PC Hardware Interfacing Part 10

Last month we got the PC serial card working internally, but left it without any way to communicate with the outside world. This time around we'll have a look at finishing up the hardware involved in adding this useful facility to the design.

STEVE RIMMER

Despite the obvious importance of doing so, making our emerging serial card actually able to communicate with external devices is pretty simple. There's no complex binary math to concern ourselves with, and really very little hardware. Of course, the hardware is a bit obtuse, but this is to be expected when one is dealing with a fifty year old standard.

The only tricky thing about serial communications is that it's designed to work with hardware which really predates microcomputers by quite a long while. Whereas everything inside a computer uses predictable logic levels and timing, serial data is a world unto itself. Regrettably, it's something which must be endured, having become entrenched in our universe.

Bilevel Tuba Solo

The first serial devices were teletypes, which puts the origin of this form of communication back several decades. Early teletypes were wholly mechanical, with lots of solenoids and relays and numerous other things too arcane and horrible to contemplate. They did use these devices to synthesize a crude form of electronic logic, however. A teletype took keyboard input, translated it to serial data, sent said data over wires and ultimately turned it into hard copy at the far end of the line.

Big, clunky solenoids that can drive an old style print mechanism do not run cheerfully on five volts. For various reasons, the logic levels of those old teletypes, and hence of our modern serial communications, were such that a positive voltage was considered to be one and a negative voltage zero. Zero volts... and, in fact, anything within several volts of it... was and is undefined.

The original range was forty-eight volts either side of zero. Contemporary serial devices use twelve volts. However, because of the way this system is structured, anything beyond three volts positive is considered to be a logic level of one, and anything beyond three volts negative is a logic level of zero. As such, some devices use plus and minus five volts and get away with it.

The advantage of this bipolar system is actually fairly apparent. Devices which use differing voltage levels can communicate without specialized line drivers. In addition, bipolar logic levels can live with a great deal more voltage loss because of line resistance before they start losing data. This isn't much of a problem now unless you'll be driving serial data over fifty feel of cable, by it impressed the teletype guys to no small end way back in the middle ages.

The only real problem facing the hardware we're about to look at, then, is converting the PC's TTL logic levels to these rather more obtuse ones and back again. As we'll see, there are special parts to do this for us. The 1488 and 1489 chips... as are found in just about every microcomputer serial port design... conveniently change TTL logic levels to serial port logic levels and back again.

The actual serial lines to be interfaced are something of a wonder as well, once again dating back into the mists of prehistory. In theory, serial data can be managed using only three lines, one of which is ground. Labeled TXD and RXD on our schematic, these things send and receive serial information respectively. As long as the hardware at the other end of the serial

PC Hardware Interfacing, Part 10

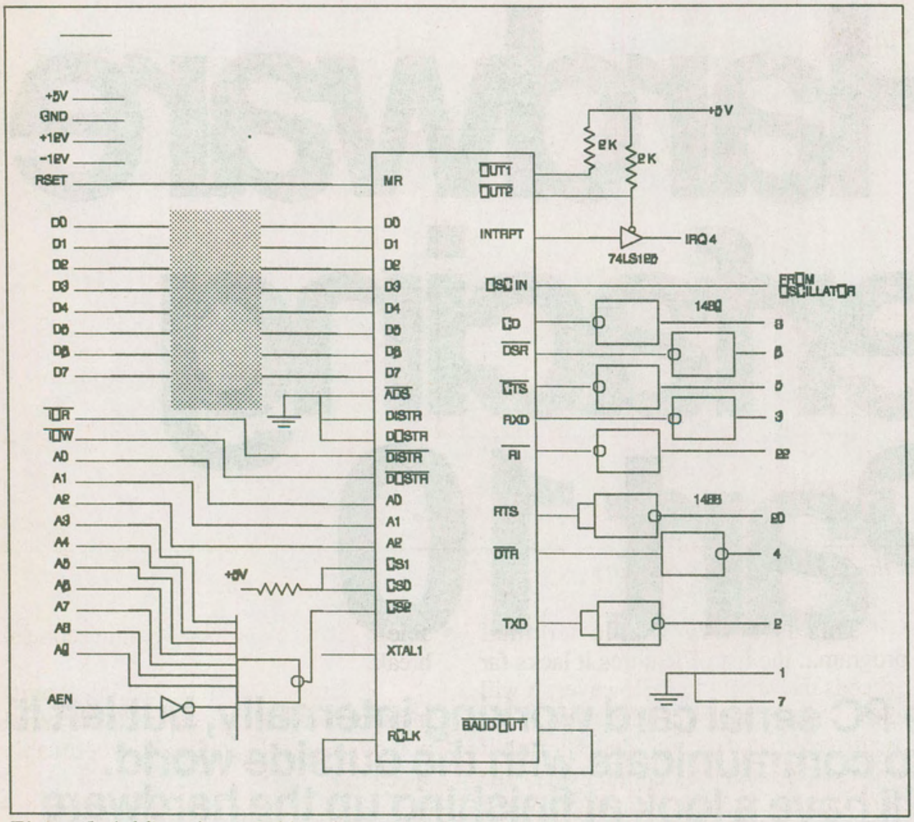

Figure 1. Adding the serial hardware interface. Numbers to the right of the drawing are RS-232C Interface pins.

link is up for dealing with the data at the rate our computer wants to send it, all will be well.

In practice, this *is* usually the case with contemporary computers, but it didn't used to be. As such, serial connections also include some "handshaking" signals which allow the sending computer to ask the receiving computer whether it's free to receive a byte and the receiving computer to answer. The DTR line tells the computer on the other end that the local machine can accept a byte of serial information. It means "data terminal ready". The CTS line inquires as to the status of the remote computer. It means "clear to send".

There's also a "carrier detect" line... marked CD on the schematic... which may or may not be required, depending upon your application. This tells the host computer whether there is a modem carrier present at the serial port's attached device... which only means something if the attached device is, in fact, a modem.

The ring indicator line... RI on the drawing... can be used to tell the program driving the serial port that the telephone is ringing. The 8250 can be programmed to generate an interrupt when this line changes state, so it's quite possible to write software which pops up out of the background only when someone calls in. We'll look at this sort of driver a bit later on.

The serial port's oscillator is a simple crystal and two inverters, a pretty standard way to handle one of these things. The frequency of the crystal is chosen to allow the internal frequency divider of the 8250 to generate common standard baud rates. This is the same frequency that the actual ports in a PC uses, as well as most other implementations of this chip.

I should point out that this schematic is not complete... I've deliberately omitted the power connections and such to the chips in order to keep the wiring as clear and easily understood as possible. If you actually elect to build this card as it stands, you'll want to fill in these details. In most cases, though, you will probably want to build variations on it. Basic serial ports for the PC are as common as politicians at election time and just as easy to find.

Testing the Serial Port

Exercising the card to test *all* its facilities actually requires quite a bit of sophisticated software and hardware. However, if we allow that the actual chip is probably performing properly, we can check the basic circuitry with a few simple code fragments. To make sure that the fundamental communications facilities of the serial port are actually working, let's write a very basic terminal program.

To begin with, you don't actually need a second terminal or computer to test a serial port. Because an RS232 port has separate incoming and outgoing data lines, it can send and receive data at the same time. As such, it's quite easy to test the beast by simply connecting the TXD and RXD lines together... that's pins two and three... and sending data out. If the data comes back, the card's working.

If you have a terminal or modem program for your PC... such as Telix, Q-Modem, CrossTalk and so on... you can test the serial port with this. However, as we'll be looking at writing proper drivers for the port beginning next month, you might feel like getting your fingers dirty with a bit of assembly language now.

In order to talk to the board at its most primitive level, we must initialize the card... set its baud rate and other communications parameters... and then create a loop in which our test program polls for keyboard and serial data. If it finds a character waiting at the keyboard, it sends the character to the serial port. If it finds a character at the serial port, it sends the character to the screen.

There are a few holes in this simple model which make it unsuitable for use as a real terminal... things like interrupts, character translation and a few other things enter into it... but it's good enough for testing.

All of the following assumes that you have jumpered pins two and three of the card. I should point out that this test will work on any serial port configured as COM1 on a PC... you can use a known serial port to test the test program if you're not sure of it.

To begin with, we must set the baud rate. We'll use three hundred baud here, although the card will support up to fifty-six kilobaud if you've used reasonably good parts.

```
MOV DX,3FBH
MOV AL,80H
OUT DX,AL ;OPEN DLAB

MOV DX,3F8H
MOV AL,80H
OUT DX,AL ;SET LOW ORDER BAUD

MOV DX,3F9H
MOV AL,01H
```

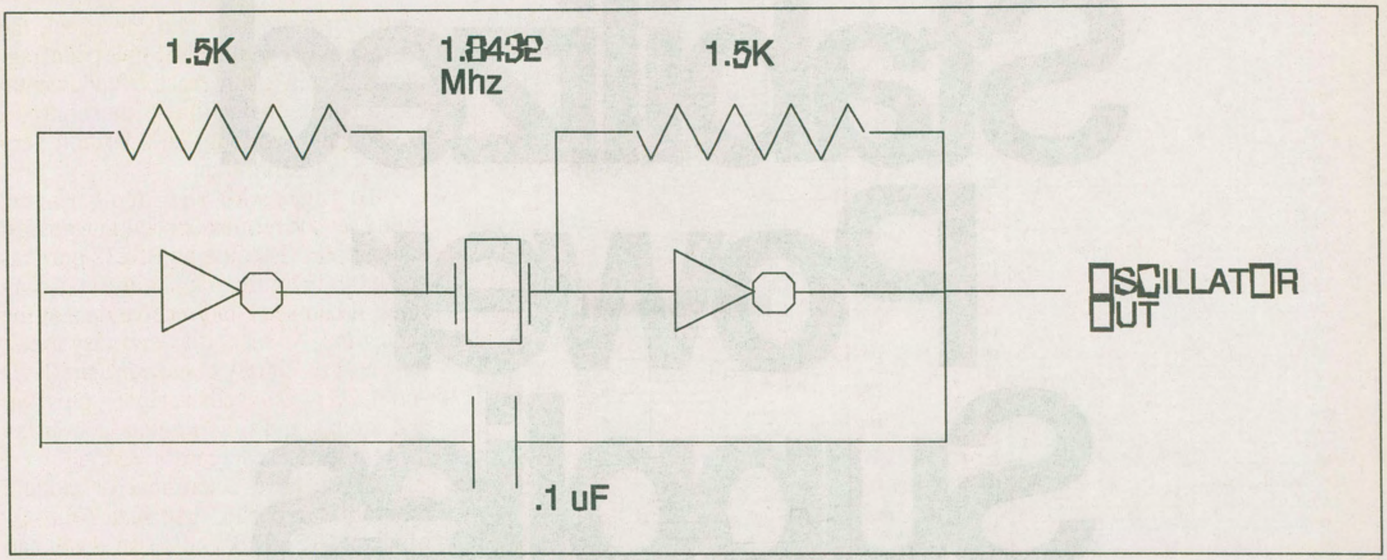

Figure 2. A crystal controlled oscillator to drive the 825D serial port circuitry.

```
OUTDX,AL;SETHIGHORDERBAUD

MOVDX,3FBH
MOVAL,1AH
OUTDX,AL;SETCFW
```

We'll get into how this works in more detail later on.

Next, we have to create the basic terminal loop. This is what it looks like.

```
T_LOOP:MOVDX,3FDH
INAL,DX;GETTHERXPORTSTATUS
TESTAL,1;ISBITSETFORBYTEWAIT-
ING?
JZT_KEY;IFNOT,CHECKTHE
KEYBOARD

MOVDX,3F8H
INAL,DX;IFSO,GETTHEBYTEFROM
PORT

ANDAL,7FH;MASKOFFANYGAR-
BAGE

MOVDL,AL
MOVAH,2
INT21H;PRINTTHECHARACTER

T_KEY:MOVAH,1
INT16H;ISTHEREAKEYWAITING?
JZT_LOOP;IFNOT,CHECKTHEPORT
AGAIN

MOVAH,0
INT16H;IFSO,FETCHIT

MOVDX,3F8H
OUTDX,AL;SENDITOUT

JMPT_LOOP;GOLOOPAGAIN...AND
AGAIN...
```

This is a very simple terminal program... the list of features it lacks far exceeds the ones it has. However, it will allow you to check out a serial port. It's pretty easy to understand what it's up to if you read through the comments to the right of the assembly language. We'll have a proper look at what all the numbers mean starting next month.

You might have noticed that there's no obvious way to get out of this program. A proper terminal would probably provide an escape clause. In this case, being a test program, you can just hit control break to abort the look. It's inelegant, to be sure.

By the way, if you've been following the C language serial which has also been in this magazine over the past few months, you might be interested in seeing how this would be done in C. The following C program would result in much the same actual executing code when it was compiled. If you're into C, you might find this a lot easier to key in than the assembly language routines above.

```
/*setbaudrate*/
outport(0x3fb,0x80);
outport(0x3f8,0x80);
outport(0x3f9,1);
outport(0x3fb,0x1a);

/*terminalloop*/
do{
if(inport(0x3fd) & 0x01)
  putch(inport(0x3f8));
if(kbhit())
  outport(0x3f8,getch());
}while(1);
```

Once again, we're counting on being able to get out of the loop by hitting control break.

Next month we'll be looking at writing some actual drivers to make our serial port do its stuff elegantly and at reasonable speeds. ■