FEATURE

PC Hardware Interfacing Part 11

This month we'll start looking at the machine language driver for the PC serial port card. This will include the mysterious and justifiably feared interrupt handler mechanism.

STEVE RIMMER

With the design of a PC serial port behind us, we can start looking at the software to drive the serial port card. Obviously, the exact nature of the software, and where it will reside, will be determined by you application. However, this discussion will show you how to write drivers for hardware which require that the software really tickle things at a very low level.

The serial card, like most of the more interesting PC hardware peripherals, is an asynchronous device. There's no telling when data will appear at it, nor how much of it is likely to turn up at a time. As such, we have to find sneaky ways to deal with its requirements of the computer.

Interrupts for Free

As we have discussed, the serial port card can be dealt with in a number of simple ways, most of them involving "polling" the status port of the card to see if there's data waiting in the input buffer. The phrase "input buffer" may be a bit misleading when it comes to the 8250, the buffer which the chip provides is only one byte long, which isn't much of a data stash.

The problem with polling the port is that the program doing the polling has to check the port at least as frequently as the data is likely to come in, and it has to be able to handle any data it finds in the interval between two incoming characters, or data will be lost and the information being transmitted over the serial port will be seriously mangled. On a good day this simply means typing up your whole computer is a largely mindless task. On a bad day it doesn't work at all—if the baud rate is high enough or the data handling process complex enough, polling universally falls apart.

This is why the 8250 provides for an interrupt driven strategy as well. The chip can be programmed to throw an interrupt every time a byte comes in, such that the computer can stop what it's doing, save its registers, get the byte, handle it, restore its registers and return to what it was up to, all without the operator of the machine knowing that anything has happened.

The usual function of an interrupt driven serial program is to store the incoming bytes in what is called a "circular queue". This may take a bit of explaining. While serial data frequently comes in at a rate which exceeds the ability of the host software to process it, it rarely does so for any length of time. For example, XMODEM style file transfers work by sending a block of data, say one hundred and twenty-eight bytes, and then waiting

for an acknowledgement from the other end of the conversation before sending another block. Even if the block itself comes down the wire very quickly, the receiving end will have a chance to process it — in this case, possibly to write the block out to a disk file, in the interval between acknowledgements.

The only difficult part is being able to stash the bytes somewhere until the pause. This is how the interrupt handler works. It maintains three things which it works with. These are the *data buffer*, the *head pointer* and the *tail pointer*. If you understand these you'll have a much better idea of how to successfully process asynchronous data from any external hardware device.

For what it's worth, the data from a PC's keyboard is handled in exactly this way.

The data buffer can be any length. Let's say that it's five hundred bytes long for the moment. This would not actually be a very good choice for most applications — the buffer size should be a multiple of the block size in which your data is likely to appear, but it will do for this example.

The head pointer and the tail pointer start by pointing to the start of the buffer. The buffer is said to be empty initially.

When a character comes in, it's placed where the head pointer points. The head pointer is then passed through an increment routine which increments it and then checks to see if it has exceeded the end of the buffer, which it will not have done, as there are still four hundred and ninety-nine bytes free. If it had exceeded the end of the buffer the increment routine would reset it to point to the beginning. After five hundred bytes, the next byte which comes along will automatically overwrite the first byte in the buffer.

The buffer thus behaves like a circular carousel as far as the data is concerned. As it's incremented, the head pointer always runs around the buffer.

At the same time as all this is happening, hopefully, there will be a foreground task which takes bytes out of the buffer. It does this in much the same way. It checks to see that the head pointer and the tail pointer do not point to the same location, a condition which would indicate that the buffer was empty. Assuming that this is not the case, the code which retrieves data fetches the byte currently pointed to by the tail pointer and then runs the tail pointer through the aforementioned increment routine.

As you can see, then, the tail pointer will chase the head pointer around the circular queue, the gap between them growing and shrinking based on the difference between the speed at which data is appearing at the serial port and that of the software which is handling it in the foreground. However, as long as the buffer is big enough to allow for this slack, no data will ever get overwritten. By the time the head pointer returns to the start of the buffer, the tail pointer should have retrieved the bytes that were there.

Having the head pointer overrun the tail pointer is really the only potential failing of this system. This will happen if the program which fetches the bytes consistently does so at a slower rate than the data appears at the serial port, and if this process continues long enough to fill the buffer. Unfortunately, there isn't really any way for the interrupt handler to cope with a buffer overrun condition which does not involve losing data. It has a choice of either refusing to handle any more input until the tail pointer moves, in which case incoming data will get lost, or it can overwrite the tail pointer, in which case previously stored data will get crunched.

In writing software which deals with interrupt driven serial data, it's important to make sure that the buffer is checked and cleared frequently, and that you allow for an interrupt driver data buffer which is big enough to handle the worst case of a foreground program going for lunch.

A Byte of Code

Here's a simple interrupt handler in machine language. This is just the handler mechanism itself — there's a lot of ancillary code which accompanies it.

```
SERIO_SIZEEQU500

;THIS GOES IN THE CODE SEGMENT
SERIO_HANDLERPROCFAR
STI;ENABLE OTHER INTERRUPTS
PUSHAX
PUSHBX
PUSHCX
PUSHDX
PUSHSI
PUSHDI
PUSHDS
PUSHES;SAVE AFFLICTED REGISTERS

MOVAX,_DATA
MOVDS,AX;GET THE LOCAL DATA
SEGMENT

MOVDX,SERIO_BASEPORT
INAL,DX;GET THE BASE SERIAL PORT
;AND GET THE BYTE WAITING
MOVBX,SERIO_HEAD
MOVSI,BX;HANDLE HEAD POINTER
CALLBUMP_POINTER
MOVSERIO_HEAD,BX
MOV[SI],AL;SAVE BYTE

CLI
MOVAL,20H;SIGNAL END OF INTER-
RUPT
OUT20H,AL
STI

POPES
POPDS
POPDI
POPSI
POPDX
POPCX
POPBX
POPAX;RESTORE REGISTERS
    IRET;RETURN FROM INTER-
RUPT
SERIO_HANDLERENDP

BUMP_POINTERPROCNEAR;INCRE-
MENT A POINTER
PUSHAX
MOVAX,OFFSET SERIO_BUFF-
ER+SERIO_SIZE
INCBX

CMPBX,AX
JGEBUMP_PTR1
POPAX
RET

BUMP_PTR1:MOVBX,OFFSET
SERIO_BUFFER
POPAX
RET
BUMP_POINTERENDP

;THIS GOES IN THE DATA SEGMENT
DGROUPGROUP_DATA,_BSS
_DATASEGMENT WORD PUBLIC
'DATA'

SERIO_BASEPORTDW03F8H
SERIO_TAILDWOFFSET SERIO_BUFF-
ER
SERIO_HEADDWOFFSET
SERIO_BUFFER
SERIO_BUFFERDBSERIO_SIZE DUP(?)
```

Through an as yet mysterious processes, this bit of code will be called every time the 8250 senses a byte in its input buffer and thereby throws an interrupt. Now, this is a very simple serial interrupt handler — for one thing, it's under the belief that the *only* reason that an 8250 interrupt might come down the pipe would be because a character is waiting. In fact, as we'll see later on, there are lots of other reasons and it would be prudent to have the handler check the 8250 interrupt iden-

PC Hardware Interfacing, Part 11

tification register to make sure that the data at the input buffer really is valid.

We'll deal with that another time, however.

When this function is called, the interrupt enable flag of the 8088 will have been switched off by the 8088. This means, for example, that if a keyboard character showed up while this routine was deliberating it would be lost. As such, the first thing to do is to turn this flag back on with the STI instruction. It's desirable to disable interrupts for as brief a period as possible on a machine like the PC which relies upon them so heavily.

The next thing to do is to preserve any registers which might get mangled in the course of executing the code of the handler. We do this by pushing them onto the stack. It's a good idea to just stick all the common registers up there, as in the course of developing an elaborate interrupt handler you may use registers you had not initially planned upon, which can lead to some spectacular system crashes that will be very hard to debug later on.

The actual interrupt itself saves three words on the stack before the processor gets to our handler. These are the current instruction pointer and code segment, which the interrupt will require in order to return to place in the foreground code which it got interrupted from, and the flags register. As such, we can happily trash the flags register without preserving it explicitly, as the 8088 does this one for us.

The next task is to fetch the data segment which the head and tail pointers and the data buffer live in. If this routine was part of a driver which was a resident code module, for example, everything would be in the same segment. In this case you would replace this

```
MOVAX, DATA
MOVDS,AX
```

with this

```
PUSHCS
POPDS
```

The second version just makes the current data segment equal to the current code segment. The code segment is, by definition, the segment in which our handler lives when it's being executed.

The next bit of code fetches the byte which has been received — the input buffer of the 8250 lives at the base port address. We then fetch the head pointer value and save the byte to it. The BUMP_POINTER routine handles the circular increment process, as we've discussed.

In this case we don't really use pointers but, rather, offsets into the serial data buffer. There's a good reason for this — on an 8088 pointers are thirty-two bits long, or four bytes, while these offset values are only sixteen bits. Since our buffer is somewhat less than sixty-four kilobytes in length, about sixty-three and half kilobytes less — we can get away with sixteen bit numbers, saving some space and some code.

The next bit of code is very mysterious.

```
CLI
MOVAL,20H
OUT20H,AL
STI
```

This has to do with the 8259 interrupt controller chip in the PC, which we haven't really had to deal with as yet. The 8259 allows the PC to cope with multiple interrupt sources, serving as a traffic cop when multiple interrupts happen at once. It makes sure that each hardware interrupt winds up having the processor call the correct vector, and it handles interrupt priorities.

The 8259 must know when our interrupt is done so it can be allowed to fire off another one if needs be. The aforementioned mysterious code is an "end of interrupt" signal. Because it would confuse the hell out of the 8259 if an interrupt happened in between these two instructions, we turn off the interrupts for the brief time during which the end of interrupt is being passed to the interrupt controller.

We'll speak of the interrupt controller in greater detail in future installments of this series.

The rest of the handler simply restores the registers and then returns from the interrupt.

Vectors in Space

While this should explain the mechanism of a very simple handler, there's still a lot of additional code required to make this thing into a workable serial port hardware driver. As it stands, this elegant little handler is sitting in space doing nothing because the PC doesn't know of its existence.

Next month we'll look at how it gets hooked into the guts of a PC so that it really starts to grab those bytes and bounce them into the buffer of your choice.