FEATURE

PC Hardware Interfacing

Part Three, in which the art of discoursing with hardware is briefly touched upon.

STEVE RIMMER

We're a bit short of space this month. All right, then, we're exceedingly short of space this month. Circumstances have dictated that I have a page or so for the hardware thing. Since any useful circuitry would take quite a bit more than this, we're going to look at a non-hardware part of hardware design for the PC. This time around, we're going to touch briefly on the software which talks to the boards we're going to be designing in the months to come.

It was going to get down to this sooner or later. The cleverest cards in the world are useless without software to drive them.

Any Old Port

As we've noted in the past two episodes of this saga, it's usually the case that one communicates with a peripheral card in the PC using ports. This isn't always the case. If we want to move a lot of data around, we might allow the card to read or write directly from or to the system's memory. This is called, not surprisingly, "direct memory access", or DMA. It's a complex subject on the PC, because the processor doesn't particularly like it. We won't be dealing with it here.

For the most part, though, we'll be using ports.

You can read the data from any of the PC's ports in machine language with a few simple instructions. For example,

```
MOV AL,20H
```

will read the information from port 20H. This port happens to be part of the interrupt controller on a stock PC, but this is irrelevant to our discussion here. Having executed the above instruction, the AL register contains authentic read-in data. Whether or not it's of any use remains to be seen.

You can read a port at any time, but, depending upon the design of the card which is driving the port, the data may or may not be good when you read it. As such, it's usually the case that the card must be able to indicate when it has good data. The simplest way for it to do this to use a flag. This usually means that it needs a minimum of two ports.

Let's say that our card uses its base port address as its data port and the first bit of the next port along as its status flag. When this bit is high, the data at the data port is valid. Here's some code to wait for good data and read it. We'll say that the base port is 300H.

```
PORTEQU300H
PORTLP:INAL,PORT+1
TESTAL,01
JZPORTLP
INAL,PORT
```

This would not be a terribly clever bit of code to actually use. If no data ever appears at PORT, your computer would lock up. However, it illustrates the principal of how to wait for a byte of valid data. In practice, we would probably have the program wait for a specific time and then exit the loop, or, more sensibly still, we'd have it test PORT+1 every so often, doing other things in the mean time.

The technique of checking for valid data like this is called "polling". It's not the most desirable way to read incoming data most of the time. It assumes that the computer will always be free to poll the status of whatever is sending it the data at least as frequently as the data appears. If the machine goes off and is unavailable to poll the status port for long enough, a second byte of data might appear at the port, destroying the first one before it could be read.

Besides which, polling for data like this wastes the machine's time. By rights, it should be able to ignore our peripheral card until some data actually shows up.

There is, of course, a better way. ■

Interruptus

The proper way to handle this situation is to make the card send interrupts when it has data to give the PC, and to make the code which reads the card "interrupt driven". The only drawback to this is that there are very few available hardware interrupts available in a PC... most of them, as we touched on last month, are already spoken for.

We're going to use interrupt two here. It's not usually dedicated to anything. In complex applications, it's possible to avail the PC of extra hardware interrupts by daisy chaining it's internal 8259 interrupt controller chip to a second controller. We're not going to pry open this particular can of worms today, however.

In this example, let's say that our peripheral card is so designed that, rather than raising the first bit of PORT + 1 to indicate the availability of data, it raises interrupt line two of the PC's bus. When this happens, the PC will immediately stop what it's up to, push its current code segment, instruction pointer and flag register onto the stack and leap to wherever the vector for this interrupt tells it to go.

Hopefully, the program which we've written to drive our card will have had the sense to point this vector to a suitable bit of code. The code which deals with a hardware interrupt is called a "handler".

This is a simple interrupt handler.

```
HANDLERPUSHAX
INAL,PORT

;DO SOMETHING WITH THE DATA

MOVAL,20H
OUT20H,AL
POPAX
IRET
```

An interrupt handler must always save all the registers it might corrupt onto the stack before it does anything and then restore them when it's done. In this case, only the AX register gets bopped. We can inhale the data from PORT with impunity, as we know that our card would only have thrown the interrupt if there was good data there. The handler would normally put the data somewhere so that it could be accessed by another program in a more conventional manner.

The next two lines of code signal that the interrupt is complete. Finally, we return from the interrupt with an IRET instruction, which will put us right back where we started before the interrupt took place. So long as we have preserved everything properly, the interrupted program will never know that anything happened.

And Then We Were Three

There's a lot more to it than this, of course. Next month we're going to get back into the seething world of hardware design, but we'll be doing so with some of the things we've looked at here in mind. It's important to understand the relationship between hardware and the software which will drive it if one's peripheral cards are not to become the doorstops and hamsters' grave stones of generations yet unborn. ■