FEATURE

Interfacing the PC
Part 6

This month we'll start looking at real world solutions to PC interfacing problems. Let us transcend the world of discrete chips and move on up to large scale integration.

STEVE RIMMER

Until now we've been looking at interfacing solutions based on large numbers of fairly simple devices. It is, of course, possible to build virtually anything with nothing more than ten or twenty pounds of LS logic and a printed circuit board the size of Alberta. Such a device would not be elegant but, to be sure, it would be a lot easier to understand than a whole computer on three chips.

The circuitry we've looked at to date has had the advantage of being easy to understand. It hasn't been terribly practical, not because it wouldn't have worked, but because it would have been excessively tedious to have implemented. It had a lot of parts for relatively little functionality.

All the chips involved in our interface card to date went to support but a single port. As we saw early on, most interface applications require at least two. In addition, our port lacked any hint of software programmability. It was exceedingly inflexible. To have added this sort of feature, however, would have sucked back quite a few more chips.

Unless you're right out there with your toes curled over the leading edge and your face staring into the void, you'll probably find that your interface project has, to some extent, been done before. In fact, most applications for PC peripheral cards have been done so many times before that there are now complex chips available to handle the works. These sorts of chips typically do everything that a herd of small scale devices will handle, but they do it faster, with less power consumption, more reliably and cheaper.

The only drawback, of course, is that one can have no idea as to *how* they do it. They are, quite literally, black boxes. However, in the real world no one really cares how it works so long as you can't get sued for it.

In designing practical PC interface cards, getting the "package count"... the number of chips... down is really quite important. It's usually a lot more desirable to use one big expensive chip than lots of little cheap ones, as the labour costs involved in placing all the little chips... or the time involved on a pick and place machine if the card is to be assembled automatically... along with the greater potential for error or bad parts usually far outweighs any potential saving in the cost of the chips themselves.

Ports of Call

As we've been looking at building a parallel interface card for the PC with discrete parts, it seems reasonable to start looking at a large scale device which will do the same thing. In this installment we're going to get introduced to the Intel 8255, which is a rather huge chip that manages three bidirectional I/O ports, each of which is programmable eight ways to Sunday. Designed by the same company that does the processor which drives a PC, it uses somewhat similar interfacing techniques and most of the same names for things as does the 8088.

Interfacing an 8255 to a PC is quite a lot simpler than having the PC talk to our handful of chips last month. However, for the moment, let's concentrate on the beast itself.

The 8255 is a general parallel port chip designed to be used with a port driven microprocessor. As we've seen in the past few months, when the processor wants to talk to a port, it puts the port address on the lower part of the address bus and the data on the data bus, and it then pulls the IOW line. In the case of the 8255, the port chip maps onto several consecutive ports, as we will require a range of ports to be able to properly control it.

There are twenty four input/output lines squirting out of the 8255. These can be regarded as being three eight bit ports and, for reasons which will be obvious shortly, this is really how the chip deals with them. However, they can also be programmed into other combinations. For example, we can treat them as two twelve bit ports. We'll get into this momentarily.

We're going to pretend that we've already gotten past all the technical details of interfacing the 8255 to our PC for just a moment so we can look at the characteristics of the chip itself. While this may seem like an awkward order in which to handle the task, it's usually how things are done with large scale devices like this one. It's important to understand how the 8255 behaves from the point of view of the software which will drive it so that we can understand how best to handle its hardware.

In this case, we're going to have our hypothetical 8255 decoded so that its range of ports starts at 0300H... when we get to the actual interfacing circuitry you'll notice that some of our previous card will have made the transition into this new incarnation.

Interfaced like this, the 8255 will appear as four consecutive ports to the processor. In machine language terms, we would define these as

```
PORT_AEQU300H
PORT_BEQU301H
PORT_CEQU302H
CONTROLEQU303H
```

This is very much less mysterious than it seems at first.

In its simplest mode, every input/output line of the 8255 can be set up in one of two states, to with, as input or output. Now, the chip doesn't actually allow us to individually set the state of every line, but, rather, implements a number of reasonable permutations. The actual configuration of the twenty four lines is determined by the setting of a mode byte. This byte, not surprisingly, is sent to the chip through the control port, that is port 303H in this case.

In order to use the 8255, we must set

the mode byte to tell the chip which of its ports are going to be input ports and which will be output ports. We can then deal with the ports directly by reading from and writing to the three data ports from 300H to 302H. For example, if the first port of the chip were to be defined as an output port, we would subsequently write to it by saying

```
MOVDX,300
MOVAL,45
OUTDX,AL
```

The number 45 is the data being written to the port.

Initialization

Initializing the 8255 is fairly complex. However, it's the only complex thing about the chip. Once you've initialized it, communicating with it couldn't be simpler.

The 8255 can be set up to work in three modes. These are

```
Mode 0Basic single directional ports
Mode 1Interrupt driven mode
Mode 2Bi-directional mode
```

We're going to look at mode zero right at the moment. The other two modes are powerful, and will prove useful later on when we're developing interrupt driven strategies for the card. However, for the moment, let's just make the beast talk.

The mode we choose is communicated to the 8255 through a mode byte, which, just for the sake of perversity the chip's documentation likes to think of as a "mode word". As with most of the well bred bytes, this one is made up of eight bits, and every bit has a meaning all of its own. Here they are

```
Bit 0Port C lower : 1 = input, 0 = output
Bit 1Port B : 1 = input, 0 = output
Bit 2Mode select : 0 = mode 0, 1 = mode 1
Bit 3Port C upper : 1 = input, 0 = output
   Bit 4Port A : 1 = input, 0 = output
Bit 5Mode select
Bit 6Mode select : 00 = mode 0, 01 = mode
1, 10 = mode 2
Bit 7Mode set flag : 1 = set mode
```

This actually defines the functions of the chip very elegantly if you stare at it for a while. The 8255 behaves like two separate devices, or, at least, it can if you ask it to. Obviously, if you program both devices with the same mode information it behaves like one device. The first device consists of port A and the upper four bits of port C. The second device, logically enough, consists of port B and the rest of port C.

This distinction allows us to program each device with a separate mode if we want to. This is useful because the interrupt mechanism of the chip uses port C to actually do the interrupting.

In our examples here we'll be programming the 8255 so that both of its sections will be running in mode zero. As such, it will behave like a single device, and we can ignore its schizophrenia.

In order to set the mode of the chip, then, we must assemble the control byte out of the appropriate bits to program the 8255 for the functions we want. Let's walk through a simple example. In this case we want port A to be an input port and ports B and C to be output ports.

First off, bit seven must be one, as this is a flag to tell the 8255 that the data being sent to its control port is to be regarded as being a mode change. Thus, we start with 80H.

Next, let's set the mode for port A and the upper part of port C. We want this to be mode zero, the simple I/O mode. This means that bits five and six must both be zero. Our mode control byte is still 80H.

Port A is to be an input port. To do this, bit four must be one. We thus OR 80H and 10H... 10H being what you get if the fourth bit of a byte is set. The result is 90H.

Port C is to be an output port. We can set the upper hap of it so by making bit three zero. We're still at 90H.

The mode for port B and the lower half of port C is zero, so bit two will be zero. Port B is an output port, so bit one will be zero. The lower half of port C is an output port too, so bit zero will be zero.

Our mode control byte is 90H.

Having worked this out, we can set our chip up with the following bit of code

```
MOVDX,CONTROL
MOVAL,90H
OUTDX,AL
```

If we now do this

```
MOVDX,PORT_B
MOVAL,55H
OUTDX,AL
```

every other line of port B will go high. The number 55H is a particularly useful one for these kinds of tests, as it has all its odd numbered bits high and all its even numbered bits low. It's unlikely to occur as a garbage byte, and so it's a useful check to see if something like this is working.

Calculating the mode bytes for 8255 initialization is a genuine pain. To this end, the following bit of assembly language is really helpful. Aside from being easier to use, it keeps you from making mistakes. This is worth the effort, as having the 8255 incorrectly initialized can make it do the weirdest things.

```
PORTC_MODEEQU00001001B
PORTB_MODEEQU00000010B
PORTA_MODEEQU00010000B
8255_MODEEQU01100100B

   MOVAL,80H
   ORAL,8255_MODE
   ORAL,PORTA_MODE
   ORAL,PORTB_MODE
   ORAL,PORTC_MODE
   MOVDX,CONTROL
   OUTDX,AL
```

I've set the significant bits of each of the four equates to one so you can see where they are. In this case, we've set all the ports to output and the chip into mode two for both devices. The mode control byte would be 0FFH. In practice, you'll probably want to change this.

Actually, in practice you'll probably let the compiler OR the four bytes together, rather than doing it in assembly language, that is

```
MODE_BYTEEEQU8255_MODE OR
PORTA_MODE OR PORTB_MODE
OR PORTC_MODE
MOVDX,CONTROL
MOVAL,MODE_BYTE
OUTDX,AL
```

The first line has wrapped here... it all has to go on one line in an assembler program.

Post Initialization

The 8255 is quite an old chip, predating the 8088 that the earliest PC's were based on. Its documentation speaks of its being used to control things like the hammer relays of a teletype. However, it's a good chip to use for this sort of application. It's cheap, fast and pretty easy to get up and running.

As we'll say later on, it's also incredibly flexible... possibly more so than one might ever require. ■