FEATURE

PC Hardware Interfacing Part 6

With parallel ports pretty well beaten into submission, this month we're going to look at the rudiments of that most vexing sort of interface, the serial port.

STEVE RIMMER

Last month we looked at the 8255 parallel port chip, a wonderful chunk of silicon which replaces all sorts of discrete logic. In the months to come we'll look at the details of actually using this chip in a real word PC interface. However, before we get into the grotty details of wiring up one of these monsters, it makes some sense to look at the other sort of commonly found PC interface, the serial port.

Serial ports are equally as common as parallel ports on most PC's, and in many respects they're more useful. A greater number of peripherals are equipped with serial ports... excluding the exotic boxes and trolls, serial ports allow us to talk to things like laser printers, mice, modems and other PC's.

The important thing to realize about serial ports is that they allow data to flow in both directions, whereas parallel ports only send data out of the computer. In theory there is no reason why this should be so, but the conventions of personal computers at the moment do not use the standard parallel interface for bidirectional data transfer.

Actually, there is one good reason. A standard serial port allows data to flow in both directions at once. A standard parallel interface... even if it's tricked into accepting input as well as output... only allows for data flowing one way at a time. Changing directions would require some sort of handshaking to make sure that both ends of the conversation don't start talking at once.

If you're designing a custom interface for a custom peripheral, of course, you are free to use whatever sort of port you like. However, there is a lot to be said for using an interface design which already exists.

The other useful aspect of serial ports is that they can move data over fewer wires. A parallel port cable is an ugly great ribbon with all sorts of data lines, handshaking lines and so on. Even the simple ports we've looked at in this series have required eight data lines, a ground line and usually a handshaking line. Unidirectional data over a serial link requires but two wires. Bidirectional data can get by with three, although at high speeds it's very often desirable to have more than this.

This month we'll take a look at the basics of serial ports and have a quick introduction to the chip that makes them possible.

Bytes Bitten Sideways

Figure 1 shows a basic serial port. Now, this looks ridiculously crude, being all mechanical, but this is actually how serial data was transmitted back thirty years ago, before large scale integration was particularly common.

As we've seen in previous months, we can represent a byte of data as eight wires, or lines, each line having either zero volts or some arbitrary positive voltage on it. The zero volt ones are said to have logic values of zero, and the positive ones logic values one one.

If we use a mechanical switch like the one shown here to step the values on the wires onto a common line one bit at a time, the result will be serial data of a sort. In theory, a similar switch at the far end of the serial line could re-assemble the serial byte into a parallel byte.

In practice, there is a lot more to it than this. The two mechanical switches would have to be synchronized somehow, such that

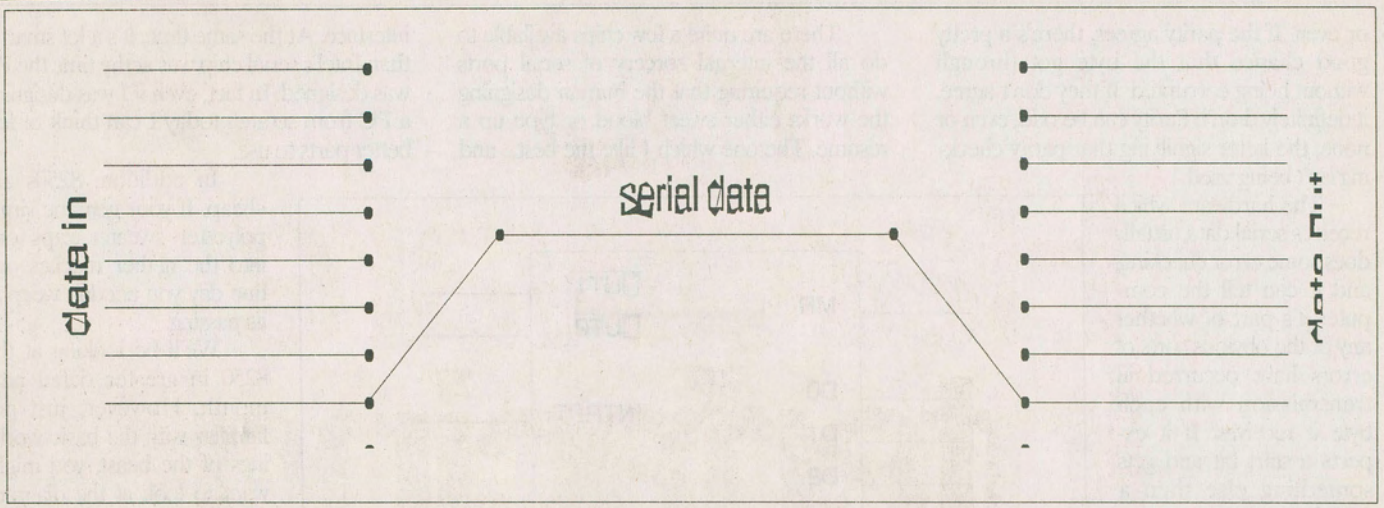

Figure 1. A very crude serial link. Parts for this project are no longer available... fortunately.

when the sending switch was sending bit zero, the receiving switch would be connected to line zero of the receiving parallel port. One could arrive at this by having a long metal rod between the two switches, such that they both rotated together, although the usefulness of a serial link seems to fall apart at this point.

In the real world, serial data is not generated mechanically, of course, or at least, it hasn't been for a very long time. In using data levels which come out of transistors, it's possible to generate a much more sophisticated, and much faster, serial data format.

In order to allow the receiving end of a serial conversation to know what's going on at the sending end, it's necessary to establish a "protocol" which defines how the data is going to be sent. The protocol defines a number of things about the structure of the serial bits. This is what a serial protocol tells us.

- Is the transmission synchronous or asynchronous?
- What is the speed (baud rate)?
- How many bits in a byte?
- How many stop bits in a character?
- What parity is being used?

This probably doesn't make a lot of sense just yet. Read on.

Baudy Tales

The first question in the above list, whether the transmission is synchronous or asynchronous, is one which users of PC's rarely have to answer. The serial port which is commonly found on a PC, the RS-232C port, is always asynchronous. Synchronous serial transmission involves sending not only data over the serial link, but also a common clock signal, such that the sender and the receiver agree on exactly when each pulse on the data line will begin and end. A synchronous serial port can be much faster for this reason, although it requires more hardware and more wire to move synchronous data around. The sorts of speeds which asynchronous ports are capable of are usually suitable for microcomputers, and one usually only sees asynchronous ports on PC's if they're intended to talk to mainframe computers.

You may want to implement a synchronous port in a custom interfacing application if you have to move a lot of data around in a large hurry.

The speed of data through a serial interface is called the baud rate, a term which you'll probably have heard before. The baud rate is the number of bits of data which move through the interface in a second. As we'll see in just a moment, this is usually more than eight bits per character.

The baud rate can be any speed you like, although there are a number of fairly common standard baud rates. These are shown here.

45.45 baud - old Baudot teletypes
300 baud - very slow modems
1200 baud - less slow modems, some printers
2400 baud - only mildly slow modems
9600 baud - fast modems, laser printers
14,000 baud - USR HST modems
19,200 baud - fast laser printers
56,400 baud - intersystem hard wired links

For practical purposes, 56,400 baud is the upper limit of the hardware in a standard PC serial port. This may also be pretty close to the red line for many old style first generation PC compatibles which, even if they were equipped with hardware which could handle faster data, would have a hard time keeping up with it.

The number of bits in a byte will usually be seven or eight. Eight bit bytes are required if we wish to transfer programs or other full byte characters though the port, and virtually all serial communications are handled with eight bit characters. Seven bit characters are used when all we want to send is straight ASCII, which only has seven significant bits. This means that we save one bit per character. If a character has ten actual bits, this represents a ten percent increase in the throughput of the port.

The aforementioned old Baudot teletypes only required sixty four possible combinations for their data, so they had five bit characters.

In order for the receiving port to know when a character's bits start, it's necessary to "frame" each eight bit character with a start bit and one or more stop bits. These framing bits are electrically different from data bits, so that if the integrity of the serial line between two devices gets temporarily mangled, the receiving port can resynchronize itself by simply throwing away all the bits it gets until it identifies a start bit.

There is always a single start bit, so we needn't speak of this. There can be one or two stop bits.

Finally, each byte sent over the link has a parity bit. This is a check to allow the receiving port to know whether the character it has received is what was actually sent. Actually, it's more of an educated guess. The parity bit simply tells the receiving port whether the sending port thought that the ASCII value of the character sent was odd

or even. If the parity agrees, there's a pretty good chance that the byte got through without being corrupted. If they don't agree, it definitely didn't. Parity can be odd, even or none, the latter signifying that parity checking isn't being used.

The hardware which receives serial data usually does some error checking, and it can tell the computer it's part of whether any of the obvious sorts of errors have occurred in transmission with each byte it receives. If it expects a start bit and gets something else then a framing error has occurred. If it doesn't find the parity it expects then a parity error has occurred.... and so on. It's up to the computer to take the appropriate action to deal with the these errors.

Usually there is no appropriate action. Data which has been mangled cannot usually be unmangled.

A common protocol is 9600 baud, eight bits of data, no parity and one stop bit for ten bits per transmitted character. This means that, at this speed, 960 actual bytes of data would move over the link in a second. It would take less than three seconds for all the raw data on a text screen to be sent over the serial port. A stock 4.77 megahertz PC redraws its screen through BIOS calls at less than this speed.

Chips From Hell

If you've been mentally designing circuitry to handle all the stuff we've just been talking about, your head will now be full of wires. Compared to the relatively tame hardware of a parallel port, a serial port is an engineer's nightmare. The only reason that hardware designers were ever able to implement them before dedicated serial port chips became common was because they'd have been fired if they hadn't.

There are quite a few chips available to do all the internal sorcery of serial ports without requiring that the human designing the works either sweat blood or type up a resumé. The one which I like the best... and the one which is found in the IBM PC's standard asynchronous serial port circuitry... is the Western Digital 8250. This chip has the advantage that it looks very much like an Intel part to the hardware which is driving it... all the rest of which actually comes from Intel in a PC... so it's wonderfully simple to interface. At the same time, it's a lot smarter than Intel's serial chip was at the time the PC was designed. In fact, even if I was designing a PC from scratch today I can think of few better parts to use.

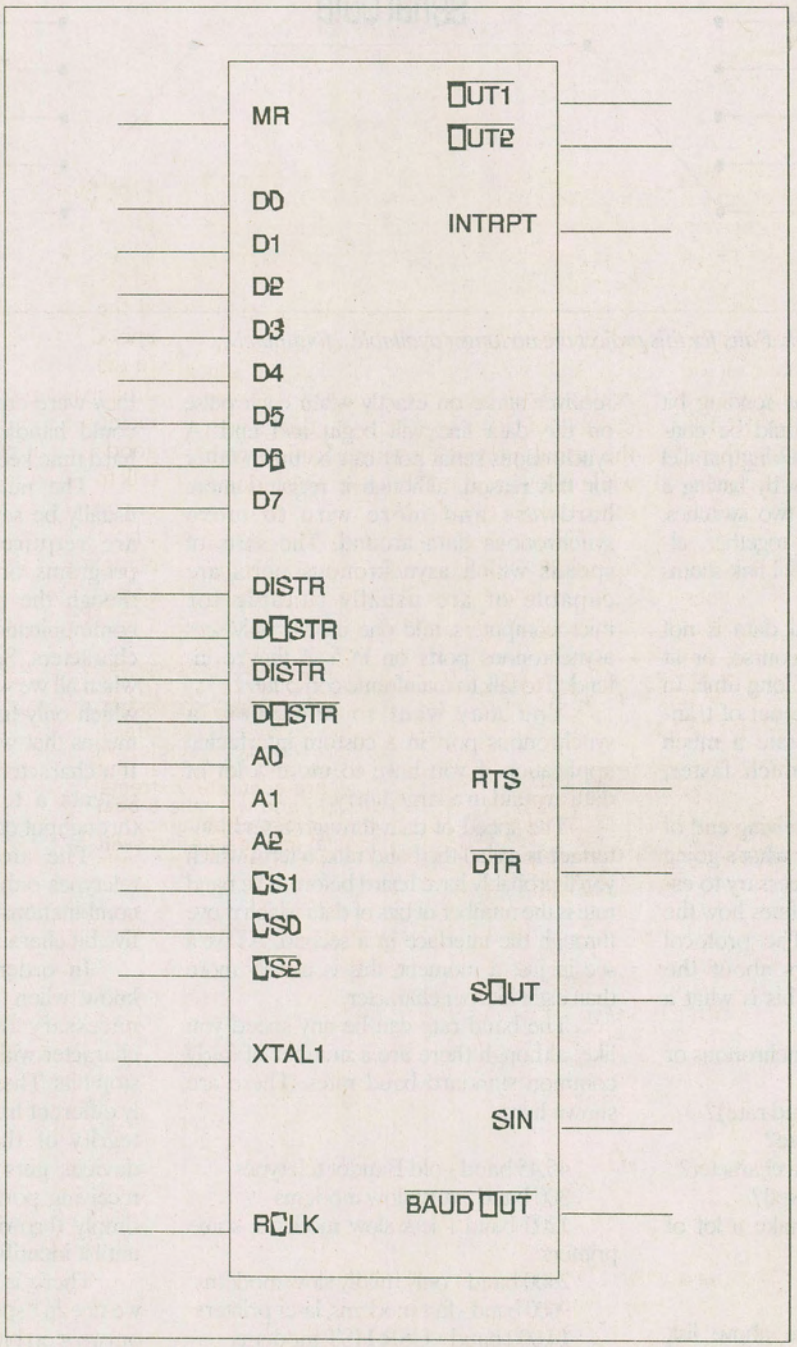

Figure 2. A simplifiedprintout of the B260, suitable for framing.

In addition, 8250s are cheap. If your genuine virgin polyester sweater zaps one into the nether reaches one fine day you needn't weep at its passing.

We'll be looking at the 8250 in greater detail next month. However, just get familiar with the basic workings of the beast, you might want to look at the diagram in figure two. It's not as horrible as its seems. The wonderful thing about the 8250... especially when you start actually trying to write software to drive one... is that it can be programmed quite crudely and then improved upon once you're sure it's working.

We'll leave the programming of the chip for another time.

If you peer at it for a moment, most of the 8250's pin consignment will prove to be pretty familiar. It does things in much the same way that we saw the 8255 parallel port chip doing them last month. The address and data lines behave in the same ways... the 8250 uses a range of ports to communicate with its host computer, just as the 8255 did. The lines which may require detailed explanations are few.

The MR line is the master rest. Pulling this line resets the 8250 to its default state. Since the chip comes up, when power is first applied to it, with garbage in its registers, it is essential that it be reset before anything sensible is expected of it. The RESET line of the PC's bus handles this.

The DISTR and DOSTR lines... and their negated complements... are the data direction lines for the chip's communication with the PC's bus. They come in both polarities because the 8250 was designed to interface to as many different processors as possible. The PC uses the negated

lines... the positive going versions are tied low. When the processor wants to write data to the 8250, it pulls $\overline{\text{DISTR}}$. When it wants to read data, it pulls $\overline{\text{DOSTR}}$. These correspond to the $\overline{\text{IOR}}$ and $\overline{\text{IOW}}$ lines of the PC's bus.

The CS lines are the chip select. Again, they come in both flavours to facilitate painless interfacing to different sorts of processors. The positive lines... CS0 and CS1... get tied high through a small resistor on a PC. The inverted line, $\overline{\text{CS2}}$, enables the chip when its range of ports has been addressed. You might want to look at the port address decoder in the first installment of this series to see what this is about.

The XTAL1 line is, obviously, the input from a crystal clock. To make all the baud rates work out conveniently we use an 18.432 megahertz clock. The RCLK and $\overline{\text{BAUD OUT}}$ lines get tied together... this facility allows for an external frequency divider in the baud clock loop, which is ordinarily not required.

The SIN and SOUT lines are serial data in and out. The $\overline{\text{DTR}}$ and RTS lines are involved with serial hardware handshaking, which we'll speak of at greater length later on.

Finally, there are two really useful sets of lines which make the 8250 such a powerful chip. The INTRPT line can be programmed to go high every time the 8250 has a character to received, every time it has nothing to do and could be sending a character and every time it encounters an error. Assuming that it is set to blast off for all of these conditions, an internal register in the 8250 allows the computer deriving the chip to know which condition has triggered any particular interrupt. The result of this is that the software driving the 8250 can be made wholly interrupt driven.

You will recall our speaking of this previously. It makes the driving software capable of doing sophisticated things with the serial port it's communicating with while allowing the PC to continue doing other tasks at the same time.

Finally, the lines $\overline{\text{OUT1}}$ and $\overline{\text{OUT2}}$ are general purpose I/O lines which the 8250 provides us for controlling anything we feel like controlling. The state of these lines can be programmed using the chip's internal registers. This allows the 8250, for example, to manually pulse dial a phone line, flash an LED when it encounters an error, turn on a fourteen megawatt siren when a serial connection has been broken... through a suitable fourteen megawatt relay, of course... and so on. These come in really handy in dedicated interface applications.

Hot Serial

Once you understand how the 8250 works, it's pretty simple to actually make it work when interfaced to a PC. We'll be looking at the hardware to do so in an upcoming episode of this long and complicated tale. In the mean time, you might want to start dreaming up some things that need interfacing to. By now, you should have a pretty good idea of exactly what a PC is able to talk to. Short of yaks and some rare species of partially extinct carburetors, there isn't a whole lot which *can't* be interfaced to a computer if you can concoct a reason for doing so. ■