FEATURE

PC Hardware Interfacing Part12

This month we'll look at some additional interfacing code for the serial port card, including the actual functions which allow higher level languages to deal with interrupt driven serial data.

STEVE RIMMER

Last month we saw how the serial port hardware could be driven by a background interrupt handler function. While it's possible to deal with it as a foreground task by "polling" it, much as is done with simpler computers like the Apple][+, this is both inefficient of the computer's resources and probably not all that effective a way to deal with the problem even if your program has time to burn. There's too high a chance of your machine's processor not being around when a byte comes in and, as such, losing data.

The driver fragment we looked at last month illustrated a way to have the card itself call a routine which would store an incoming byte in a circular buffer, oblivious to whatever the computer happened to be doing at the time, and then return to the foreground task. This is really the correct way to deal with asynchronous serial communication.

There were several important elements left out of the code from last month. One fairly obvious element which was missing was a way to get the bytes out of the circular buffer for use by whatever program is supposed to deal with the incoming data.

This month we'll have a look at the foreground part of the driver, the routines which are called to manage the queue.

Heads or Tails

As you will recall from last month, a circular buffer, or "queue", has two pointers into it. The first pointer, the "head", is used to indicate where the next byte will go when an interrupt is thrown by the serial card. This was pretty well covered in the last installment of this series. The other pointer, the "tail", indicates where the next character to be retrieved from the buffer lies.

After each pointer operation the pointers in question are incremented circularly, that is, they're run through a routine which sets them to the next position in the queue. This will be the next location in memory unless the next location lies beyond the agreed upon end of the buffer, in which case it will be the start of the buffer.

There are several states which a circular queue can be in. They are as follows:

Empty: This is indicated by the head and tail both pointing to the same location in memory.

Data pending: This is indicated by the head pointing to a place further along in the buffer than the tail. Note that this may actually place the head before the tail in memory, as it may have "wrapped" around the ed of the buffer. This is still a legal condition.

Over run: This is indicated by the head having wrapped clear around the buffer and passed the tail. If the buffer was one hundred bytes long, this would happen if more than a hundred bytes were written into the buffer before bytes began to be removed from it.

We have spoken this far about the head and tale as being pointers, which in theory they are. For example, you could use the ES:DI registers to point to the place in memory for the next byte to be stored, and then store it there with a single STOSB instruction, which is very elegant.

Whether of not you should, in fact, implement a queue this way will be dependant upon how big a buffer you think you need. A true pointer on a PC is a thirty-two bit number, and a very awkward quantity to work with at the assembly language level. You will find that you'll have to do a lot of juggling between the DI and ES registers, for example, to see if the pointer has exceeded the end of the buffer.

If you wanted to write a terminal program which used all the otherwise unspoken-for DOS memory for as a huge serial buffer... making the possibility of an over run error condition exceedingly remote... this would be a good way to handle it. On the other hand, if you were writing a modem program in which the maximum chunk of data sent at a time could never exceed a few kilobytes, it amounts to a considerable degree of overkill. Sixteen bit numbers will suffice in this situation with room to spare.

If the buffer is a fixed area in memory, that is a buffer created with the DB directive in your assembly language program, you can address it very easily like this. We'll allow that the position in the buffer being addressed is held in the BX register.

```
MOV [BUFFER+BX],AL
```

If it's an allocated buffer, you can load the buffer segment into ES and set DI to point to the buffer itself. The buffer can be addressed like this.

```
MOV [DI+BX],AL
```

Both of these are really pointers in disguise, but the only number you have to work with is a simple sixteen bit integer.

Fetching Bytes

There are two primary functions which must be performed on the byte queue of a serial card by a foreground task. The first is to test the queue to see if data has been placed there by the interrupt handler since the last time all the previous data was removed. The second is to be able to actually remove data, that is, to fetch bytes from the buffer.

The first function can be handled very simply with the following code. This assumes that we'll be using sixteen bit offsets into the buffer rather than true thirty-two bit pointers.

```
TEST_SERIAL PROC NEAR
MOV AX,SERIO_HEAD
```

```
SUBAX,SERIO_TAIL
RET
TEST_SERIALENDP
```

If you call this function and AX is zero, there are no bytes waiting in the buffer. It may seem that this function will return the number of bytes waiting in AX, but this is not actually true. Recall that the buffer is circular. The head pointer can point to memory which lies before the tail pointer in absolute memory locations and still in front of it as far as the buffer is concerned. If this happens... a common occurrence, actually... the number returned in AX will be enormous and quite meaningless.

If this function returns a non-zero value you can retrieve the next byte in the buffer with the following function.

```
SERIO_SIZEEQU500

FETCH_SERIALPROCNEAR
MOVBX,SERIAL_TAIL
MOVAL,[SERIO_BUFFER+BX]
CALLBUMP_POINTER
RET
FETCH_SERIALENDP

BUMP_POINTERPROCNEAR;INCRE-
MENTAPOINTER
PUSHAX
MOVAX,OFFSETSERIO_BUFF-
ER+SERIO_SIZE
INCBX

CMPBX,AX
JGEBUMP_PTR1
POPAX
RET

BUMP_PTR1:MOVBX,OFFSET
SERIO_BUFFER
POPAX
RET
BUMP_POINTERENDP
```

You will notice that this uses the same pointer adjustment routine as turned up in the interrupt handler last month. In a real world driver, it's convenient to actually use the same code, as the head related functions and the tail related functions generally live in the same program. This ensures that both the head and tail pointers will always be treated the same way.

There are a number of other things you might want to be able to do with the buffer. For example, if you wanted to throw away all the data in the buffer you could use this routine.

```
FLUSH_BUFFERPROCNEAR
MOVSERIO_HEAD,0
MOVSERIO_TAIL,0
RET
FLUSH_BUFFERENDP
```

This simply sets both pointers to the start of the buffer again.

Real World Data

In a complex program which wants to interface to a generic serial driver, the author of the driver must often be unaware of what the author of the foreground application will want to do with the serial port. For example, a terminal program which might send over complex screens with lots of ANSI codes and data could well send down eight or ten kilobytes of data without pausing. The foreground task... the terminal program... would expect the serial driver to deal with this without hiccuping and losing some of it. On the other hand, a simple XMODEM program would never encounter more than one hundred and thirty-two bytes of data before the foreground task fetched it from the buffer, chewed on it and then sent the command back up the line for another chunk.

There are a number of useful ways in which the basic driver concept we've been looking at can be enhanced. The first one involves allowing the foreground task to specify the size and location of the buffer. Under the C language, for example, a programmer can *allocate* blocks of memory which are then accessible through pointers. Since the programmer writing the application which will talk to the serial port should know what sort of data is likely to arrive, it makes more sense to have the calling task allocate the buffer and pass a pointer to it to the serial port driver.

This is how the PC BIOS serial handler probably should have worked.

If this is the case, the as yet undiscussed code which initializes the serial card, sets up the interrupts and so on would be passed a pointer to a serial buffer, that is, a segment value and an offset value. Allowing that these will be stored in memory as BUFFER_SEGMENT and BUFFER_OFFSET, you would load bytes into the buffer like this. We'll allow that the value _DATA is the data segment where all the variables get stored for our driver.

```
PUSHAX
MOVAX,_DATA
MOVDS,AX
POPAX

MOVES,BUFFER_SEGMENT
MOVBX,BUFFER_OFFSET
MOVES:[BX],AL
CALLBUMP_POINTER
```

Likewise, you would get a byte from the buffer like this.

```
MOVAX,_DATA
MOVDS,AX

MOVES,BUFFER_SEGMENT
MOVBX,BUFFER_OFFSET
MOVAL,ES:[BX]
CALLBUMP_POINTER
```

This assumes that a record is kept of the original value for BUFFER_OFFSET, such that BUMP_POINTER could restore it when a buffer pointer had to wrap around past the end of the buffer.

The value of SERIO_SIZE would also be passed to the interrupt setup routine, such that the calling code would be able to define the size of the interrupt buffer.

This offers us a different way to look at handling serial data. For example, let's say that we wanted to send a screen of text from one computer to another over a serial link. One way to get this together would be to fetch each byte from the screen, add in some ANSI escape sequences if there were different colours and such and send the data through the port. At the other end, the receiving software would fetch the bytes from the serial port driver's buffer, as we've seen, and print them to the screen. This is very, very slow.

There's a better way. Suppose we were to define the serial data buffer as being the screen buffer itself. We'll tell the serial port driver that this buffer is at least four thousand bytes long. The sending computer would just transmit the raw contents of its screen buffer and the receiving interrupt handler would place it in exactly the right place, thinking that the screen buffer is really its queue.

It would, of course, be essential to flush the buffer after each screen was sent as allowing the head pointer to actually wrap would be messy. Obviously, there is no need to retrieve characters through the tail pointer in this example, as they will already be in their ultimate destination.

The slick part of this approach is that it really doesn't involve any modifications to the serial driver as we've discussed it thus far. It simply makes clever use of the concept of a circular serial queue.

Rubber Biscuit

This driver is still a long way from being complete. We haven't seen how it gets hooked into the hardware of a PC as yet, nor have we really talked about interfacing it to higher level code. The latter subject is one which calls for a bit of forethought and head scratching to come up with a system which is both flexible enough to make the driver worth using and fast enough to improve on the performance of the BIOS and simple polled communications.

We'll continue to wear the problem down next month. ■