PC Hardware Interfacing Part 4

Deeper and deeper into the mystery of port addresses.

STEVE RIMMER

In the first two installments of this series, we looked at how to decode the PC's address bus and access specific ranges of hardware ports. We're going to get into this area a bit more this month and actually talk about getting useful data out of the PC.

There are those who will hold that this is a largely quixotic task, and that nothing useful has ever come out of a PC, or ever will. There are days when I'd be wont to agree with this.

One of the most interesting things you can do with a PC when you're first getting into designing hardware for one is to get it to make something happen external to itself, preferably something other than what it could normally do with its existing peripherals. Having it drive a printer, for example, is not very impressive — it's been done. Getting it to turn on all the lights in your house is a bit better. Having it locate and summarily vapourize all the cats in a 20-block radius of your digs is getting up there. When we get done learning how to interface to a PC, perhaps we could do a laser death ray project.

This month we'll press on ever closer to getting useful information out of the PC.

Toad in the Hole

In our previous look at the circuitry of PC interfacing, we saw how to recognize when the processor wanted to talk to a port, and how to narrow down the port in question to a range of 32 consecutive ports. In the example we looked at, our hardware decoded ports in the range of 300H through 31FH. We'll stick with this range here, although, as you'll recall from the second installment, the base port range can be switch selectable, as it usually is on commercial cards.

Having decoded the port address down to something manageable, it would be exceedingly handy to be able to tell if a single port in that range is being addressed. This will be our first task for this month.

Given that the processor is talking to a port in the range of 300H to 31FH, the number of the port within that range can be expressed in five bits. In fact, it is. It's the binary number that lives in the lowest five address lines. As such, if we were to take the data on these lines as a number at the time that our board is being addressed and add it

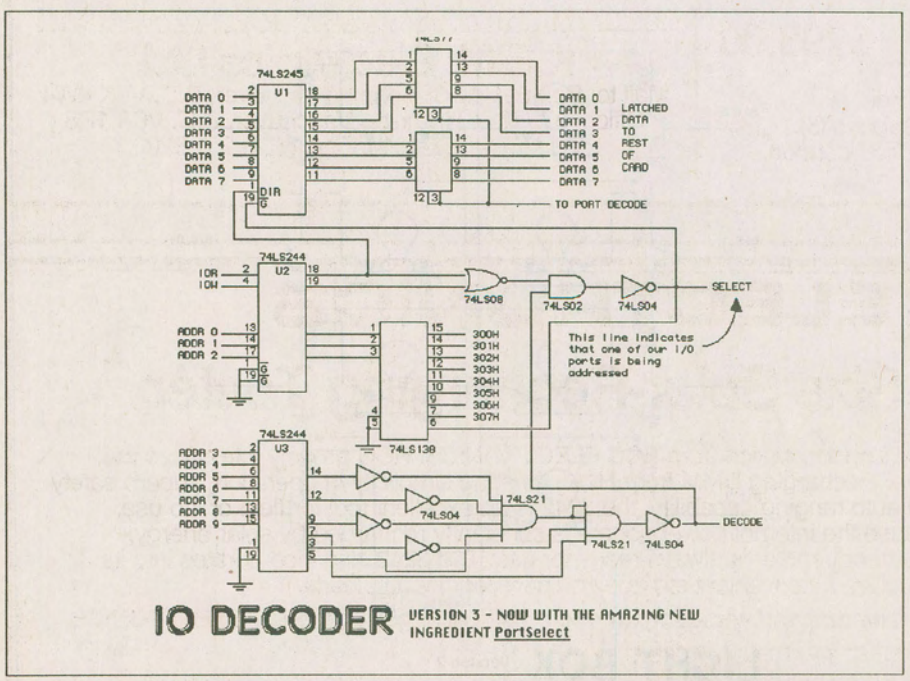

Figure one. This is the latest version of the peripheral card. It features single port decoding and a data latch. The eight numbered outputs of the 74LS138 represent port select lines. One of these, corresponding to the port we want to read from, should be connected to the combined latch lines of the 74LS77's at the top of the diagram.

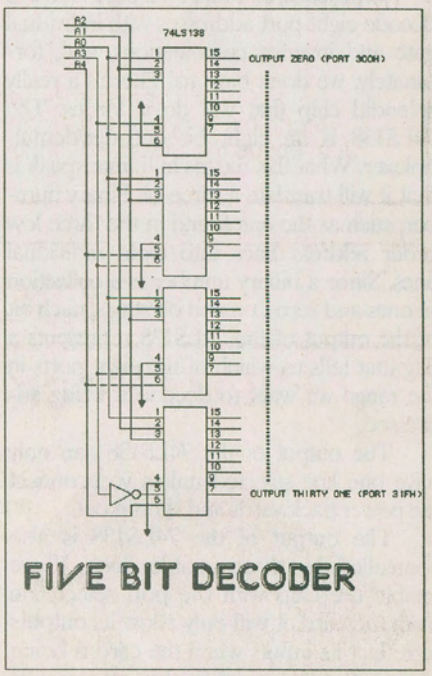

Fig. 2. This is a five bit decoder using four 74LS138's. It allows you to select among thirty two ports... assuming you can find a use for 32 unique ports on your card

and add it to the base port address of 300H, we would know the actual port within the range that the processor is trying to talk to.

For the purposes of this discussion, and to keep the circuit diagram accompanying this feature down to a manageable hugeness, we're going to assume that just about any sensible card can get its act together with eight or fewer ports. As such, we're actually only going to decode the first *three* lines. However, as will be obvious once we get into this, you can work in the fourth and fifth ones if you need the whole range of 32 ports.

Let's start with a simple example. Assuming that we want to decode port 300H, we would want all three of our lines to be low. This is dead easy. Simply connect three inputs of a quad input NAND gate across the three lowest address lines. When the card is selected and the output of the gate goes high, the processor is speaking to port 300H. This is the simplest decoding possible.

Since we're ignoring the fourth and fifth address lines, in theory the processor could be speaking to either port 300H or to the "phantom" ports 308H, 310H, 318H or 31FH and have this decoding turn up valid. It would be the responsibility of the software driving our card not to attempt to write to ports we haven't planned on decoding.

In practice, it would be a pig to have to decode eight port addresses with individual gate and inverter combinations, and, fortunately, we don't have to. There's a really splendid chip that will do it for us. The 74LS138 is an eight bit decoder/demultiplexer. What this means in humanspeak is that it will translate a three bit binary number, such as the one found in the three low order address lines into eight individual lines. Since a binary number is a collection of ones and zeros, on and off states, each bit of the output of the 74LS138 represents a flag that tells us which of the eight ports in the range we want to decode is being addressed.

The output of the 74LS138 can only have one line selected unless you connect the power backwards and short it out.

The output of the 74LS138 is also controlled by three enable lines. If we enable the chip with the port select line from the card, it will only allow its outputs to reflect its inputs when the card is being spoken to, that is, when the processor is attempting to read from or write to a port in the range of 300H to 31FH.

Assuming that we wanted something to happen when the processor wrote to port 302H, for example, we would wait for the card to be selected, as determined by the circuitry we looked at in previous months, and then wait for the third output line of the 74LS138, pin thirteen, to go low. Actually, as the chip is enabled by the card select logic, we can cheat and just watch the output of the 74LS138 all by itself.

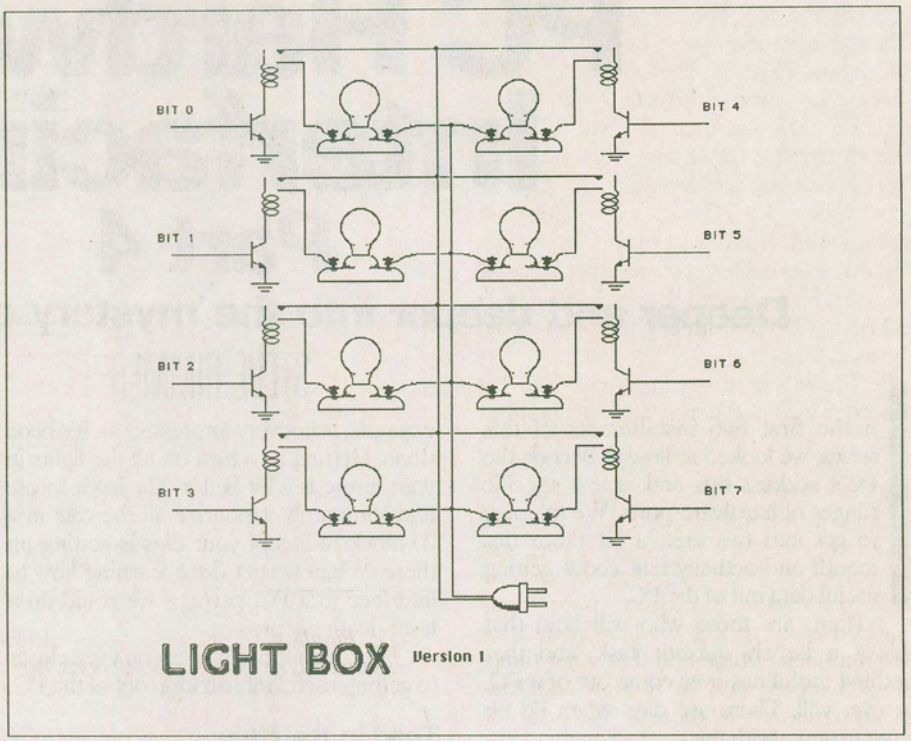

Fig. 3. A simple peripheral, this device will light up eight light bulbs based on the binary number at its input. This is not a complete schematic — don't try to build it.

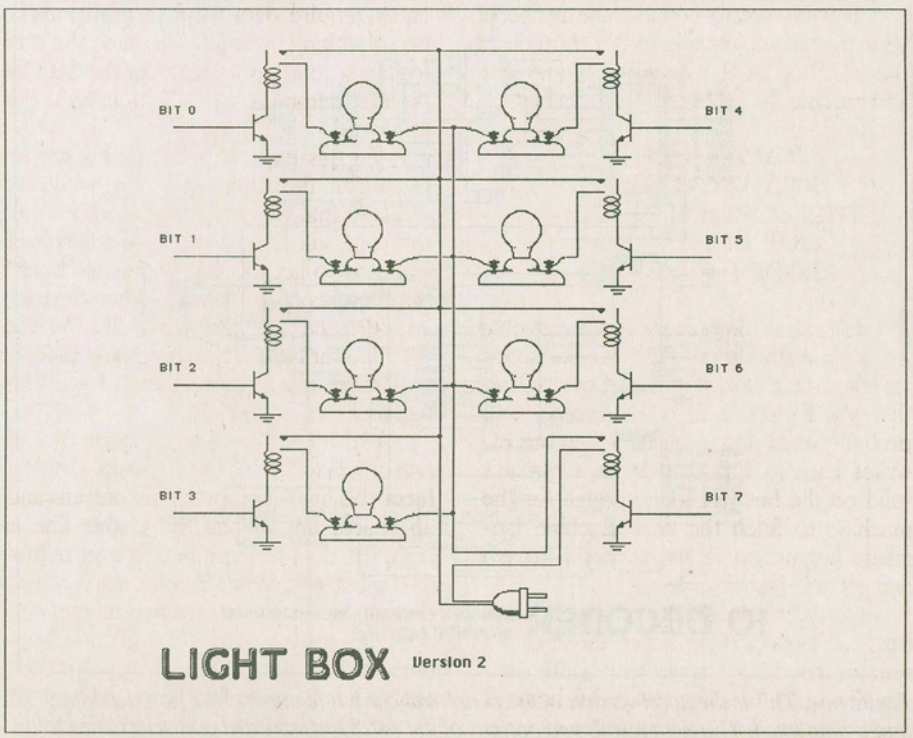

Fig. 4. The modified light bulb box with seven bulbs and a valid data flag. None of the bulbs will come on unless the high order bit of the port is set.

By the way, you can actually cascade these things to form a 32-bit decoder if you need one. This will allow you to decode all five of the lower address lines and let you build a card that uses all 32 of the ports we've decoded for at once. Short of some sort of extreme burglar alarm, the furnace controller of the gods or a laboratory system to monitor the systematic investigation of several thousand mouse mazes all at once, I have a hard time imagining what 32 ports (256 individual data lines) would be used for.

There's a diagram for a five bit decoder included herein for your pondering.

Lifting the Latch

Until now, we've been designing our card in a rather sterile, extremely forgiving, sort of universe. In this universe, we've been able to assume that computer phenomena happen more or less predictably, last for long periods of time and that other things don't happen until we're well and truly ready to ready to let them.

Very little of this is actually the case. About the only part of it that is true is that we exist in a universe, and some of the more existential philosophers about the planet would certainly question even that. The important point herein, however, is that the universe, if it does exist, is a very nasty place for computers.

If you were to connect the probe of your trusty oscilloscope to pin thirteen of the 74LS138 in the circuit in figure one and execute the following program...

```
    MOV DX,0302H
    MOV CX,00FFH
THELOOP MOV AL,CL
    OUT DX,AL
    LOOP THELOOP
```

256 extraordinarily short pulses would grace the cathode ray tube of your scope and the whole works would be over so fast that you'd miss it if you blinked... and probably even if you didn't. When the PC writes data to a port, that data remains valid on the bus just long enough for the machine to fetch the next machine language instruction in its current program and get on with things.

It's the responsibility of a peripheral card to do something about the data it's sent in the short time that said data remains valid. In some cases this involves using the data for its ultimate purpose in that interval, but it's more often the case that the data has to be stored somewhere.

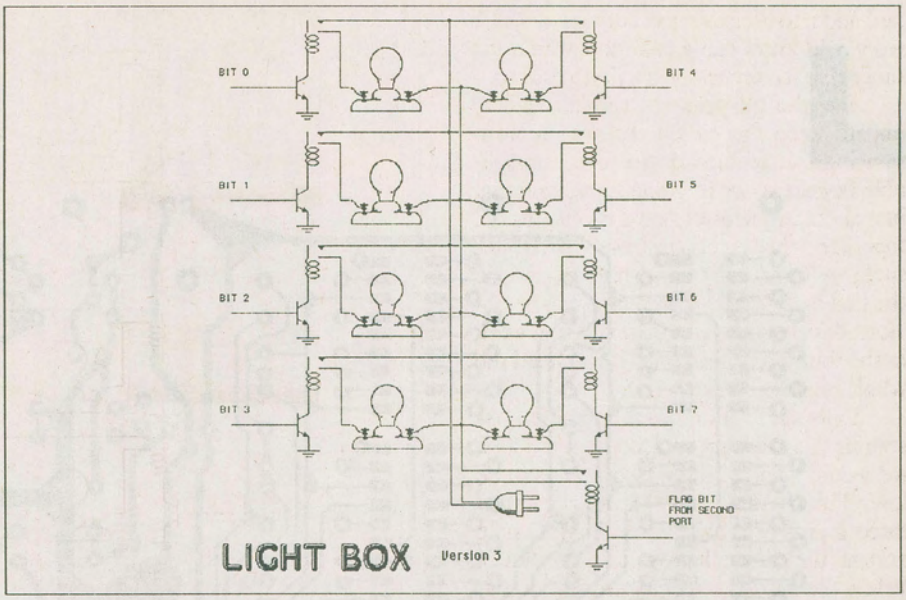

Fig. 5. Finally, civilization is saved — a light bulb box with eight bulbs and a valid data flag. Note that this requires two input ports, however.

In the case of a simple interface card, such as a printer port, it's dangerous to assume that the peripheral device that's being interfaced to is fast enough to catch a fleeting byte of data in the short time that it exists at a PC's port. In fact, it's ridiculous... printers aren't anywhere near that quick.

More to the point, most complex cards require data from several ports to do something useful. By the time the data for the second port shows up, the data for the first one must, by definition, have disappeared.

To this end, we must find a way to memorize the data at a port while it's valid, so that once it's gone from the bus the card still knows what has transpired. The way to get this together is to "latch" the contents of the data bus when the card is addressed and the select line on the 74LS138 for the port in question goes low.

There are, of course, chips to get this together too.

The 74LS77 is a four bit latch; it takes two of 'em to latch one byte. In its simplest form, this has four inputs, four outputs and an enable line. When the enable line is high, the data at the inputs is transmitted to the outputs. When it goes low, the last data at the inputs is retained at the outputs, no matter what subsequently happens at the inputs. As such, by pulling the enable line low, the latch turns into a four bit memory chip for as long as the enable line stays low.

You'll note that the outputs of the 74LS138 go low when their corresponding binary number appears at its inputs. This fits together very nicely. If we connect one of the eight outputs of the 74LS138 to the enable lines of our latches, the data at the output of the latch will always reflect the last data sent to the appropriate port.

Port Strategy

At this point, we're going to digress a little. You've probably had port addresses up to your clavicle. For the rest of this month's feature, then, we're going to look at some strategies for using the data that appears at our newly found port in sensible ways.

Data which just appears at a port is seldom useful, as the device which it's connected to can't know whether it's good data or not.

Consider a simple peripheral device, as seen in Fig. 3. This is a box with eight inputs, eight transistors, eight relays and eight light bulbs. Certain components of this thing have been omitted for clarity. There should also be a power switch and, as with most projects which involve controlling large amounts of current and voltage with a computer, a fire extinguisher nearby. Because this is a hypothetical project, a hypothetical fire extinguisher should suffice.

This box will turn on a light for every line that goes high at the output port of our little board. When our PC first boots up, some lights will probably come on because there is garbage data in the latch. Writing

Continued on page37

numbers to the port will cause different combinations of lights to come on by design.

The important thing here, however, is that the lights don't know what's good data, and they're quite content with the garbage data left over from the PC's powering up. If all you're after is a light show, this probably won't matter. On the other hand, if all you're after is a light show you could probably think of an easier way to get it than by tying up a computer.

What we need is a way to signal our hypothetical peripheral when the data at the port is good. We need a flag.

The simplest flag is simply one bit of our port. For example, consider changing the relay situation around such that there are now only seven lights. The relay for the highest bit of the port is connected between ground and the common side of all the light bulbs. Now, in order to light the bulbs the seventh bit of the port has to be set. The seventh bit of the port has become our flag. When it's high, the light bulb peripheral knows that the data at the port is good.

This has one serious drawback. We've been forced to give up a light bulb to get our flag.

Since we're just oozing with available port addresses, it would be a lot more practical to go back to our eight original light bulbs and use a second port to handle the flag. The light bulb port is called the "data" port, and the flag port the "status" port. We would now need a second set of 74LS77 data latches on our card, one for each port. We could connect the common relay to the lowest line of the status port, such that if it went high the light bulbs would be free to light based on the data at the data port. If it went low, they would all be inhibited no matter what appeared at the data port.

In more complex applications, such as interfacing to slow data peripherals, the role of the status port becomes more involved. Consider a printer, for example. We know that in the time it takes the printer to print a single character, the computer could have sent a few thousand to the port address of our choosing. Obviously, we would want a status flag to tell the printer that there's data a'waitin', but we would also need a flag to tell the computer that the printer is free to accept data, lest it send a new byte to our latch before the printer has had a chance to read the previous one. This requires some still more complex hardware, as our latch must be bi-directional.

We'll look at the details of this in a future installment.

Byte Size pieces

Having reached the point wherein there is actual, useable, information appearing at the output of your PC, you might want to start thinking about things you'd like to interface to. There's still quite a bit to look at before we have a complete, practical two way data path between the reality of the universe and the decided unreality of a computer, but things are looking more and more like we'll get there in the end.

There's a light at the end of the tunnel... with any luck, it won't turn out to be someone with a particularly bright monitor playing video games. ■