PC Hardware Interfacing

Exploring the mysteries of designing PC interface cards.

STEVE RIMMER

The mysteries of computer hardware are like no mysteries you'll ever check out on Masterpiece Theater. These are nasty ones. A good bit of hardware microcomputer design will make you long for boat building.

This being the case, of course, designing microcomputer circuitry is one of the most rewarding things you can turn your soldering iron to. It's tricky and demanding, but when you finally do get the thing going it's a hell of a rush.

Designing interface cards for the IBM PC is a slightly tamer subset of microcomputer hardware design. Notice that I said "slightly". It has the advantage that you don't actually have to design a whole computer, and the disadvantage that you have to work with an architecture which was originally perpetrated by a three foot two inch tall green skinned troll from Mars who had it in for everything bigger than he was... this, of course, being the creature which IBM actually employed to design the original PC. Betcha didn't know that.

It's not horribly difficult to design boards to plug into a PC, although you do need to know a few things first. Successful hardware design for the system involves both knowing where to put the chips and, subsequently, knowing how to talk to them from within the computer.

Over the next few months we're going to see how to do both. You might also want to check out the C language series that's starting this month. The two will kind of fuse together later on, inasmuch as you'll need a programing language to communicate with the cards you'll build.

In this first installment, we're going to look at the evil secrets of I/O decoding. Some people have described I/O decoding on a PC as being more fun than sex — not very many, though.

Any Old Port in a Storm

In order to understand how to talk to a PC at the hardware level, you have to begin by getting your head around how it talks to itself. You can start by putting your ear up to the ventilation grill and listening, but this usually just gets you a drafty ear.

The "bus" of the 8088 microprocessor that drives a PC is actually several buses... sort of a transit company. Looking at things from the point of view of the I/O bus connections — essentially the processor's bus brought out to slots so we can associate peripherals with it — we have a power bus, a data bus, an address bus and a miscellaneous signal bus. An AT has still more buses, and an 80386 based system looks like downtown at rush hour — we won't get into them just now.

When the 8088 says that it feels like writing a byte of data to a specific location in memory, here's what actually happens. The processor puts the number it wants to load into the memory onto its data bus. The data bus has eight lines, so the number can be anywhere from zero to two hundred and fifty five. Next, it puts the number of the location in memory where it wants the data to go on its address bus. The address bus has twenty lines, so this number can be anything from zero to a megabyte. Finally, the processor pulls the

MEM W line of the miscellaneous signals bus, which is a signal to write data to memory.

We think of the memory in a PC as being part of the computer, but the processor sees it as being a peripheral. It's a black box which stores things in a way that's useful to the processor. As such, we can envision the memory system of the PC as a machine which stores numbers when the MEM W line is pulled and spits them back out when the MEM R, the memory read line, is pulled.

In theory, the memory doesn't have to be actual silicon memory. We might imagine a tape transport with a long tape capable of holding a million items of data. When the MEM W line is pulled, the tape transport moves the tape so that the item whose number is contained in the address bus is over the tape head and writes the number on the data bus to it. In theory, the 8088 could drive such a memory device just as it drives normal IC memory. In practice, of course, such a contraption would be too slow to possibly be good for anything, but it illustrates how the 8088 actually regards its memory.

I bring this up because it points out that memory doesn't have to be memory, and, in some machines, frequently is not. Suppose we built a circuit which would watch the address bus for a particular number and light up an LED every time that number appeared on the bus. This is pretty easy to envision; it would just be a series of gates and inverters. If the number it was looking for was one hundred, every time the processor tried to read or write to location one hundred, the LED would light up for an instant.

Now, suppose we expanded this circuit so that the number on the data bus was shown on a display whenever the processor accessed location one hundred and it pulled the MEM W line, that is, whenever it actually wrote data to this location. Ah hah... we've invented an output port.

We could make this into an input port by having the circuitry read an external condition in eight bits and jam it onto the data bus whenever the processor accessed location one hundred and pulled the MEM R line.

This is a very simple sort of I/O called "memory mapping". It's used on Apple][+ computers, for example, because it's the only sort of I/O that computer's microprocessor knows how to do. Memory mapped I/O has the advantage that it's fast and easy to do, but it ties up

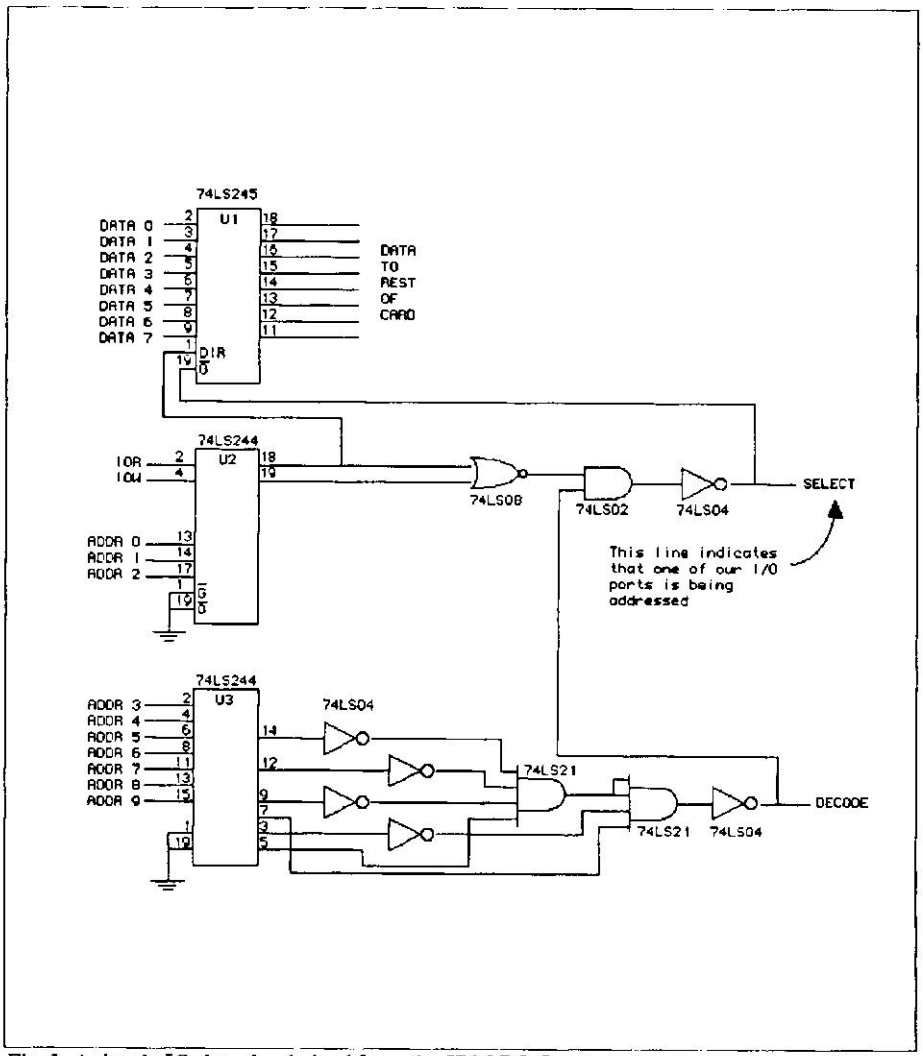

Fig. 1. A simple IO decoder derived from the IBM PC. It selects the port range from 300H to 31FH.

memory space for what is actually a non-memory function. As you might have noticed, PC users invariably want all the memory they can get, and would be loath to give any of it up if they could avoid it.

The PC does use some memory mapped I/O. The video board of a PC is just a big chunk of circuitry which makes a range of memory appear as data on a picture tube. If the processor writes to this area of RAM, the screen contents change. Of course, there is actually memory here, although it lives on the video board and is essentially part of the display circuitry.

Memory mapped I/O is faster than the other kind of I/O, which we'll get to in just a sec, because in effect, the processor gets to talk directly to the screen. In the case of the PC's screen, which we'd like to have operate as quickly as possible, the design of the computer allows that it's worth giving up some memory space to come up with speedy communications.

For most of its basic I/O, however, the PC uses I/O ports. A port is a funny sort of phenomenon. In some ways it behaves like memory, but it lives off the memory bus and, as such, the processor can access a full house of memory and still have lots of ports going.

The PC's use of ports is a bit erratic, as we'll see.

One way of looking at I/O ports is that they're just memory locations that the memory system ignores. When the 8088 wants to write to memory location one hundred, it puts one hundred on the address bus, data on the data bus and pulls the MEM W line, as we've seen. When it wants to write to I/O port one hundred, it puts one hundred on the address bus, the data on the data bus and it pulls the IOW line. Because the MEM W line does change, the data doesn't affect the contents of memory. The hardware which is associated with the port, assuming there is

any actually there, will watch the two buses and this line and do something with the data when its port is written to.

The whole process seems a bit artificial, and, in a sense, it is. It's a protocol by which the 8088 uses the same address and data buses to communicate with both memory and I/O devices.

In software, by the way, the machine language to do these two analogous functions is also similar. This

```
MOV DX,100
MOV [DX],AL
```

would write to memory location one hundred, while this

```
MOV DX,100
OUT DX,AL
```

would write to port one hundred. As memory access is considerably more common in a PC than is port access, there are considerably more ways to handle variations on the first case.

By using the address bus in this way, the PC allows for multiple devices to sit on the I/O bus at once. Each device has... presumably... a unique I/O address, so that the computer knows, for example, that talking to port 03F8H will involve chatting with the COM1 serial port while port 03D8H is one of the registers of the video card controller. The actual slots what the devices sit in are all identical, and peripheral cards, as a rule, don't care which slot they're plugged into. They know they're being spoken to when they see their port numbers appear on the address bus and the IOW or IOR – the I/O read line – pulled.

Kibbles and Bits

In practice, most peripheral cards have more than one port, and we usually talk about associating a range of ports with a given I/O function. If we consider a simple parallel printer interface, for example, at the very least we would need one port to actually send data to the printer and a second one to control the beast... to tell it when the data on the first port is valid and ready to print, to tell the printing software when the printer is ready to accept data, whether it's out of paper and so on.

A basic subject of board design for the PC, then, is the decoding of I/O addresses. Before we can do anything clever, we need a circuit which will look at the buses of the computer, decide when our card is being communicated with and extract the appropriate data from the data bus. As we get into this, we'll find that there are other considerations to contend with... for example, it's desirable in many cases to make the base of the range of addresses which our card uses somewhat variable, so that it can accommodate other cards which might be in the system using fixed ranged of port addresses.

Fig. 1 is a simple I/O decoder, derived from the now antediluvian original IBM PC prototype card; it selects the port range from 300H to 31FH, which is a bit huge. This means that the line I've marked SELECT will go high if one of these addresses is placed on the address bus and one of the I/O lines is pulled, and it will stay low for the rest of the time.

Let's see what the beast is up to. We'll start by looking at how to decode for the address 300H

We can represent the hexadecimal number 300H in binary as

1100000000

If you imagine an LED connected to each line of the address bus, when the number 300H was on the bus, the pattern of on and off LEDs for the first ten lines would correspond to the ones and zeros of our binary representation of the data.

If we designed a circuit to look for the following pattern in the upper two bits

11

we'd have a way of knowing whether the address ranged from 0300H to 03FFH... this involves simply watching for the number three in the upper two lines of the pertinent part of the bus. You can envision such a circuit pretty easily: it's just a two input AND gate. Watch these two lines and the IOW and IOR lines as well and you've got an I/O port decoder for this admittedly too large range of addresses.

If we watch a slightly larger number of lines.

110

and only raise our SELECT line when that extra bit is zero, we'll have narrowed the range down to 0300H through 037FH. If we watch this range

1100

we can narrow it down to 0300H through 033FH. Finally, watching

11000

gets it down to 0300H to 031FH, which is what we're after. If this pattern of bits exists on the address bus in these positions... and either the IOW or IOR line has been pulled... the processor wants to talk to a port in the range of 0300H through 031FH, the ports our card is interested in.

The lower five lines are immaterial to the task of deciding whether our port address range is being talked to, as these represent the numbers from zero through 1FH, and our range of ports encompasses all of them.

We'll need to deal with the lower five bits later on, when we want to know which particular port in our range of ports is being dealt with, but that's another story.

The arrangement of gates at the bottom of the circuit diagram in figure one is a fixed address decoder. If you care to trace through it, you'll find that it watches address lines five through nine for the pattern 11000, raising the DECODE line if it spots them.

The NOR gate up in the centre of the drawing watches the IOW and IOR lines. It tells us when either of the lines is pulled, as this indicates that some sort of port I/O is happening. The status of the IOR line is used to set the direction of U1, which buffers the data. We only want data to go out... that is, to be jammed onto the 8088's data bus... when the IOR line is high.

The 74LS02 AND gate after the NOR gate gets the whole party together. It tells us when the address has been decoded *and* an I/O line is high. As such, its output can be used to say when we actually should be ready to do some work. Notice that it also enables data transmission through U1; most of the time this is in its tri-state mode, and does nothing.

The Load Out

At this point, we know how to make the card decide whether it's being addressed in a general way. The next step is to see how to make it recognize individual port addresses when it's spoken to, and then what to do with these addresses. We'll have a peek at some of this next month.

In the mean time, you might want to dream up some actual applications for cards you build yourself. Once you know how to make your circuitry actually communicate with the PC – pretty simple stuff in the end – the rest is just solder and blinding inspiration. We'll check out some of that in the months to come too. ■