 10% of the rate that the memory could support.

That is, for every 10 memory cycles, nine are used to instruct the processor and only one is used to place data for I/O. Only very simple data transactions can be performed with nine machine instructions. If formatting or code conversions are necessary, many more instructions are needed.

#### Bypassing the processor

The only way to speed up the I/O process is to eliminate the slowest link in the data path. For high-speed peripherals, the slowest link is clearly the processor itself! How can we eliminate the processor when that is the component that links the I/O and memory busses and is required for the generation of the signals that actually make these busses work?

The answer is to build a specialized circuit that is designed to transfer data at the full memory speed. Because the only function this circuitry must perform is this transfer, the capability may be wired into the circuit and instructions from memory are not needed and do not reduce the effective memory bandwidth.

If we place this specialized circuitry so that it, too, bridges the I/O and memory busses and if we also give it the capability of generating the address and control signals required by these busses, then we have a machine that is capable of performing I/O at the full memory speed. This specialized circuitry is called a direct memory access or DMA machine. All that remains is to select which device will have control of the busses, the processor or the DMA machine.

#### Controlling DMA

Normally, the processor will have

control of the busses because the DMA I/O must be infrequent enough to allow at least some processing to be done. It is therefore necessary for the DMA machine to acquire bus control from the processor whenever necessary.

The processor can enable the DMA machine to request bus control, but it is the interface that must actually request service through the DMA machine. Only the interface knows when the attached peripheral requires DMA service. Thus we must add some connecting signals between the interface and the DMA machine, and between the DMA machine and the processor.

#### DMA handshake

The interface must have some means of requesting service from the DMA machine. A signal called DMA Request (DMAR), added to the collection of signal lines on our I/O bus, will be sufficient. Upon receipt of this request, the DMA machine must request bus control from the processor.

The processor may decide that the time of the request is inopportune and wish to hold off the transfer of control temporarily — This is a job for the everpresent handshake!

We will create two handshake lines called Bus Request and Bus Grant. The DMA machine will ask for bus control with Bus Request and wait to actually take control until it receives a signal on Bus Grant. Thus the processor can maintain control of the memory and address busses as long as required.

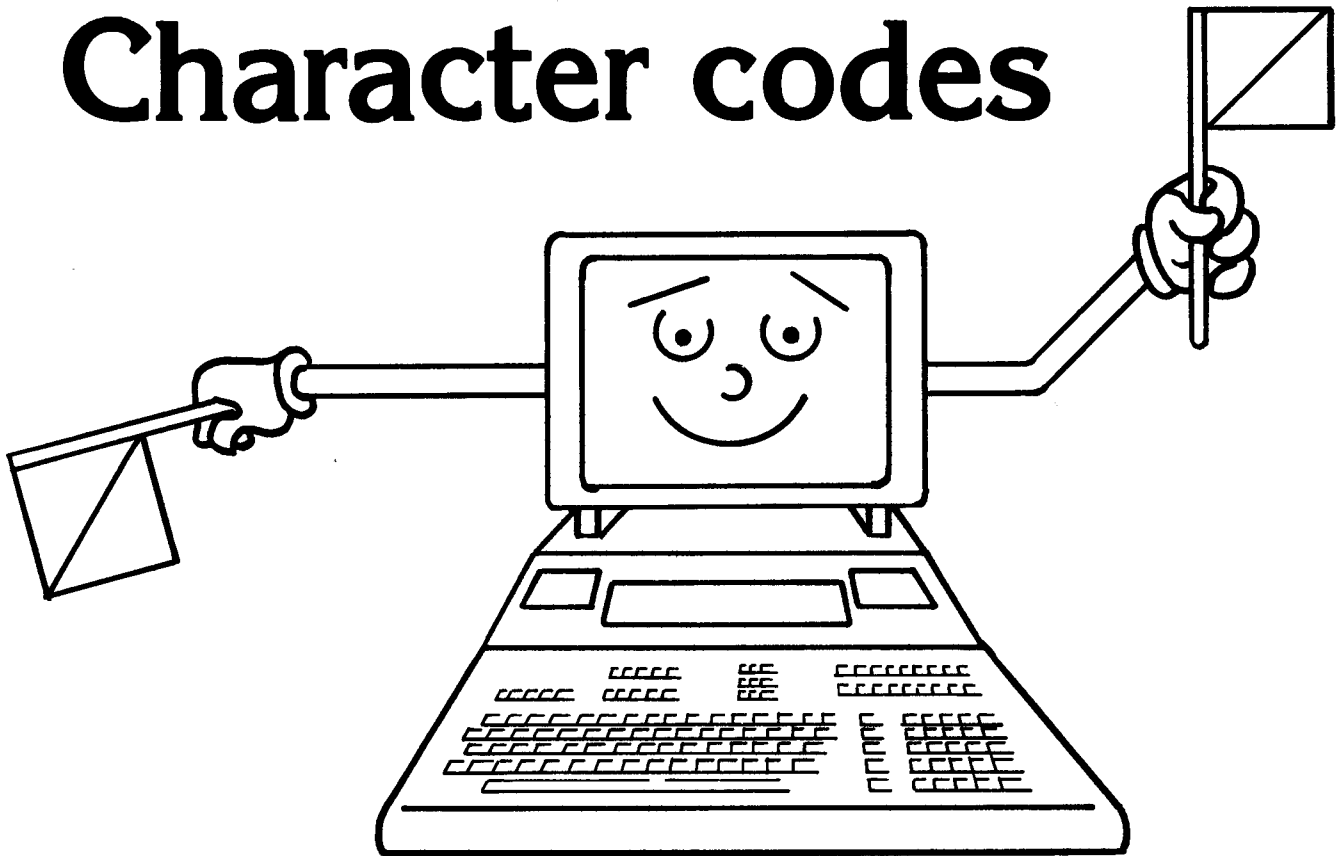
#### Burst and cycle-steal

The DMA that we have been discussing is called burst DMA because data transfer is done in a burst where the DMA machine totally controls the I/O with the full speed of the memory bus at the expense of completely halting any processor activity.

If half the memory bus bandwidth is sufficient to solve the high speed I/O problem, another type of DMA can be employed. Called cycle-steal DMA, the DMA machine alternates control of the busses with the processor, each unit using every other memory cycle. Cycle-steal DMA allows the processor to operate at 50% efficiency while still providing relatively high speed I/O.

At this point in the I/O series, we have discussed the basic hardware needed for interfacing computers to peripherals. We have covered the four basic interfaces: Parallel, BCD, HP-IB, and Serial and we have discussed specialized I/O; interrupt and DMA. Now that we have our devices talking, we will discuss how to overcome the language barrier. Next issue: character codes. ☒

# Character codes



by Steve Leibson  
Hewlett-Packard Company  
Desktop Computer Division

Language is quite possibly the most powerful component of civilization. Humans could not purposefully organize without shared language. Furthermore, the roots of all major human languages are verbal rather than visual.

Speech, our verbal use of language, would not be possible without the evolutionary heritage humans share that has produced our marvelously complex vocal tract, with lips, teeth, tongue, larynx and other organs we need to produce sound which others may understand. But the hardware of speech is not sufficient for shared understanding — a common language is also required.

## Computer's alphabet

This series has been discussing the hardware components with which

computers are built, allowing them to communicate with other machines. It is now time to discuss the languages computers use to communicate with other machines, rather than the equipment they use to do it.

## Wanted: standard code

As covered previously, digital computers use a binary language for their internal communication. There are several methods for representing data internally in a computer, however, and it would be advantageous if there were some standard language that computers could use for communicating with other equipment.

In addition, it is important that such a language be compatible with human communications, since some of the devices that the computer will be communicating with are intended to interact with people. Printers and CRT terminals are examples of this type of equipment.

## History of codes

The problem of creating a code, or computer language, that corresponds to an alphabet existed prior to the advent of computers. Even before electricity was harnessed for communications, man-made devices such as flags and semaphores were used to send messages.

Samuel Morse perfected the first code for electric data transmission, the Morse code. This set of dots and dashes is capable of representing the English alphabet and Arabic numerals so that intelligible messages may be interchanged between remote stations.

Early in this century, interest developed in replacing human telegraph operators with machines. Morse code was too difficult to mechanically decode, due to its variable length per character.

But the idea of a standardized code was retained. The dots and dashes evolved into the concept of bits. Each bit could either be a "1" or "0",



b7 b6 b5 Bits				0 0    0 0 <sub>1</sub> 0 <sub>1</sub> 0    0 <sub>1</sub> 1    1 0 <sub>0</sub> 1 0 <sub>1</sub> 1 <sub>1</sub> 0    1 <sub>1</sub> 1								
b4	b3	b2	b1	COLUMN ROW	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	\	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	\$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	/	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Mnemonic and  
Meaning<sup>1</sup>

NUL Null

SOH Start of Heading (CC)

STX Start of Text (CC)

ETX End of Text (CC)

EOT End of Transmission (CC)

ENQ Enquiry (CC)

ACK Acknowledge (CC)

BEL Bell

BS Backspace (FE)

HT Horizontal Tabulation (FE)

LF Line Feed (FE)

VT Vertical Tabulation (FE)

FF Form Feed (FE)

CR Carriage Return (FE)

SO Shift Out

SI Shift In

Mnemonic and  
Meaning<sup>1</sup>

DLE Data Link Escape (CC)

DC1 Device Control 1

DC2 Device Control 2

DC3 Device Control 3

DC4 Device Control 4

NAK Negative Acknowledge (CC)

SYN Synchronous Idle (CC)

ETB End of Transmission Block (CC)

CAN Cancel

EM End of Medium

SUB Substitute

ESC Escape

FS File Separator (IS)

GS Group Separator (IS)

RS Record Separator (IS)

US Unit Separator (IS)

DEL Delete

represented by either the presence or absence of an electrical signal.

## Shift codes

Such special codes were called shift codes because they shifted between different character sets. There were two five-bit codes that were in wide use, called Baudot and Murray.

led naturally to the first I/O incompatibility problems.

Character codes that relied on shift characters for proper operation were troublesome, because the interpretation of the incoming codes relied on the previous history of the message. Unless the receiving device knew which character set to use, there was a 50% chance of erroneous decoding.

## Modern codes

A single manufacturer could simply go out and invent a solution and expect the rest of the industry to follow. This was the route taken by IBM, which invented the EBCDIC (Extended Binary Coded Decimal Interchange Code) character code. EBCDIC is an eight-bit code allowing 256 characters to be represented. Since there aren't that many printable characters, there are some unused codes in EBCDIC.

The other method for obtaining a standard was through compromise in a committee. Other manufacturers did meet in order to develop a national standard called ASCII (American Standard Code for Information Interchange).

In addition, several codes exist to control the operation of the device receiving the message. Codes representing Carriage Return and Line Feed are evident to anyone who uses a typewriter. Other control codes

*Just as differences in language can create communication problems for humans, character code incompatibility can render an otherwise operable interface useless.*

include Form Feed, Bell and Horizontal and Vertical Tabs. These codes are clearly for control of various printing devices, although manufacturers of some products have used these codes for other machine dependent actions.

Finally, there are codes used to control how the receiving device will interpret subsequent data. There are two shift characters, called Shift In and Shift Out, used to switch between character sets (English letters aren't the only kind, you know). There are also control codes that delimit text; STX (Start of Text) and ETX (End of Text).

ASCII has been a very successful character code. Thousands of instruments and computer-related products use this code for I/O. Even IBM now offers equipment that uses ASCII. Several interfaces have been covered in this series, and all except the BCD interface may be used to transmit and receive ASCII code.

#### **Planning the escape**

The planners of ASCII tried to foresee as many different applications as possible. That is the reason for including the various control codes. They recognized that technology's advance could not be totally predicted and therefore gave themselves an escape clause.

One of the ASCII characters is called the "escape" character. This character designates that the characters following have a special meaning.

The intent in creating the escape sequence was to extend the range of the character set by selecting from a range of available sets. Graphics, nationalized character sets, and special application sets have been developed for selection with certain escape sequences. Escape character sequences allow for a much richer variety of characters than the simple shift in/shift out scheme of the five-bit codes.

The now common CRT terminal has provided the escape sequence its widest application, however. The inclusion of microprocessors in terminal design has greatly augmented CRT capabilities. The serial communications link to these terminals has not been changed in years. One data channel to the host computer is all that is available.

Ordinarily, any characters that are received via this channel are printed on the terminal screen. But capability for character and line deletion, display enhancements such as inverse video and underlining, and even control of tape drives in the terminal does not exist in the ASCII standard. The escape sequence allows for these new capabilities.

#### **Creativity**

Manufacturers of CRT terminals are now adding increased performance to their products via escape sequences. Unfortunately, since the actual effect of these sequences is not covered in the ASCII standard, the terminal designers have felt free to create their own standards.

For example, one major feature now found on most CRT terminals is cursor positioning. The ability to place a cursor anywhere on the screen directly is important for many types of form-filling applications. There are about as many escape sequences for performing this task as there are CRT terminal manufacturers.

They all work similarly. The host computer sends the terminal an escape character. This is followed by a second character indicating that the escape sequence is for cursor positioning. Two more characters follow, giving the X and Y positions for the cursor. Usually the sequence is self-terminating, meaning that four characters including the escape are all that the computer need send.

After receipt of the fourth character, the terminal performs the

action requested and prints any further characters received. Note that the ASCII characters in the escape code sequence are not interpreted as printing characters, but as control characters. The escape character has the effect of temporarily converting all ASCII characters to control.

#### **Code conversion**

The majority of computer equipment uses ASCII character representation today. Unfortunately, some of the older equipment still in use may not.

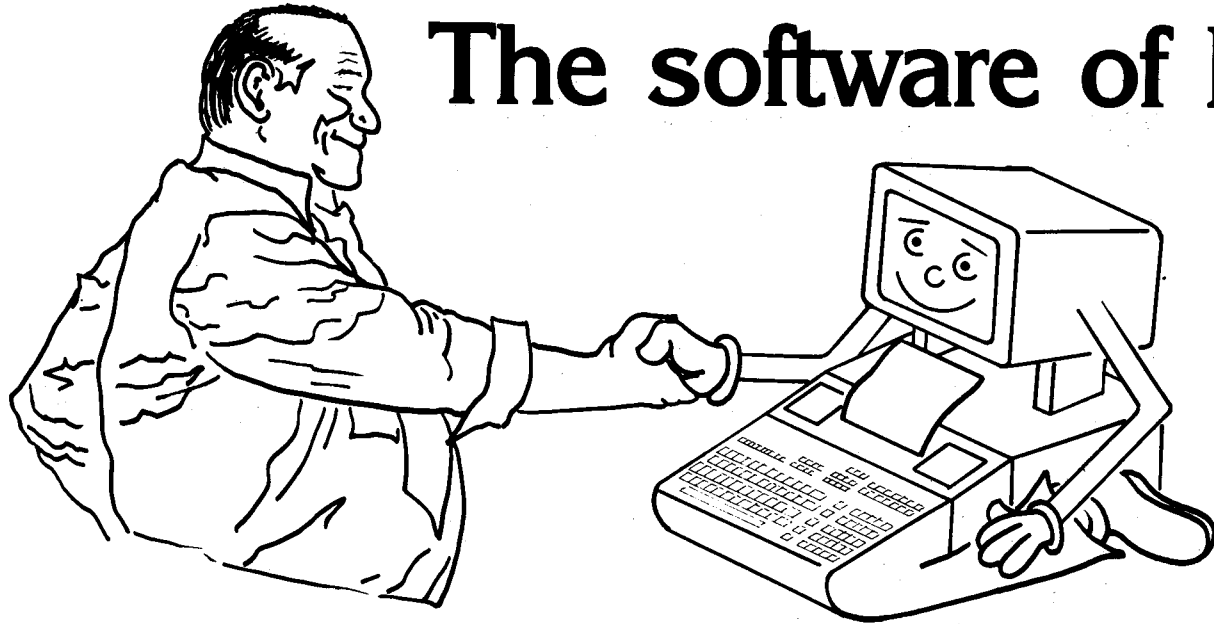
Interfacing to these devices requires that the ASCII characters the computer would like to send must be converted to the characters that the peripheral would like to receive. Here we are assuming the hardware interfacing requirements have already been met.

In addition, some modern peripherals may have odd requirements that can be met only through code conversion. An example is a printer that automatically inserts line feeds whenever it receives a carriage return. Unless the application calls for double spacing, the printout won't be as desired, since many computers send both carriage return and line feed to denote the end of a line of text.

One solution to this problem is to have the computer convert all line feeds to non-printing characters, such as "nulls." Most peripheral devices ignore the null character, which is the ASCII zero.

Character codes are yet another source of incompatibility in the world of I/O. Just as differences in language can create communication problems in humans, character code incompatibility can render an otherwise operable interface useless.

Fortunately, if the computer has a language that is rich in I/O capability, even this language barrier can be overcome. ☐



# The software of I/O

by Steve Leibson  
Hewlett-Packard Company  
Desktop Computer Division

One of the most critical parts of a computer system is the human operator. During the development of a system, the programmer will communicate with the machine through a programming language.

This language could be at the machine level, though that practice is growing less common. More likely, it will be at a high level, through a sophisticated programming language provided by the machine manufacturer.

After the system software is developed, people will use it for accomplishing tasks. The computer must be able to interface with these people in a concise manner. They are not concerned with the software running the system, but with the results produced. This article will address high-level software of I/O.

---

## Representing information

---

There are many different kinds of information in the world, such as prices, quantities, voltages, written documents, drawings and innumerable other forms.

Information is one of humankind's most powerful tools. One of the reasons that computers have become such a major factor in current human endeavor is their information processing power.

Yet for all this capability, computers can only store information

in two forms: numbers and non-numbers. There are no prices, inventory quantities or voltages in a computer memory. There are only the two forms with which a program associates these values.

There are no pages in my text editor program, but only character data that is processed by the software to print pages. The software of I/O is a tool that instructs the computer how to accept data from the outside world and how to provide internally stored data to the world.

---

## Storing information

---

We must first study how information is stored in the computer in order to use the software I/O. Numbers are usually stored in something called internal format.

The author of the programming language for the computer decided the best way or ways to represent numbers inside the machine. The ones and zeroes that make up the number are probably not easily recognized as a number. For example the number 21 in 16-bit binary is 000000000010101.

In addition, though we have many ways of writing numbers such as \$2.69,  $6.02 \times 10^{-23}$  and 3.14159, the computer will have only a limited number of numeric types.

The most common numeric types are integer and floating point. But in whatever type a number is stored, we want the computer to print the number out in the format that makes

the most sense to humans.

Bank tellers will laugh if our payroll program prints checks that read 6.02E2 dollars. We won't find it funny, however, when that number is interpreted as six dollars and two cents instead of six hundred two dollars.

---

## Formatting

---

Most high-level languages make it possible for the computer to input or output numeric values in the form desired. This is called formatting.

The capability may come as a format statement or as a format field within a statement that causes the computer to output information. In either case the format specification describes exactly how the number is to be output or input.

We may use the above check writing example to demonstrate formatted I/O. Suppose our program has the following statement in it:

210 PRINT Pay

That is a very simple program statement. The computer, being the simple machine that it is, will print the value of the variable Pay, in whatever format the computer is currently using.

If the machine is in fixed 2 format and  $\text{Pay} = 602$ , we get "602.00" printed, just what we want. If the machine is in fixed 0 format, we get "602", which is close. However, if the machine is in FLOAT 9 format, we get "6.020000000E+02". This last

*The best way to learn how to perform formatted I/O  
in a given language is to read the manual — several times.*

printout is not even close to being acceptable.

---

### Finding a solution

---

What can we do about this PRINT statement to prevent unacceptable output from the program? A first attempt might be to change the default format of the machine just before the print statement:

```
200 FIXED 2
210 PRINT Pay
```

This approach is taken by programmers who don't know about or don't want to learn about formatting their output. The disadvantage of this approach is that when the state of the machine is altered, all subsequent printing will be done in the fixed 2 format unless another FIXED or FLOAT statement is executed. We are also missing the dollar sign that precedes the number on the printout.

The program could be changed to:

```
200 FIXED 2
210 PRINT "$",Pay
```

Now we get "\$ 602.00" on the printout. Clearly the machine is just not understanding what it is we want.

The means for telling it exactly how the number is to be printed is the format field in the PRINT USING statement.

Now we can change the program to:

```
210 PRINT USING "A,000.00";"$",Pay
```

The printout reads "\$602.00" which is exactly what was desired.

Just as different computer languages have different statements for performing similar functions, format techniques vary widely from language to language. Even differing dialects of one language, such as

BASIC, may vary as to how formatted I/O is performed

The best way to learn how to perform formatted I/O in a given language is to read the manual — several times.

---

### Stringing things together

---

As mentioned earlier, not all data can be represented in numeric form. Text, such as magazine articles, is best represented as a linear array of characters. Such arrays are usually called strings. This data type is useful for storing letters, instructions and even command sequences for some instruments.

The previous article in this series discussed character codes. They are used to represent text data in a form that may be transferred from machine to machine. Each character is represented by five to eight bits. The most popular code is ASCII, which is a seven-bit code.

Eight bits is a very convenient size for data storage in most modern digital computers. Therefore, strings are usually composed of eight-bit parcels of data. Since ASCII is only seven bits, one bit of each string character is usually wasted.

Input and output of strings is much simpler than for numerics. The internal representation for strings is what the printout might look like — almost. The exception to this statement is the terminator.

Input of a string must stop at some point so that the data can be processed. The terminating character tells the computer when it has reached the end of the message.

A common default terminator is the line-feed character. It is so common that most input statements default to terminating upon receipt of a line feed. Most output statements automatically add a line feed at the end of a string output.

Just as with numeric I/O, everything runs fine until you don't want the defaults any more. At some time, you will have to read data in from a device that outputs carriage return as a message terminator.

Or perhaps you will have a printer that needs an ENQ character as a terminator instead of a line feed. Eventually, a situation will arise where the defaults won't work.

What can we do? It's time to use a format statement again. Suppose we have a device that requires only a carriage return as a message terminator. The program might contain the following statement:

```
200 PRINT A$
```

This program will output the string A\$ and follow it with the carriage return and line feed characters. Since the device we are outputting to will terminate one message on the carriage return, it will interpret the line feed as the start of a new message.

This may be suppressed by changing the program to:

```
200 PRINT USING "#";A$,CHR$(13)
```

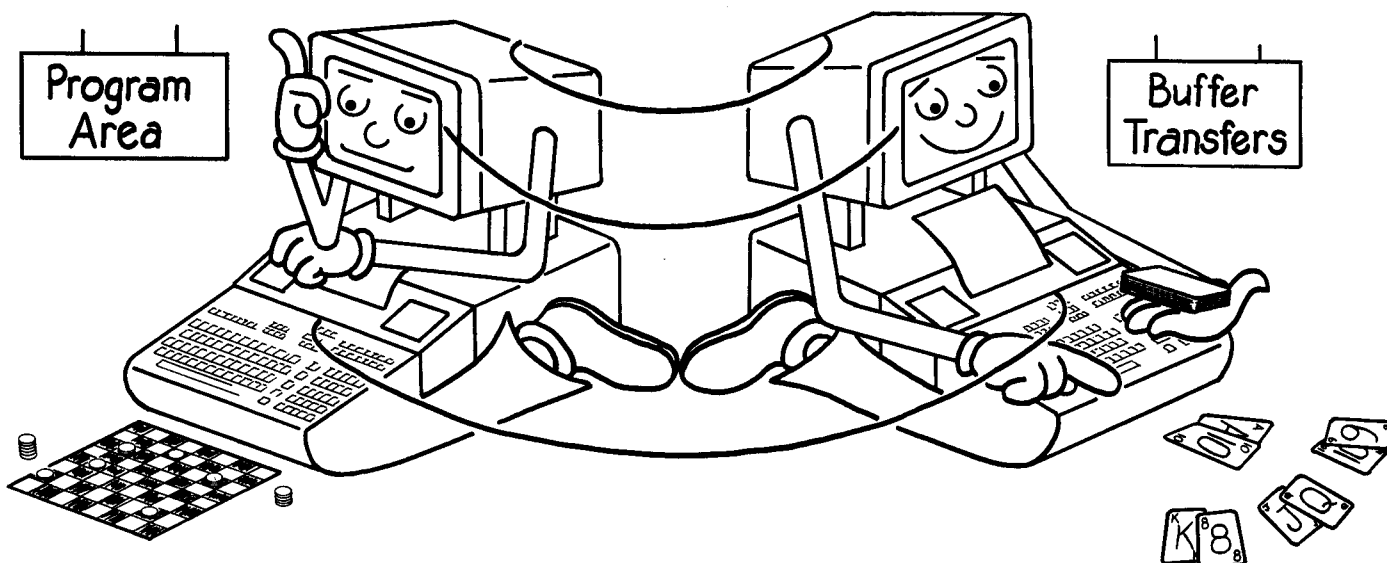
The "#" specifier tells the computer to refrain from adding any embellishments to the string being output. The CHR\$(13) is a carriage return, which is the proper terminating character. Again, the formatting capabilities of the language have allowed us to specify exactly what we want the I/O to do.

The software of I/O is an extremely important topic in interfacing. It is the interface between the computer user and the computer.

By understanding how to communicate system I/O needs through explicit software statements, a user can make a "dumb" computer work as the flexible problem solver it was intended to be.



# Interrupts and buffers



by Steve Leibson  
Hewlett-Packard Company  
Desktop Computer Division

In our discussions about I/O hardware, we considered the needs of a wide range of peripheral devices. Some devices are much slower than internal computer processes, some are about the same speed and some are faster than the computer can comfortably handle.

We discussed the three hardware handshakes associated with these three classes of peripherals. Slow devices are best handled by interrupt (see Jan/Feb 1980 issue). Only when the device is ready for another data transfer is the processor interrupted so that it can service the peripheral.

Medium-speed devices can interact with the processor directly, since they will not degrade system performance. High-speed devices require special hardware for Direct Memory Access (DMA) because the processor alone is not fast enough to service them (see Mar/Apr 1980).

The hardware to perform interrupt I/O and DMA is useless unless there is software to support the capability. In the previous article, we discussed formatted I/O and

referred only to the simpler handshake or programmed I/O. Most computers support this type of I/O even if it is only by using the PRINT statement.

Hewlett-Packard desktop computers support interrupt I/O in two ways: user interrupt service routines and buffer transfers. DMA is supported only through buffer transfers.

## Processor interrupts

High-level languages frequently have subroutine capabilities. In HPL, subroutines are invoked with the "gsb" statement. Return to the main program is accomplished using "ret". BASIC uses the corresponding statements GOSUB and RETURN.

User interrupt service routines are a variation of the subroutines. After interrupts are enabled, the subroutine is invoked because a peripheral interrupts.

The subroutine is written in the high-level language of the computer and is terminated with an interrupt return statement such as "iret" in HPL. The following HPL program fragment illustrates how user interrupt service routines are written:

```
10: 1→I
11: oni 6, "send"
12: eir 6
•
•
•
87: "send": wtb 6,A$(I,I)
88: I+1→I;if I<=len(A$);eir 6
89: iret
```

Line 10 sets a counter that points to individual characters in string A\$. Line 11 directs the program to line 87, labeled "send", when an interrupt occurs. Line 12 enables the interface hardware and software to accept interrupts.

Line 87 sends a single character from string A\$ each time the user interrupt service routine is called. Line 88 increments the counter I to the next character and re-enables interrupts if there are more characters to transmit. Line 89 forces a branch back to the main program.

## Getting bitten

There are several things to note from this example. The "eir 6" enables the interface. The meaning of an interrupt is that the interface is not busy. The first interrupt will occur

immediately after the computer executes line 12.

Novices at interrupt routines are always bitten by this the first time they write one. If the interface has not been made busy by sending it a character before interrupts are enabled, interrupt is immediate.

Note that a counter must be kept by the program to keep track of where the next character will come from in A\$. Also note that interrupts must be re-enabled in the interrupt service routine if the transfer is not finished.

This is necessary because the "eir" is canceled when it is invoked. That prevents the interrupt service routine from being interrupted.

### Buffers are better

High-level-language program lines are slow compared to the processor's machine code speed. Only low data rates can be supported with user interrupt service routines. Buffer transfers are a much better choice for data transfers, leaving user routines to service special situations.

Buffers are blocks of computer memory allocated for I/O (see Figure 1). Data passes through the buffer on the way into or out of the computer. Enabling of interrupts and character counters is automatic.

Data transfers can be terminated on a count as in the above example or by a character match for buffered input. The following example performs the same task as the first, but uses buffered I/O.

```
10: buf "OUT",100,1
11: wtb "OUT",A$
12: tfr "OUT",6
```

As you can see, this is much simpler. Line 10 creates a buffer of 100 characters, line 11 fills the

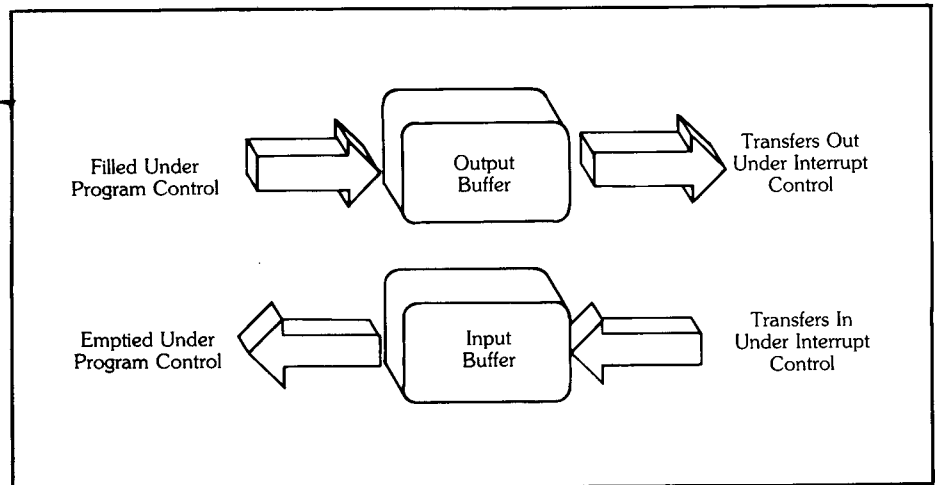


Figure 1.

buffer with the contents of string A\$ and line 12 sends the data to the peripheral. The 1 at the end of line 10 specifies an interrupt buffer.

Why is this technique superior to simply writing out the data directly to the peripheral? Line 12 only initiates the data transfer. After that process is started, the program will continue with line 13. When the peripheral interrupts, it will automatically be given the next character. Meanwhile, the computer is executing the rest of the program.

### End of the line

Interrupt buffers are faster than user interrupt service routines for one primary reason. The only safe place to interrupt a high-level language program is at the end of a line. In the execution of a line of high-level language code temporary locations are set up, addresses are calculated and a whirl of activity is taking place.

An interrupt routine must be able to return to where the program left off after the interrupt is serviced. If the user routine accesses variables being used by the main program, or worse yet, changes them, there could be disastrous results.

That is why high-level language interrupts are restricted to the end of a line. Things are safe there.

Conversely, the routines used by the buffer transfer interrupt service routines are in machine code and are restricted. Their affect on the system is well known because all they are allowed to do is data transfer.

Buffer interrupts are allowed any time they are enabled. Thus, interrupt buffer transfers can be much faster than user interrupt service routines for data transfer. They are also easier to use.

### Limit: one DMA

Once you understand interrupt buffer transfers, DMA buffers are easy because they work the same way. A buffer is set up, filled and transferred. The syntax is the same too. The only parameter that changes is the buffer type.

Only certain interfaces can support DMA transfers and only certain devices require DMA service. Since DMA requires special hardware, Hewlett-Packard desktop computers have one set of DMA hardware. Thus, only one DMA transfer may be active at one time.

Buffered I/O is a real convenience. It is another way of taking I/O hardware such as interrupt and DMA circuitry and making the capability available in an easy-to-use form. ☐



# How high is the ground?



by Steve Leibson

It is a paradox that of all the signal wires used in interfacing, the most complex is the one that seems the most simple.

Ground wires are usually ignored in the design of computers and interfacing circuitry. No signals are intentionally impressed on them.

Often, the number of ground wires in an interface cable is determined by how many conductors are left over after signal wires have been allocated. This type of interface design can lead to signal degradation, loss of data and even destruction of equipment.

Why do designers include ground wires in the first place? Electricity flows in loops. Current must always return to its point of origin according to the laws of physics.

If we want to send a logic signal

to a peripheral device we will be sending it in the form of a current. This current must have a return path of low impedance so that the full signal strength is observed by the peripheral device. Any impedance in the pathways will diminish the signal observed by the peripheral.

One reason to provide a ground is to supply a low-impedance signal return path. This type of ground is called a logic ground because it is associated with the logic signals.

---

## Plug problems

---

A second type of ground serves to ensure that the devices at either end are at or near the same potential. One of the "laws" of interfacing states that there are never enough sockets on a wall power outlet to supply a complete computer system. At least one device will be

plugged into another wall outlet several feet away.

Most computer devices now are sold with three-pronged power plugs and use the third wire of the power outlet as an earth ground. This "earth ground" is used as a safety ground to keep the voltage of exposed metal parts within strict safety limits.

Unfortunately, due to haphazard wiring practices, there may be several volts of potential difference between the third wire of one electrical outlet and the third wire of an electrical outlet only a few feet away in the same room.

This potential difference is usually not large enough to pose a hazard to humans but can be death to a computer system. Signal levels for most interfacing systems today are five volts. A potential difference of only two or three volts can destroy

all trace of a signal. A potential difference of twenty or thirty volts can destroy circuitry.

A safety or earth ground between devices can minimize this potential difference. One again, we seek the lowest impedance possible so that the potential difference is as small as possible.

### Ground tools

Now that we have good logic and safety grounds between our computer and our peripherals, we can relax, right? Probably not. Chances are we have created a ground loop.

Figure 1 shows a system with just such a problem. The computer and the peripheral are tied together with three grounds. There is a logic ground for the signal return, a safety ground for potential minimization and the third wire grounds in the power cords.

The safety and third wire grounds are connected together intentionally. That loop cannot be avoided. The logic grounds in both devices are connected to third wire ground which is common in computer design. Loops are formed between logic and safety, logic and third wire and third wire and safety grounds.

Two sources of problems exist for this system. First, current may be flowing in the third wire conductor due to a faulty or leaky device someplace else in the power system. This will cause a voltage difference at the two power outlets A and B. That is why we installed the safety ground, to add a low impedance path and minimize this difference.

### The third path

The current sees the dual paths of third wire and safety grounds and the voltage difference will indeed be

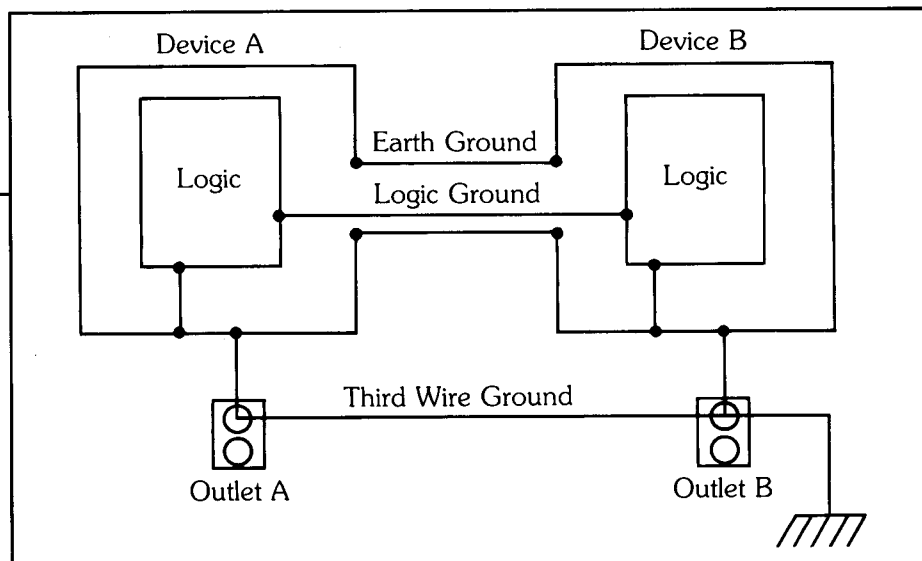


Figure 1

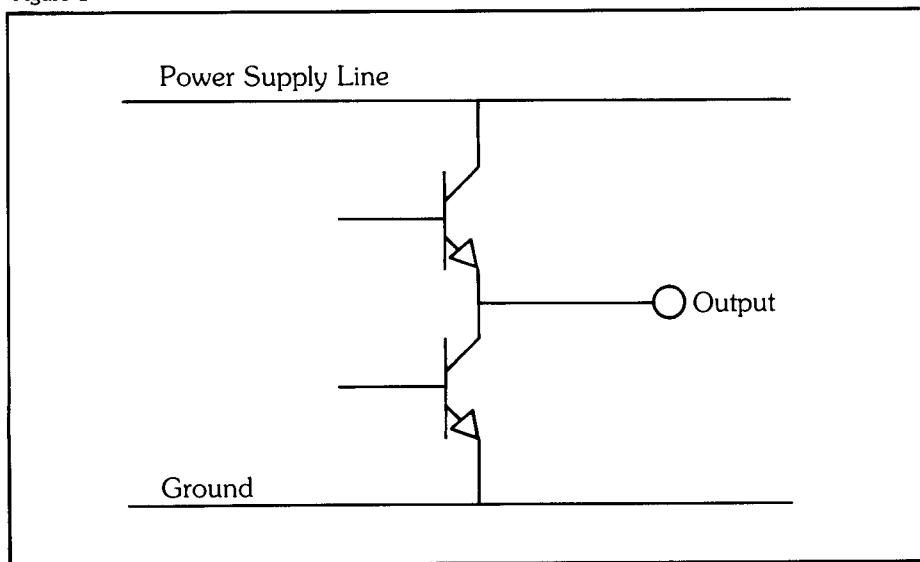


Figure 2

small. Unfortunately, the current will also see a third path to flow through, the logic ground.

Logic grounds are not typically designed to carry power fault currents. They have higher impedance. Thus a large current flowing through signal ground may prevent communications.

Since we put the grounds in to allow the logic signals to be received reliably, how do we prevent ground loops from destroying that reliability? The best method is to plug all devices in a computer system into one electrical outlet.

This assumes that there is

enough current capacity on that circuit to supply the computer and all of its peripherals with power. If there are not enough sockets on the outlet, use a power strip. The third wire ground in a power strip is short, well defined and will be of low impedance.

### EMI

Now that we have eliminated the effects of ground loops and our system is performing flawlessly, we can relax. Unfortunately, we notice that whenever the computer system is on, there is a lot of static on our

radio. Worse, our neighbor down the hall notices the same effect on his or her radio. Welcome to the world of electromagnetic interference (EMI); the second problem in interfacing grounding.

Figure 2 is a picture of the output stage of a typical logic circuit. There is a transistor connected between ground and the output signal line and another transistor connected between the power supply and the output signal line.

If both transistors are turned on at the same time, a large current will flow and destroy the circuit. If only the top transistor is turned on, the output voltage will be close to that of the power supply. If only the lower transistor is turned on, the output voltage will be close to ground potential. The signal is switched by changing which transistor is on and which is off.

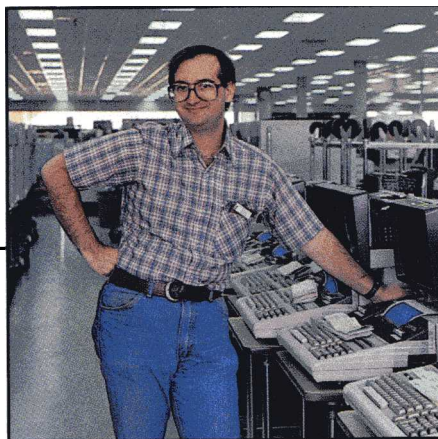
### Computing antenna

When this switching takes place, both transistors will be partially on for a brief period of time. One is partially on, going off and the other is partially off but turning on.

At this instant, a large current is allowed to flow from the power supply to ground through both transistors. This current spike will make the ground jump a bit through the small but finite impedance of the ground line.

There are literally thousands of these output circuits in a computer, switching constantly. All are adding their share of noise to the logic ground. This noise is carried out to the interface cable and over to the peripheral on the logic ground wires we ingeniously ran between the devices in our computer system.

The voltage spikes in the ground are too small to affect the interface logic signals but the interface cable



Steve Leibson joined the Calculator Products Division (now the Desktop Computer Division) of HP in June, 1975. Since then he has worked on a variety of hardware and software projects, all relating to interfacing of desktop computers. His products include the 9878A I/O Expander, the 98036A Serial Interface, the 98224A Systems Programming ROM for the 9825A and the I/O backplane for the System 45.

In 1980 Steve co-authored a book: *The Great Small Business Computer Ripoff* with a close friend, Bill Scott. He has published the book himself by forming a small company called Data Press.

Steve now works for the Auto-Trol Technology Corporation of Denver, Colorado.

Auto-Trol Corporation  
12500 N. Washington Street  
Denver, Colorado 80233  
U.S.A.

acts as an antenna and transmits this noise for all to receive. The thousands of output circuits team up to form a low voltage but high current signal. The actual logic signals are much lower current and don't cause as much trouble.

### Solutions

There are two solutions to this problem. The first involves the use of low impedance ground planes in the computer and peripherals to minimize the ground noise. The second is to shield the interface cabling to prevent the noise from escaping. Both of these techniques

are used in Hewlett-Packard desktop computers.

Finally, interface designers are attacking the ground loop and EMI problems using a new interfacing technology: fiber optics. Glass optical fibers carry modulated light signals between devices. There are no grounds, thus no loops. There are also no "antennas" to pick up and transmit noise.

Currently, fiber optic interfacing costs more than the conventional interfaces we have covered in this series. Some applications requiring long distance or good noise immunity are already using this interfacing technology. Many more applications will use fiber optics in the future. ☐

# An I/O Glossary For Hewlett-Packard Desktop Computer Users

by Steve Leibson, Hewlett-Packard,  
Desktop Computer Division

One of the most difficult problems encountered when entering a new technical field is that of jargon. Every discipline seems to have developed its own unique vocabulary, and the world of computer I/O (input/output) is no exception. To aid the computer user inexperienced in such matters, this I/O glossary is presented.

## A

**accumulator** a register inside the computer processor used to store operands to be operated upon and to receive the results of such operations. A computer may have several accumulators.

**alphanumeric** pertaining to a device, system, character set, etc. that is capable of representing letters and numbers.

**ASCII (American Standard Code for Information Interchange)** a seven bit code capable of representing letters, numbers, punctuation marks and control codes in a form acceptable to machines.

**analog** a characteristic that is continuous in form as opposed to digital, which is characterized by discrete levels.

**analog-to-digital (A to D) conversion** a process which quantitates an analog quantity and produces a digital representation of this quantity.

**APL** a high level computer language which is strongest in the procedural/algorithmic area. Specially developed mathematical operators are used.

**assembler language** a low level computer language used for implementing higher level functions. One assembler statement produces one machine instruction.

**asynchronous device** a unit which operates at a speed not associated with any particular portion of the system to which it is connected and is therefore not a time-critical component.

**asynchronous data communications** a serial I/O protocol in which each byte transmitted is self-sufficient and bears no exact time relationship to preceding or succeeding bytes.

## B

**background program** that portion of the resident computer program which is run when no immediately pressing needs exist in the system.

**base** the radix or number of characters in a particular number system. The decimal system is base 10.

**BASIC language (Beginners All-purpose Instruction Code)** a high level language which is particularly easy to learn. The American National Standards Institute (ANSI) has standardized a minimal set of BASIC. Hewlett-Packard has a set of BASIC statements that is compatible across a wide range of machines.

**baud rate (bit rate)** the rate in bits per second at which information is transmitted over a serial data link.

**BCD (binary coded decimal)** a four-bit system of coding the numerals 0 through 9, leaving the six most significant codes unused.

**benchmark** a test program used to compare the relative speeds of two or more systems.

**bidirectional lines** links between devices in a system that may carry information in either direction, but not both simultaneously.

**binary** a number system of radix 2, using the numerals 0 and 1.

**bit (binary digit)** a single digit of a binary number.

**binary synchronous (BISYNC)** a synchronous data communications protocol that is byte oriented.

**bipolar** an integrated circuit technology characterized by high speed, medium power and wide availability.

**bit rate** see baud rate.

**BPS (bits per second)** see baud rate.

**buffer, hardware** a register or set of registers used to temporarily store information, usually to act as a transition medium between a fast and a slow device.

**buffer, software** a location or set of locations in memory given a name by the resident program and used to hold information until it can be utilized.

**bus (buss)** a set of hardware lines that may be used to connect several devices together for communications purposes.

**byte** a group of eight bits.

## C

**card, interface** a device that converts a computer I/O bus into some standard I/O configuration (eight or sixteen bit parallel, BCD, RS-232, IEEE-488, etc.).

**character** one of a set of elements in a set used together with other elements in the set to express information.

**character set** a group of elements, which taken as a whole can express all of the information desired in a particular system.

**checksum** a quantity, usually following a string of characters, used in several error-checking algorithms.

**chip, integrated circuit** an electronic component comprised of a large number of basic devices all combined on a single silicon chip.

**CMOS (Complementary symmetry metal-oxide semiconductor)** a logic family of integrated circuits characterized by extremely low power, medium speed, wide availability and static discharge susceptibility.

**clock** a periodic signal used throughout a system for timing and synchronization.

**code, machine** the basic instructions of a computer processor.

**compiler** a program having a high level language program as its input and machine code as its output.

**complement, ones** the inversion of every bit of a binary number, i.e. all ones are changed to zeros and all zeros to ones.

**complement, twos** a ones complement plus one.

**compute bound** a program which is speed-limited by the computations being performed.

**control character** an element of a character set which may produce some action in a device other than a printed or displayed character. A character may become a control character in some systems by a special preceding character or set of characters.

**controller** the device in a system that dictates the occurrence of events in that system.

**control line** a line in a data link which causes information to be transferred.

**CRT (Cathode Ray Tube)** a popular display device used in computer systems to display multiple lines of text or graphics.

## D

**data bus** a set of lines for carrying data or characters between devices.

**data communications** generally taken to mean serial data I/O but may include any I/O between digital devices.

**data set** a device used to encode digital data onto voice phone lines. Also called a modem.

**data terminal** a class of devices characterized by keyboards and CRT displays.

**decimal** pertaining to the number system with ten numerals.

**digital** a quantized method of representing a quantity or information.

**digital-to-analog (D to A) conversion** a technique for converting a quantized representation of a quantity into a continuous signal.

**DMA (Direct Memory Access)** an I/O technique for transferring data between a device and memory without the aid of the computer processor. Special hardware is required to operate the memory independently.

**driver, hardware** a circuit used for impressing a signal on a conductor.

**driver, software** a program that is used to transmit information to a device using a device-dependent protocol.

**DTL (Diode Transistor Logic)** a logic family, compatible with TTL, now extinct.

## E

**EBCDIC (Extended Binary Coded Decimal Interchange Code)** a special IBM character set.

**emulator** a circuit or program that imitates another circuit or program in real time.

**erasable programmable ROM (EPROM)** an integrated circuit used to store programs or data which may be erased. Usually used in development work.

**exponent** the power of ten used in scientific notation.

## F

**fan in** the load a logic circuit input places on a signal line.

**fan out** a measure of the drive capability of a logic circuit output.

**firmware** a program placed into ROM. Hewlett-Packard places the operating systems of desktop computers in firmware.

**flag line** a line in a data link used to signal the status of a device.

**foreground job** a portion of a program that has highest priority and runs whenever possible.

**full duplex** a characteristic of serial I/O where data may flow between two devices in both directions simultaneously.

## G

**gate** the minimal logic element.

**GIGO (Garbage In Garbage Out)** the usual explanation for "Why doesn't my program work?"

**ground, earth or safety** a wire that is at earth potential, or at least is supposed to be.

**ground, logic** a level that is used as a reference for digital signals in a system. Not necessarily at the same potential as earth or safety ground.

## H

**half duplex** a characteristic of serial I/O where data may flow between devices in only one direction at a time.

**handshake** may be either hardware or software and characterizes a protocol for transferring information between devices.

**hardware** the circuitry in a system.

**hardware interrupt** a mechanism by which the computer processor may be interrupted from what it is doing to perform a more urgent task.

**Hewlett-Packard Interface Bus** the Hewlett-Packard implementation of the IEEE 488-1975 Instrumentation Bus used to interface multiple devices together with a well-defined hardware protocol.

**high-level language** a computer language characterized by powerful statements and highest ease of programming.

**HPL (High Performance Language)** a high level computer language implemented in the 9820, 9821 and 9825 Hewlett-Packard desktop computers. Characterized by extensive I/O capabilities.

## I

**IEEE (Institute of Electrical and Electronic Engineers)** a professional organization that has produced several I/O standards.

**initialization** a process which takes place whenever the state of a device or program must be known at startup.

**input** a process of transferring information into a computer.

**input/output (I/O)** a set of processes for information transfer.

**interface** the boundary between two devices or programs.

**interpreter** a program which executes a high-level language directly.

**interrupt** a disruption in the normal flow of a process.

**inverter** a logic element that outputs a one for a zero input and outputs a zero for a one input.

**I/O bound** a program that is speed-limited by the information interchange taking place between devices in a system.

## K

**k 1024** used in specifying memory size.

**K - 1000** used in specifying resistance and dollars.

**kluge** a concoction of hardware and software which is neither pretty nor producible.

## L

**latch** a logic device which is used for memory.

**LCD (liquid crystal display)** a display device characterized by extremely high visibility in high light levels and no visibility in darkness.

**LED (Light Emitting Diode)** a display device characterized by high visibility in darkness and less visibility in high light levels.

**logic** a group of circuits that perform Boolean arithmetic and memory functions.

**LSI (Large Scale Integration)** highly dense logic circuits on single chips.

## M

**machine code** the instructions executed by the computer processor.

**mainframe** the physical computer without devices attached by external cabling.

**mantissa** the significant digits of a number in scientific notation.

**mass memory** a device for semi-permanently storing data and programs in a readily retrievable form.

**MOS (Metal-oxide Semiconductor)** an integrated circuit process characterized by high density, medium speed and medium power.

**modem** see data set.

## N

**negative-true logic** a logic system in which the voltage representing a logical 1 has a lower or more negative value than that representing a logical 0. Most parallel I/O buses use negative-true logic due to the nature of commonly available logic circuits.

**network** a term used in data communications to describe a group of devices with varying degrees of intelligence that are interconnected to form a large system.

**non-volatile memory** a memory within a device that will retain information even when the device is switched off. Implementation is usually with ROM, PROM, EPROM, or RAM with battery backup.

**nybble** half a byte (four bits). BCD data is packed into nybbles.

## O

**object code** a program in machine code, the ultimate form that any program must be reduced to before it can run on a processor.

**octal** a base-eight number-representation system using numerals 0 through 7. Used in the creation of machine code programs and useful in visualizing bit patterns.

**ones complement arithmetic** a binary arithmetic system in which negative numbers are created by inverting individual bits in the binary representation of the positive number.

**open collector** a type of output structure found in certain bipolar logic families. The output is characterized by an active transistor pulldown for taking the output to a low voltage level, and no pullup device. Resistive pullups are generally added to provide the high level output voltage. Open collector devices are useful when several devices are to be bused together on one I/O bus such as IEEE-488-1975 (HP-IB).

**operating system** a systems program that provides the programmer with utilities including I/O routines, peripheral handling routines, and high-level languages.

**output** the act of providing information from a device to the outside world. Generally accompanied by a device that inputs the information being output by the first device.

**overlap** a mode of computer operation in which several processes take place seemingly simultaneously. In a multiprocessor system, simultaneous operation is truly possible. In a single processor system, processes timeshare the processor and appear to happen simultaneously while actually occurring in a time-sequential mode. In either case, real time savings can be realized, especially when extensive I/O to many devices of differing speeds is taking place.

## P

**packed data** information which has been compressed to make optimal use of memory. Four BCD digits can be packed in a 16-bit memory location.

**paper tape** one of the oldest, slowest and cheapest methods of storing archival information in a computer system. Data is stored in punched-hole sequences on a strip of tape.

**parallel I/O** the fastest, simplest method of interconnecting two devices using a minimum of circuitry. Data is transferred in a bit-parallel format, with the width of the interconnect bus generally equal to the computer memory width, in bits. Eight-bit buses are common, as they are ideal for character code transmission.

**parity** an error detection method used in I/O where noise is a possible problem. Parity is determined by counting the number of ones in the data word. Odd parity sets the parity bit so that the total number of ones sent is odd. Even parity sets the parity bit for an even number.

**peripheral** a device connected to the computer's processor and used to accept or provide information from/to the external environment.

**peripheral processor** a processor used to interface to external devices. Generally provided to increase program throughput by allowing simultaneous computation and I/O.

**polling** a technique used to discern which of several devices on an I/O connection requires service. In a simple form, the processor may periodically interrogate each peripheral device in order to determine the device's status.

**priority interrupt** an interrupt structure in which devices with higher priority may interrupt the servicing of devices with lower priority. In other systems, priority may only be used in the arbitration of simultaneous interrupts, disallowing interruption of an in-process interrupt service routine.

**program** a series of statements defining a process or procedure in some form that can be used by a computer.

**programmable read only memory (PROM)** a logic circuit which can be programmed once in a special PROM programmer and is used to store data and/or instructions that are invariant. Also comes in an erasable model called EPROM.

**protocol** a set of conventions for transference of information between devices. The simplest protocols define only the hardware configuration. More complex protocols define timings, data formats, error detection and correction techniques, and software structures. The most powerful protocols describe each level of the transfer process as a layer, separate from the rest, so that certain layers such as the interconnecting hardware can be changed without affecting the whole.

## Q

**queue** a list of processes to be executed in sequential order, information blocks to be processed in sequential order, or a mixture of the two.

## R

**random access memory (RAM)** a misnomer applied to read-write memory.

**read only memory (ROM)** a memory device in which the memory locations are set to fixed patterns when the device is manufactured. Used for invariant programs and data.

**read-write memory** memory that may be both stored into and read from by the attached processor. Used for storing variable programs and data.

**real time** operation of a system at a speed sufficient to perform the required tasks within the actual amount of time in which they must be performed.

**real time clock** a device which measures time at a rate consistent with the tasks being performed. Sometimes used for pacing the occurrence of events within a system.

**register** a device used for holding a piece of information to be processed or transferred.

## S

**schematic** a drawing showing the interconnection of circuits to form a device. Generally needed when interfacing two devices that are not plug-to-plug compatible and sometimes for those that are.

**SDLC (synchronous data link control)** a protocol specifying a layered approach to serial data communications.

**serial I/O** a type of interconnection in which information is transferred one bit at a time. The most common serial I/O hardware schemes are RS-232 and current loop. Both of these are pseudo-standards in that most interfaces implementing these schemes work similarly but are not necessarily plug-to-plug compatible.

**simplex** a unidirectional implementation of an I/O protocol.

**software interrupt** the interruption of a user-level program in response to the acknowledgement of a hardware interrupt by the operating system. In high-level language programs, software interrupts can safely occur only at the end of a program line.

**status** information pertaining to the current state of a device.

**status line** a simple method of representing some state of a device in an interconnection scheme.

**string** a set of characters ordered in some manner.

**strobe** a control signal used to effect information transfers at the hardware level.

**synchronous data communications** a serial I/O hardware protocol in which transmitter and receiver are synchronized to a common clock signal.

**synchronous device** a device that transfers information at its own rate and not at the convenience of any interconnected device.

**synchronous transfer** an I/O transfer which takes place in a certain amount of time without regard to feedback from the receiving device.

## T

**threshold** the signal level at which a change in logical state is encountered in a circuit, such as 1 to 0 or 0 to undefined transitions.

**transceiver** a circuit or device that is capable of both sending and receiving.

**transistor-transistor logic (TTL)** a logic family characterized by high speeds, medium power consumption and wide usage.

**tristate** an output configuration found in several logic families which is capable of assuming three output states: high, low, and high impedance. This feature is useful for interconnecting large numbers of devices on the same wires while allowing only one to control the levels of the lines at a given time.

## U

**universal asynchronous**

**receiver/transmitter (UART)** a logic circuit that converts parallel information to an asynchronous serial format, and serial information to a parallel format. Useful for connecting processors with parallel data buses to serial I/O lines.

**universal synchronous/asynchronous receiver/transmitter (USART)** a logic circuit that can interconnect a parallel I/O bus to either an asynchronous or a synchronous serial I/O line.

## V

**vectored interrupt** an interrupt scheme where each interrupting device causes the operating system to branch to a different interrupt routine. This scheme is useful for very fast interrupt response.

**voice channel** a transmission medium originally designed for voice (i.e. a telephone line). Modems can be used to impress digital information on these channels for long distance I/O.

## W

**word** the basic size of a piece of information in a computer system. Most current microprocessors have a word size of eight bits or one byte. Newer processors and minicomputers may have word sizes of 16, 24 or 32 bits.

## Conclusion

The terms given in this article do not form a definitive list of words used in the field of I/O. In addition, the definitions given are not necessarily universal. This glossary has been written only to acquaint the reader with the more commonly used terms encountered when trying to interface modern computer equipment. Whenever attempts are being made to get devices to communicate, it is always desirable to ensure that the human designers and users of these devices have communicated first.



For assistance call the HP regional office nearest you: Eastern 301/258-2000, Western 213/877-1282, Midwest 312/255-9800, Southern 404/955-1500, Canadian 416/678-9430. Ask for an HP Desktop Computer representative. Or write to Hewlett-Packard, 3404 East Harmony Road, Fort Collins, Colorado 80525.

