PC Hardware Interfacing Part 13

This month we'll look at a working version of the serial port driver code.

STEVE RIMMER

Over the past few installments of this series we've discussed the development of assembly language routines to deal with the serial port card in a practical way. This has involved the use of interrupts, which are nasty but largely essential. It has also involved the use of some fairly esoteric machine language, as this sort of code works right down there with the bits and monsters of the hardware of a PC.

While we have discussed the code in isolation, we have not seen how it works with real world software as yet. Because most applications programs which might use the serial port will be written in a higher level language, like C or Pascal, we must devise a way for such a program to communicate with our driver.

There are several approaches to this, each with its own advantages and disadvantages. We're going to look at one of them in detail this month.

To Drive or Not to Drive

Over the last few months, we've discussed how machine language functions can retrieve bytes from a circular buffer, bytes which have been placed there asynchronously by the interrupt handler for the serial port card. In devising a strategy to make these functions accessible to an application program, we must come up with a channel of communication between the assembly language code and an application.

There are four things which the machine language driver, whatever that turns out to ultimately be, should offer the application which calls it. These are as follows:

- **A setup routine to initialize the serial port, the interrupt handler and the baud rate.**
- **A character out routine to send data to the serial port.**
- **A character in routine to get data from the input buffer.**
- **A test routine to see if there is any data waiting at the serial port buffer.**

This could be expanded upon, and might well have to be for certain types of applications, but these four functions will handle basic telecommunications through the port.

The PC offers us a number ways to implement these. We'll deal with them here in order of decreasing complexity.

If DOS had its way, it would have you implement the serial port driver as a

PC Hardware Interfacing, Part 13

"device driver". This is a special type of machine language program which DOS loads through the CONFIG.SYS file of your system. The ANSI.SYS file is a device driver which is commonly used.

When DOS boots up, it creates a number of "standard" devices, such as CON and PRN. These behave somewhat like files. If you do something like this:

```
COPY TEXTFILE.DOC PRN
```

the contents of the file TEXTFILE.DOC will be sent to your printer, that is, to the parallel port LPT1. The PRN device looks like a file to DOS, but all its contents go to the printer port. There is also a device called LPT1, by the way, which does much the same thing.

You can write a device driver which will create a new device name and make it accessible to other programs. For example, you could write one which would create a device called SERIAL, such that reading from and writing to SERIAL would handle interrupt driven communications with the serial port.

This is a flexible way to deal with the port, but it has some drawbacks. It means that anyone wishing to use a program which deals with the port this way must have your device driver installed in their CONFIG.SYS file. Device drivers remain in memory, tying some of it up, even when the functions they perform aren't needed. In addition, communication between a program and a device driver is not blindingly fast.

Finally, device drivers are pigs to write.

The next most sophisticated approach is to create a dedicated high speed serial software interrupt. In this case, you would write a memory resident program which would seize an otherwise unused software interrupt and dedicate it to driving your serial port. You would, in effect, be adding another INT service to the PC's BIOS. You could then create subfunctions for this interrupt.

This is a good way to handle some sorts of serial applications, especially those which will be moving a relatively small number of relatively large blocks of data. If you were to create your driver such that you could pass it a pointer and let it hand your application several kilobytes of accumulated data with each request, this approach would have a lot of merit.

It's not too good for character by character communications, however, which tends to be how most serial applications work. Executing an INT instruction takes a considerable amount of time in machine terms, with quite a bit more tied up in the requisite pushes and pops of the handler. This is somewhat at odds with the goal of creating a high speed serial device.

The simplest approach is the least flexible but the most functional. It involves simply writing the driver so it can be 'bolted on" to your application. The driver only exists as part of your program, but it's tightly bound to it and suffers the least time penalty when your program goes to talk to it.

In this example I've assumed that the application program in question will be written in C. However, at the level that this driver works there's very little difference between languages. You'll find it just as applicable to Pascal or compiled BASIC with a minimum of fiddling.

When you compile a C language program the result will be a number of machine language functions which behave in a predictable way. You can't actually write a workable driver in C, because higher level languages rarely generate tight enough code to manage a serial port efficiently. However, you can create assembly language functions which look to C like C functions, except that they're work a lot faster.

A C function takes its arguments as integers pushed onto the stack and returns things in AX register. Under Turbo C, assembly language functions must preserve the SI and DI registers if the compiler is set to allow for the use of register variables. In addition, every assembly language function must take care to either preserve or at least not mangle the BP register, as this is what the function which called it was using to find its stack arguments prior to the call.

Having written a driver which will bolt onto a C language program, all we have to do to use it is to include it in the linking process... adding its name to the project file under Turbo C... and call the functions it provides as if they were included with the compiler.

This is the complete serial port driver written as a machine language module for a C program. I called this SERIO.ASM, so when I assembled it I got SERIO.OBJ.

```
SERIO_SIZE EQU 512 ;THE BUFFER SIZE
A_OFF   EQU 6 ;THE STACK OFFSET

;THIS MACRO SAVES ANY REGS THAT MIGHT BE USED BY THE COMPILER
SAVE MACRO
PUSH SI
PUSH DI
ENDM

;THIS MACRO RESTORES ANY REGS THAT MIGHT BE USED BY THE COMPILER
RESTORE MACRO
POP DI
POP SI
ENDM

SERIO_TEXT SEGMENT BYTE
PUBLIC 'CODE'
ASSUME
CS:SERIO_TEXT,DS:_DATA

;THIS ROUTINE CLEARS THE BUFFER AND RESETS THE POINTER
;
;  CALLED AS
;  SerioReset();
;
PUBLIC _SerioReset

_SerioReset PROC FAR
SAVE
MOV AX,_DATA
MOV DS,AX

MOV AX,OFFSET SERIO_BUFFER
MOV SERIO_HEAD,AX
MOV SERIO_TAIL,AX
RESTORE
RET
_SerioReset ENDP

;THIS ROUTINE SENDS A CHARACTER OUT TO THE SERIAL PORT
;
;  CALLED AS
;  SerioPutch(c);
;  intc;

PUBLIC _SerioPutch

_SerioPutch PROC FAR
PUSH BP
MOV BP,SP
SAVE

MOV BX,[BP+A_OFF]

MOV AX,_DATA
MOV DS,AX

MOV DX,SERIO_BASEPORT
ADD DX,5

SUB CX,CX
SP1: IN AL,DX  ;GET MODEM STATUS
AND AL,20H  ;CHECK CTS AND DSR
CMP AL,20H
JE SP2
LOOP SP1
```

```
SP2: MOV DX,SERIO_BASEPORT
MOV AX,BX
OUT DX,AL
RESTORE
POP BP
RET
_SerioPutch ENDP

;THIS ROUTINE RETURNS A
CHARACTER FROM THE BUFFER
;
; CALLED AS
; c=SerioGetch();
; intc;/*if c&0x0100, no character*/
;
PUBLIC _SerioGetch

_SerioGetch PROC FAR
SAVE

MOV AX,_DATA
MOV DS,AX

MOV BX,SERIO_HEAD
CMP BX,SERIO_TAIL
JNE SG1
MOV AX,0100H
JMP SG2

SG1: SUB AX,AX
MOV AL,[BX]
CALL BUMP_POINTER
MOV SERIO_HEAD,BX

SG2: RESTORE
RET
_SerioGetch ENDP

;THIS ROUTINE RETURNS THE NUM-
BER OF WAITING BYTES
;
; CALLED AS
; c=SerioTest();
; intc;
;
PUBLIC _SerioTest
_SerioTest PROC FAR
SAVE

MOV AX,_DATA
MOV DS,AX

MOV AX,SERIO_HEAD
SUB AX,SERIO_TAIL
RESTORE
RET
_SerioTest ENDP

;THIS ROUTINE RETURNS TRUE IF
CLEAR TO SEND
;
; CALLED AS
; c=SerioReady();
; intc;
;
PUBLIC _SerioReady
_SerioReady PROC FAR
SAVE

MOV AX,_DATA
MOV DS,AX

MOV DX,SERIO_BASEPORT
ADD DX,4
IN AL,DX
AND AL,01H

RESTORE
RET
_SerioReady ENDP

;THIS ROUTINE SETS THE BAUD
RATE
;
; CALLED AS
; SerioBaud(p);
; intp;/*baud rate divisor*/
;
PUBLIC _SerioBaud
_SerioBaud PROC FAR
PUSH BP
MOV BP,SP
SAVE
MOV BX,[BP+A_OFF]

MOV AX,_DATA
MOV DS,AX

MOV SERIO_BAUD,BX

RESTORE
POP BP
RET
_SerioBaud ENDP

;THIS ROUTINE INSTALLS THE
SERIAL VECTOR,
;
; CALLED AS
; SerioOn(p);
; intp;/*com port 1 or 2*/
;
PUBLIC _SerioOn
_SerioOn PROC FAR
PUSH BP
MOV BP,SP
SAVE
MOV BX,[BP+A_OFF]  ;COM PORT
TO USE

MOV AX,_DATA
MOV DS,AX

MOV AX,0
CMP SERIO_SEG,0  ;DON'T IN-
STALL TWICE
JE SM1
JMP SM3

SM1: CMP BX,2
JNE SM2

MOV SERIO_BASEPORT,02F8H
MOV SERIO_VECTOR,0BH
MOV SERIO_MASK,0F7H

SM2: CLI
MOV AH,35H
MOV AL,SERIO_VECTOR
INT 21H
MOV SERIO_SEG,ES
MOV SERIO_OFF,BX

MOV AL,SERIO_VECTOR
PUSH DS
MOV DX,OFFSET SERIO_HANDLER
PUSH CS
POP DS
MOV AH,25H  ;CHANGE THE
SERIAL INTERUPT VECTOR
INT 21H
POP DS

IN AL,21H
AND AL,SERIO_MASK
OUT 21H,AL

MOV DX,SERIO_BASEPORT
ADD DX,3
MOV AL,80H
OUT DX,AL  ;OPEN DLAB

MOV AX,SERIO_BAUD ;GET THE
BAUD RATE DIVISOR

MOV DX,SERIO_BASEPORT
ADD DX,1
MOV AL,AH
OUT DX,AL  ;SEND HIGH ORDER

MOV AX,SERIO_BAUD ;GET THE
BAUD RATE DIVISOR
MOV DX,SERIO_BASEPORT
OUT DX,AL  ;SEND LOW ORDER

MOV DX,SERIO_BASEPORT
ADD DX,3
MOV AL,7  ;AND THE CFW BYTE
OUT DX,AL  ;ALSO CLOSE DLAB

MOV DX,SERIO_BASEPORT
ADD DX,1
MOV AL,01H
OUT DX,AL  ;INTERUPT ENABLE

MOV DX,SERIO_BASEPORT
ADD DX,4
MOV AL,08H
OUT DX,AL  ;MODEM CONTROL

MOV AX,1  ;SAY IT WORKED

SM3: STI
RESTORE
POP BP
RET
_SerioOn  ENDP

;THIS ROUTINE UNDOES THE VEC-
TOR
;
; CALLED AS
; SerioOff()
;
PUBLIC _SerioOff
_SerioOff  PROC FAR
```

```
MOV AX,_DATA
MOV DS,AX

CMP SERIO_SEG,0000H
JE SO1

CLI
IN AL,21H
MOV AH,SERIO_MASK
NOT AH
OR AL,AH
OUT 21H,AL

MOV DX,SERIO_BASEPORT
ADD DX,3
IN AL,DX
AND AL,7FH
OUT DX,AL

MOV DX,SERIO_BASEPORT
ADD DX,1
MOV AL,0
OUT DX,AL

MOV DX,SERIO_BASEPORT
ADD DX,4
MOV AL,0
OUT DX,AL

MOV AL,SERIO_VECTOR
PUSH DS
MOV DX,SERIO_OFF
MOV DS,SERIO_SEG
MOV AH,25H  ;CHANGEINTERUPT
VECTOR
INT 21H
POP DS

MOV SERIO_OFF,0000H
MOV SERIO_SEG,0000H

SO1: STI
RET
_SerioOff  ENDP

;THISROUTINEHANDLESTHE
SERIALINTERUPTFORTHESERIO
SERIO_HANDLER PROC FAR
STI   ;ENABLEOTHERINTERUPTS
PUSH AX
PUSH BX
PUSH CX
PUSH DX
PUSH SI
PUSH DI
PUSH DS
PUSH ES

MOV AX,_DATA
MOV DS,AX

MOV DX,SERIO_BASEPORT
IN AL,DX

MOV BX,SERIO_TAIL
MOV SI,BX

CALL BUMP_POINTER
MOV SERIO_TAIL,BX
MOV [SI],AL
CLI
MOV AL,20H   ;SIGNALENDOFIN-
TERUPT
OUT 20H,AL
STI

POP ES
POP DS
POP DI
POP SI
POP DX
POP CX
POP BX
POP AX
IRET   ;RETURNTOCALLING
CODE
SERIO_HANDLER ENDP

BUMP_POINTER PROC NEAR
PUSH AX
MOV AX,OFFSETSERIO_BUFF-
ER+SERIO_SIZE
INC BX

CMP BX,AX
JGE BUMP_PTR1
POP AX
RET

BUMP_PTR1:
MOV BX,OFFSETSERIO_BUFFER
POP AX
RET
BUMP_POINTER ENDP
SERIO_TEXT ENDS

DGROUP  GROUP _DATA,_BSS
_DATA  SEGMENTWORDPUBLIC
'DATA'

SERIO_BASEPORT DW 03F8H
SERIO_VECTOR DB 0CH
SERIO_MASK  DB 0EFH
SERIO_OFF  DW 0000H
SERIO_SEG  DW 0000H
SERIO_BAUD  DW 0060H
SERIO_TAIL  DW OFFSET
SERIO_BUFFER
SERIO_HEAD  DW OFFSET
SERIO_BUFFER
SERIO_BUFFER DB SERIO_SIZE
DUP(?)
_DATA  ENDS

_BSS  SEGMENTWORDPUBLIC
'BSS'

_BSS  ENDS
END
```

The driver is a bit complex when you first come upon it, but it turns out to be pretty easy to deal with from the point of view of a C program.

We'll discuss the workings of the driver, and how to interface to it, in the next installment of this series ■