PC Hardware Interfacing Part 9

This month we'll carry on with the interfacing of the 8250 chip, seeing how to give it interrupt capabilities, among other things.

STEVE RIMMER

In the last installment of this series, we started to examine the circuitry which interfaces the popular 8250 serial port chip to a PC's peripheral bus. As we've seen before, the first step in any interfacing project for the PC is to decode the I/O addresses. With this safely behind us, we can have a look at some of the additional glue that's required to make the chip run.

In looking at a schematic of a completed serial port for the PC... we're still not there this month, although we're getting closer... it may seem that there are just wires everywhere, and that very little of the circuitry makes any obvious sense. One imagines that the whole thing really evolved out of the chip manufacturer's application notes. In fact, this is not the case... the 8250 predates the first PC's, and, unless the notes have changed recently, they don't mention anything about associating the 8250 with an 8086 series processor.

Just as it's possible to design the 8250 interface, as we're doing, wholly from a functional understanding of the chip and the bus it's to be interfaced to, so too can you read the competed schematic in the same way. The easiest way to understand anything with a large scale integration device in it is to understand the device and work your way backwards.

The 8250's support circuitry makes fairly decent sense if you stand in the middle and look outwards.

Address Unknown

As of last month, our 8250 design will know when it is being addressed by the lower potion of the address bus. It does not know when the address is valid, however, nor does it know whether it's intended to read, write or shut up when it sees one of its addresses. To solve these problems, more circuitry is required.

You probably could have guessed that.

The first problem is to make our card distinguish between, for example, this operation:

```
MOVDX,03F8H
OUTDX,AL
```

and this operation:

```
MOVDX,03F8H
MOV[DX],AL
```

In the first case, data will be sent to the port 03F8H, which is, in fact, one of the ports our card is decoding. If this happens, we want our card to sit up and do something about the goings on of the bus. In the second case, data is simply being written to location 03F8H of the current memory segment, which has nothing to do with serial port I/O.

In both cases, however, the address 03F8H will appear on the lower portion of the address bus.

The way the processor differentiates between these two operations at the hardware level is through the use of its control lines. In the first case, it would pull the IOW line. In the second, it would pull the MEMW line. By watching the former line, our card can decide whether it should be doing something with the number on the address bus.

The 8250 is capable of watching the IOW line directly. Obviously, when the processor wants to send data to a peripheral device... pulling IOW... the 8250 should read the data. To this end, it provides two read enable lines, DISTR and **DISTR**. We'll be using DISTR, and tying DISTR to ground. The two polarities of this function exist in the 8250 because it was designed a a generic serial port chip. Other processors might need a line going the other way, and they could use DISTR rather than DISTR and an inverter.

Likewise, the processor will pull IOR when it expects data to come from the 8250. In this case, we will use the write enable line of the chip, DOSTR. This, too, also comes in a reverse polarity version, DOSTR, which we'll tie low. DOSTR connects directly to the IOR line of the PC's bus.

Interruptus Once Again

We won't get into the programming of the interrupt capabilities of the 8250 for several months, but we have to wire the beast up now. Its powerful capabilities for

generating interrupts as a result of serial conditions is one of the things that makes the 8250 such a useful chunk of silicon.

The biggest problem with asynchronous serial data is that you never really know when it's going to show up. The chip itself can only buffer a single character at a time... not much of a buffer, really... which means that if you don't read the current byte out before the next one arrives, you can kiss it farewell, as it'll get overwritten.

The easiest way around this is to check to see if the input buffer of the 8250 is full at least as often as characters can arrive, allowing for the maximum speed of data transmission at whatever baud rate you're using. This has a number of drawbacks, not the least of which being that at high baud rates most of the processor's time will be taken up "polling" for data.

If you write a simple terminal program using polled communication with the 8250 and the BIOS to write to the screen, a straight XT will lose data at speeds above twelve hundred baud.

The other approach, as we've discussed in previous articles, is to use an interrupt driven strategy. In this case, the 8250 is programmed to pull its INTRPT line every time a character appears at its input buffer. This line causes one of the eight hardware interrupt lines of the PC's peripheral bus to be pulled, which causes the equivalent of a software interrupt to happen within the PC.

Assuming that the owner of the card has had the sense to install a suitable interrupt handler of some sort prior to programming the card to throw interrupts, the processor will leap to a routine to fetch the character from the 8250 and store it somewhere... presumably in a more capacious buffer... for later processing.

This approach means that the processor only has to devote as much time to actually processing the incoming serial data as there is data to warrant it... plus a bit of overhead for the interrupt handler code. It also means that whatever else is happening in the foreground of the computer can go about its life without really knowing that the serial port is busy. We'll look at the actual software architecture of this a bit later on.

The 8250 can generate interrupts for several reasons, and it has an internal register which an interrupt handler routine can look at to determine which purpose a given interrupt is intended to serve. Interrupts can be thrown by the 8250 because of a waiting incoming character, as we've seen, because of one of several error conditions or because the chip is free to send

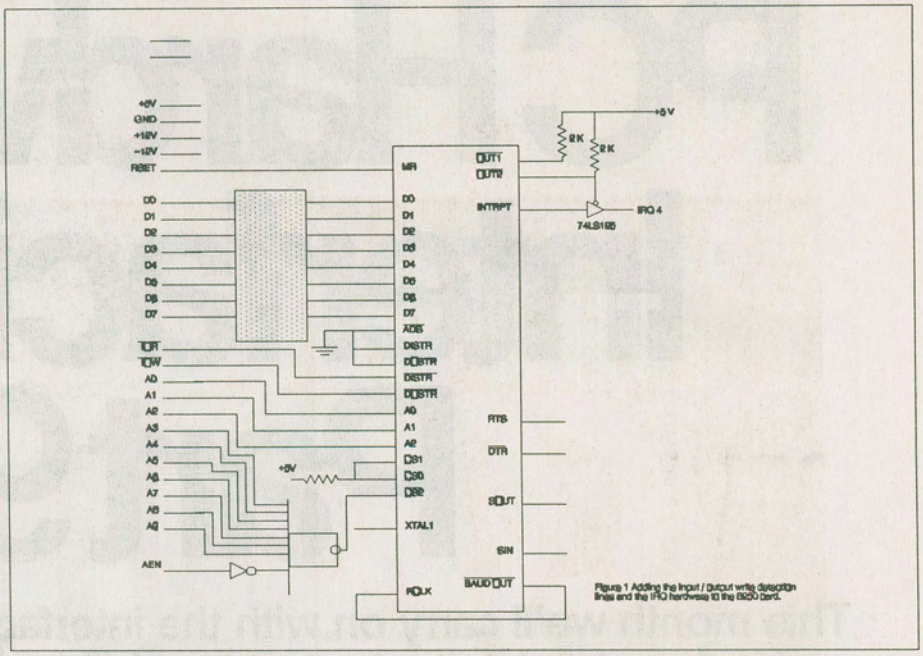

Figure 1. Adding the Input/Output write detection lines and the IRQ hardware to the 8250

a character, that is, because its output buffer is empty. This latter function may seem a bit obtuse. It's actually very useful in writing programs which communicate wholly in the background, as the chip itself can determine the maximum data transfer rate. The only software that's involved is an elaborate interrupt handler.

There are two criteria to consider in interfacing the INTRPT line of the 8250 to the PC's bus. The first is that it would be a good idea if it actually worked, and the second is that it would be a uniquely bad idea if it worked when it wasn't supposed to. This latter problem exists if the 8250 happens to throw an interrupt when it no handler has been installed for one... such as when the machine is powering up, or when the 8250 hasn't been initialized or re-initialized after an application has quit.

It is highly desirable to "gate" the interrupts that the 8250 throws.

One of the other neat features of the 8250 is its provision for having two independent output lines sprouting out of it. These lines simply reflect the status of two bits of an internal register of the chip. I suspect that they were included by the designers of the thing so that software driving the chip could dial phone numbers on old style, non-intelligent modems... with suitable timing code, you could pulse the phone line with a couple of transistors and fool the phone company into thinking your modem was actually a funky old rotary dial phone.

In our case, we're going to use one of these to drive a gate, such that only when the $\overline{\text{OUT2}}$ line is deliberately pulled will interrupts generated by the 8250 be allowed to make it through to the bus.

On a standard serial port card, the other line, OUT1, is tied high and never used. However, it has all sorts of possibilities, and you might want to experiment with it later on. As a really simple exercise, try hanging a transistor and an LED off it and writing a program to make the LED flash. Actually, this can be quite useful, and when we actually write a driver for our card you can set the LED up to flash when data comes in, making the card a bit easier to debug.

There are two hardware interrupt lines on the PC which are dedicated to serial communications. These are IRQ 3 and IRQ 4. The primary port range, starting at 03F8H, corresponds to IRQ 4. When a hardware interrupt finds its way to IRQ 4, the equivalent of an INT 0CH instruction happens inside the PC. The secondary serial port range, starting at 02F8H and using IRQ 3 generates an INT 0BH instruction.

Hard a'Port

We have now worked our card up to the point of its being able to talk to the PC reasonably well. We could program it and have it behave just like a real serial port... with one notable exception. It can't actually talk to the outside world, being as yet unequipped with an interface to a real RS-232C connector. We'll be having a peer at some of that hardware next month. ■