Microcontroller Projects Using the Basic Stamp

Second Edition

Al Williams

CMP Books
Lawrence, Kansas 66046

CMP Books
CMP Media LLC
1601 West 23rd Street, Suite 200
Lawrence, Kansas 66046
USA
www.cmpbooks.com

Acquisition Editor:	Robert Ward
Editor:	Madeleine Reardon Dimond
Layout Design & Production:	Michelle O'Neal and Justin Fulmer
Managing Editor:	Michelle O'Neal
Cover Art Design:	Damien Castaneda

Distributed in the U.S. and Canada by:
Publishers Group West
1700 Fourth Street
Berkeley, CA 94710
1-800-788-3123
www.pgw.com

ISBN: 1-57820-101-2

CMP*Books*

Table of Contents

Chapter 6

Stamp to Internet: Serial I/O

Like a lot of hard core computer guys, I've spent more than my share of time using the UNIX operating system. These days, UNIX receives a lot of criticism for its cryptic command language. For example, to request a list of files, you issue an `ls` command. One way to view a file is to use `cat`. The `cp` command copies you get the idea.

I've often said that anyone who complains that this is cryptic never had to dial into a remote computer at 110 baud on an old-fashioned teletype. At 110 baud, `ls` is the soul of wit. Modems are an example of a serial device. They transmit data one bit at a time (serially) using some prearranged scheme to reassemble the bits into bytes. Contrast this with a parallel printer, for example. A printer that uses a parallel port receives data a byte at a time.

Back in those days, it was common to speculate on when there would be a computer in most homes. That day came earlier than most people expected, but in a surprising way. The average family began buying computers unknowingly in their microwave ovens, their TV sets, and their phones. You can scarcely find a piece of consumer electronics that doesn't have a computer in it now.

Of course, today many homes have "real" computers (or even a Macintosh — sorry, I just can't resist). Even then, these are often little more than really fancy teletypes that connect to the Internet via a modem. It is clear that serial communications — in this case, communications via a modem — has become far more important than anyone expected.

The Stamp's built-in serial commands make it easy to take advantage of serial I/O in your application. In this chapter, I'll explain the basics of the most common serial protocols and show you how to use serial techniques to build some useful tools. Specifically, you'll learn how:

- the most common serial protocols work,
- the processor uses the IC2 serial protocol to communicate with other chips,
- to build a data logger that uses EEPROM to store data,
- to build a Frequency Counter, and
- to connect a Stamp to the web.

Serial communications are especially important to Stamp developers because of the Stamp's limited number of I/O pins. In earlier chapters, you've already seen A/D and D/A that communicate serially. These are ideal for the Stamp because they take two or three I/O pins to connect. A conventional 8-bit A/D chip would require at least eight (and probably nine or 10) I/O pins. A Stamp I only has eight pins to begin with! Even on the Stamp II, wasting nine or 10 pins for one device is usually unacceptable.

Definitions

You can broadly divide serial I/O protocols into two categories: *synchronous* and *asynchronous*. A *synchronous protocol* usually requires a clock and a data line. The devices transfer data in step with the clock. This makes for very simple software, but it has the disadvantage of requiring multiple I/O lines. *Asynchronous protocols* (like RS232, for example) don't require a clock. Instead, the receiver synchronizes on the sender's start bit. Also, there must be at least one stop bit to allow the receiver time to resynchronize. So while an asynchronous transmission only requires one wire (in theory), the sender must transmit at least two extra bits for each byte. This reduces the overall speed of the communications channel.

When discussing serial protocols (especially asynchronous ones), it is common to discuss baud rate. Without splitting hairs, you can think of the *baud rate* as the number of bits per second the protocol allows (for RS232, this is a sufficient definition). Remember, there are at least two extra bits per byte, so a 9,600 baud link can transfer, at maximum, about 960 characters per second.

When dealing with synchronous protocols, it is more common to specify the minimum clock width or the maximum clock frequency that you can use. This also determines the amount of information you can send. Of course, for a synchronous protocol, there are usually no extra bits. So if the maximum clock frequency is 16kHz, you can transfer about 2,000 characters per second.

You might decide that asynchronous communications is a better fit for the Stamp because it requires fewer I/O lines. This isn't always the case. First, in practical use, the transmitter or receiver may require extra lines to inform the other that it isn't ready (handshaking lines). Also, the receiver must constantly watch for start bits so as not to miss any transmissions. The Stamp can only do one thing at a time, so it requires careful design to be sure you receive all serial transmissions.

Simple Serial Protocols

There is code in Chapters 4 and 5 that deals with serial A/D and D/A converters. These converters appear as a shift register to the Stamp. To write to a device like this, you place data on the data line and pulse the clock line. Of course, you have to agree with the device if the data is valid when the clock is high or low, and if the least-significant or most-significant bit is first.

Devices like this are so common that the Stamp II has special commands (`SHIFTIN` and `SHIFTOUT`) just for dealing with these devices. Of course, even with the Stamp I, it is simple enough to mimic the operation of the commands.

Some devices extend the idea of this simple protocol to gain extra functionality. For example, the PAK series coprocessors from AWC use a modified synchronous protocol to provide functions like floating point math or concurrent I/O operations. By using a synchronous protocol, the Stamp can continue working while the coprocessor performs operations.

Interfacing with the PAK-I

The PAK-I is a good example of a device that uses a synchronous protocol that is very similar to SPI (the Serial Peripheral Interface). The chip has five pins that may connect with the Stamp (although you only need two in most cases). By using the shift instructions, the Stamp can command the PAK-I to do floating point math.

The PAK-I has separate input and output data pins to accommodate microprocessors that don't allow you to change I/O pin directions. However, the output is open collector, so you can connect the input and output pins together when you use it with a Stamp. Because the output is open collector, you'll need a pull-up resistor on it in any case. For basic operation, you can just wire the input and output pin to a Stamp pin, and wire the clock line to another Stamp pin.

Some operations you might perform with the PAK-I can take some time to execute. All lengthy commands output a return byte, and that byte will always begin with a `0` bit. This allows the Stamp to check to see if the PAK is ready to respond. If the Stamp is waiting for a response and the data line is high, the PAK is busy. The Stamp can wait until the data is available, or it can do something else and check again later.

The Stamp can also reset the PAK by using a special sequence on the clock and data lines. Normally, when the clock goes high, the data line will remain stable until the clock drops again. If the Stamp changes the data line while the clock is high, the PAK understands that this is a reset and returns to a known state.

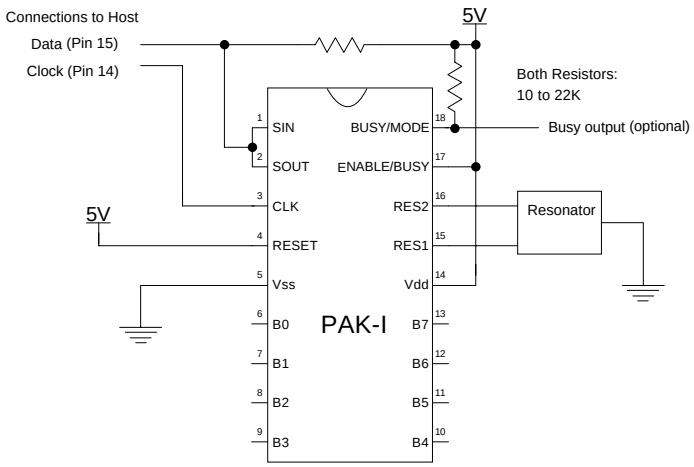
However, there are cases where you might want more than one PAK connected to the Stamp at the same time. In this case, you can wire all the clock and data lines to the same Stamp pins. You'll also use a separate Stamp pin to enable each PAK individually. This line can also reset the PAK, and the PAK can signal its readiness over the same line.

The fifth pin the PAK provides is an alternate busy indicator. If you ground this pin before the PAK powers up, it understands that you want to use the enable line as a busy indicator. In this case, you tie a pull-up resistor to the enable line. To disable the PAK, pull the enable line low. If the PAK is busy, it will pull the enable line low.

For some processors, it is difficult to treat a single pin as both an input and an output. In this case, you can install a pull-up resistor on the alternate busy indicator. Then, the PAK will signify its ready state on that pin, and only read the enable pin. You shouldn't use this mode with the Stamp because it wastes an extra I/O pin.

The circuit in Figure 6.1 shows a typical PAK-I setup. Because there is only one PAK connected, the Stamp doesn't use the enable or the alternate busy indicator pins. The code in Listing 6.1 calculates square roots using Newton's method and prints them on the debugging terminal (this is a nice example, but the PAK-II, a more capable coprocessor, actually computes square roots in hardware, so you probably wouldn't do this in real life). (Discussion continues on page 261.)

Figure 6.1 Typical connections for the PAK-I.



Listing 6.1 Calculating square roots with the PAK-I.

```
' Change these to suit your setup
datap con 15   ' Data pin (I/O)
datapin var in15
clk con 14     ' Clk pin (output)

' Constants for options
FSaturate con $80
FRound con $40

output clk
output datap

fpstatus var byte   ' FPSTATUS
fpx var word        ' Integer used by some routines
fpdigit var byte    ' Digit returned from DIGIT
fpxlow var word     ' The X register low & high
fpxhigh var word
fpb var byte        ' Temporary byte
' The X register in bytes
fpxb0 var fpxlow.lowbyte
```

Listing 6.1 Calculating square roots with the PAK-I. (Continued)

```
fpxb1 var fpxlow.highbyte
fpxb2 var fpxhigh.lowbyte
fpxb3 var fpxhigh.highbyte

gosub freset        ' always reset!

' Square root table
i var word
for i=1 to 1000
fpx=i   ' number
gosub floadint
gosub fsqrt
fpx=2
Debug dec i, "  -  "
gosub fdump
debug "  Error="
gosub fswap
fpx=i
gosub floadint
gosub fswap
gosub fsquare
gosub fsub
fpx=1
gosub fdump
debug cr
next
end


' Reset the Pak1
FReset:
LOW DATAP
LOW CLK
HIGH CLK
HIGH DATAP
LOW CLK
```

Listing 6.1 Calculating square roots with the PAK-I. (Continued)

```
return

' Wait for +,-,*,/,INT,FLOAT, & DIGIT
Fwaitdata:
input DATAP
if DATAPIN=1 then Fwaitdata
return


'Absolute Value
FAbs:
fpb=17
FSendByte:
  Shiftout datap,clk,MSBFIRST,[fpb]
  return

' Store0
FSto0:
fpb=18
goto FSendByte

'Store1
FSto1:
fpb=$92
goto FSendByte

'Rcl0
FRcl0:
fpb=19
goto FSendByte

'Rcl1
FRcl1:
fpb=$93
goto FSendByte
```

Listing 6.1 Calculating square roots with the PAK-I. (Continued)

```
' Load X with fpxhigh, fpxlow
FLoadX:
Shiftout datap,clk,MSBFIRST,[1,fpxb3,fpxb2,fpxb1,fpxb0]
return

' Load Y
FLoadY:
Shiftout datap,clk,MSBFIRST,[2,fpxb3,fpxb2,fpxb1,fpxb0]
return


' Load X with 0
FZeroX:
Shiftout datap,clk,MSBFIRST,[1,0,0,0,0]
return

' Load Y with 0
FZeroY:
Shiftout datap,clk,MSBFIRST,[2,0,0,0,0]
return

' Load an integer from FPX to X
FLoadInt:
Shiftout datap,clk,MSBFIRST,[1,0,0,fpx.highbyte,fpx.lowbyte]
' Convert to Int
Shiftout datap,clk,MSBFIRST,[7]
goto fpstat

' to int
FInt:
Shiftout datap,clk,MSBFIRST,[11]
gosub Fwaitdata
Shiftin datap,clk,MSBFIRST,[fpstatus]
if fpstatus<>0 then FInterr

' Read the X register
```

Listing 6.1 Calculating square roots with the PAK-I. (Continued)

```
FreadX:
fpb=3
gosub FSendByte
ShiftIn datap,clk,MSBPRE,[fpxb3,fpxb2,fpxb1,fpxb0]
fpx = fpxlow
FInterr:
return


' Swap X and Y
FSwap:
fpb=4
goto FSendByte


' X=X*Y
FMult:
fpb=12
fpstats:
gosub FSendByte
fpstat:
gosub FWaitdata
Shiftin datap,clk,MSBPRE,[fpstatus]
return ' status

' X=X/Y
FDiv:
fpb=13
goto fpstats

' X=X+Y
FAdd:
fpb=15
goto fpstats

' X=X-Y
```

Listing 6.1 Calculating square roots with the PAK-I. (Continued)

```
FSub:
fpb=14
goto fpstats

' Get Digit (fpx is digit #) return in fpdigit
FGetDigit:
Shiftout datap,clk,MSBFIRST,[5,fpx]
Fgetdigw:
gosub fwaitdata
ShiftIn datap,clk,MSBPRE,[fpdigit]
return


' Dump a number fpx is # of digits before decimal point
' Assumes 6 digits after decimal point
FDump:
fdj var byte
fdnz var bit
fdjj var byte
fdjj=fpx
fpx=0
fdnz=0
gosub FgetDigit
' Remove this line to print + and space
if fpdigit="+" or fpdigit=" " then Fdumppos
Debug fpdigit
Fdumppos
for fdj=1 to fdjj
  fpx=fdjj+1-fdj
  gosub FgetDigit
  if fpdigit="0" and fdnz=0 then FdumpNext
  fdnz=1
  Debug fpdigit
Fdumpnext
next
Debug "."
```

Listing 6.1 Calculating square roots with the PAK-I. (Continued)

```
for fpx=$81 to $86
  gosub FgetDigit
  Debug fpdigit
next
return

' Set options in fpx
' $80 = saturate
' $40 = round
FOption:
  Shiftout datap,clk,MSBFIRST,[$10,fpx]
  return

FXtoY:
  fpb=$17
  goto FSendByte

FYtoX:
  fpb=$18
  goto FSendByte


' Square Root, set tolerance below
fsqrt:
gosub fsto1    ' R1=target
    ' guess half the original #
fpxhigh=$8000
fpxlow=0
gosub floady
gosub fdiv
gosub fsto0    ' R0=guess
goto fsqrterr
freguess:
gosub frcl0   ' get guess
gosub fsquare      ' square it
gosub fswap   ' put it in Y
```

Listing 6.1 Calculating square roots with the PAK-I. (Continued)

```
gosub frcl1   ' get target
gosub fswap   ' x=guess squared; y=target
gosub fsub    ' subtract
gosub freadx  ' get x to fpxhigh,fpxlow
gosub frcl0   ' get guess
gosub fswap   ' y=guess
fpx=2         ' x=2
gosub floadint
gosub fmult   ' x=2*guess
gosub fswap   ' y=2*guess
gosub floadx  ' x=guess squared - target
gosub fdiv    ' x=x/y
gosub fswap   ' y=x/y term
gosub frcl0   ' x=guess
gosub fsub    ' x=guess-term
gosub fsto0   ' new guess
fsqrterr:
gosub fsquare
gosub fswap
gosub frcl1
gosub fsub
gosub fabs
' Select your error tolerance
' more precise values may fail to converge or take a long time
' check for error<.01
fpxhigh=$7823
fpxlow=$D70A
' check for error <.001
'fpxhigh=$7503
'fpxlow=$126f
' check for error<.0001 - warming may not converge
'fpxhigh=$7151
'fpxlow=$B717
gosub floady
gosub fsub
fpx=0
```

Listing 6.1 Calculating square roots with the PAK-I. (Continued)

```
gosub fgetdigit
if fpdigit="+" then freguess
' Found it!
gosub frcl0
return
End

' X=X**2 ; does not destroy Y, but destroys fpxlow/fpxhigh
Fsquare:
  gosub fswap      ' get y
  gosub freadx     ' save it
  gosub fytox      ' y->x
  gosub fmult      ' x=x*y
  gosub floady     ' restore old y
  return
```

Notice that the code uses subroutine calls to load values, do multiplication, and other operations. At the heart of these subroutines is the simple `SHIFTIN` and `SHIFTOUT` commands — coupled with the busy indication logic, where appropriate.

The PAK-I's big brothers, the PAK-II and the PAK-IX, add various features. For example, the PAK-II has more storage space and handles advanced functions like logarithms, sine, cosine, and others. It also includes 16 spare I/O bits instead of eight. The PAK-IX is just like a PAK-II except it only has eight spare I/O bits. However, in place of the missing eight bits, this PAK has five channels of 10-bit A/D. The PAK can automatically read multiple readings and provide a floating point average (see Chapter 5 for more about averaging A/D readings). Luckily, communicating with these devices is exactly the same as talking to a PAK-I. You simply have extra commands you can issue.

The I2C Bus

The PAK-I protocol is versatile (well, I think so, but then again, I designed it), and well-suited for the Basic Stamp. However, it is hardly unique. There are other protocols, not necessarily designed for the Stamp, that allow you to connect multiple devices to a few microprocessor pins. One of the most popular of these is the Interface for Integrated Circuits (IIC or I2C) bus. This bus originated at Phillips, and there are many devices from a variety of vendors that utilize it. There are other similar standards including Microwire and SPI, but if you can understand I2C, you'll have no trouble with the others.

Listing 6.3 Morse code playback. (Continued)

```
eepoll:
  gosub eecheck
  if i2cackbit=1 then eepoll
  return
```

The I2C code is different in the two listings. In Listing 6.2, I've assumed that the Stamp is the only master and that the EEPROM will always operate more quickly than the Stamp. These are good assumptions and the code is much simpler. There is no worry about arbitrating multiple masters, nor is there any provision for the EEPROM to slow the Stamp down.

Compare the I2C code in Listing 6.2 with the code in the next listing. Here, the I2C code attempts to arbitrate with multiple masters and allow the slave to slow the transfer down. The code is much more complicated. Of course, in this case, it isn't necessary to do these things, so you could easily replace the code in the second listing with the subroutines from the first with no ill effects. Even with this improved code, there is potential for failure in a multimaster system because the Stamp can't always monitor the bus. The only way to really support multiple masters is to employ interrupts, something the Stamp can't do.

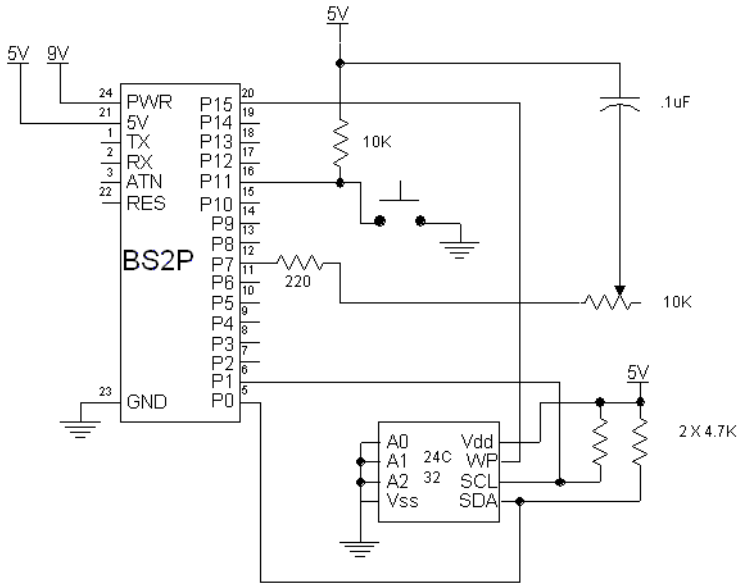
A BS2P Datalogger

An external I2C EEPROM is a good place to store lots of data the Stamp collects. With the BS2P's built-in I2C support, it is even easier to connect an I2C EEPROM to your circuit.

I decided to make a simple data logger with the BS2P. I used an I2C EEPROM for external storage and a resistive sensor for data input. I just used a pot, but you could replace the resistor with a thermistor or light dependent resistor, for example.

The circuit logs data (from the resistor "sensor") to the EEPROM periodically until you push the button. The period in the example is 30 seconds. When you push the button, it dumps the data to the serial port and resets data collection. I didn't go to the trouble of handling the case where the EEPROM is full (there is room for 2,048 samples). The schematic appears in Figure 6.3.

Figure 6.3 A simple data logger.



The BS2P requires the I2C devices to connect to P0/P1 (as shown) or P8/P9. You can't use other pins. The `I2CIn` and `I2COut` commands work like `SERIN` and `SEROUT`. You specify the first pin number (0 or 8) and the device code. Each I2C device has a unique device code. In the case of EEPROMs like the 24C32, the code is `$A0` (for writing) or `$A1` (for reading). In addition, the EEPROM uses the A0, A1, and A2 pins to allow multiple devices on the same bus. Since all of these pins are `0` in the example, the code doesn't change. However, suppose I tied A0 to 5V instead of ground. Then the codes would be `$A2` and `$A3`. This allows you to daisy chain up to seven EEPROMs by giving each a unique 3-bit code on the A0–A2 pins.

The 24C32 needs two bytes of address data, and the `I2CIn` and `I2COut` commands allow this by placing the high byte first, followed by a backslash, followed by the low byte of the address. That means you have to break apart the address into two parts. That's easy enough though. If you have an address, `AD`, you can write:

```
I2COUT 0, $A0, AD>>8\AD&$FF,[whatever]
```

The `WP` pin allows you to write protect the EEPROM. I didn't really use this feature — you could simply ground the `WP` pin if you wanted to save a Stamp pin. The BS2P has a special command that lets you redirect the `READ` and `WRITE` commands to

a different program bank. That means you could use unused program space as EEPROM and not even use an external chip. The STORE command allows you set the bank that READ and WRITE use.

Instead of querying the switch constantly, I used the polling interrupt feature to watch for a switch closure. The POLLIN command lets me specify the pin number (P11) and the desired state (0 is a switch closure). POLLRUN tells the Stamp to run a different program when the pin set by POLLIN reaches the desired state.

Polling isn't enabled by default, however. You have to call POLLMODE to set polling on and specify an action. This is easily the most confusing part to using polling. In this case I set POLLMODE to 4 (although 3 would work just as well). This activates polling and allows POLLRUN to work.

The modes are a bit tricky:

0	Turn off polling and clear all polling setup
1	Turn off polling, but remember all polling setup
2	Turn on polling, allowing POLLOUTPUT and POLLWAIT to work
3	Turn on polling, allowing POLLRUN to work
4	Turn on polling, allowing all commands (POLLOUTPUT, POLLWAIT, and POLLRUN) to work
5	Clear POLLINPUT setup
6	Clear POLLOUTPUT setup
7	Clear both POLLINPUT and POLLOUTPUT setup

Modes 3 and 4 automatically switch to mode 1 or 2 when triggered. This prevents them from retriggering in an endless loop. Modes 5–7 don't actually change the mode, they just clear the action and maintain the current mode.

I didn't use polled output in this example, but you can set modes 8–15 to make the outputs latch instead of following the input state when using polled output.

If all that was confusing, don't feel bad. It is easier to look at the sample code in Listings 6.4 and 6.5. There are three lines at the start of the program. The POLLIN command tells the Stamp to watch the switch and, when it goes to 0, to generate a polled event. The POLLRUN command tells the Stamp that the action you want to take is to run a new program when the event occurs. Finally, the POLLMODE command turns polling on. It stays on forever, so even though the POLL commands only execute once, their effect goes on for the duration of your program. As soon as the Stamp sees the push button turn on, it will run program #1.

Since the Stamp only checks the polled input pins between instructions, it is important not to use a long PAUSE statement. Instead, break your pause statements up into smaller pauses like the example does. This allows the Stamp to check for

polled input often. There is also a POLLWAIT which puts the Stamp in low-power mode until a polled event occurs. You can set how often the Stamp wakes up to check the polled inputs.

There are two programs. The first is the main program (see Listing 6.4) and the second (Listing 6.5) is the push button response program. Notice that it waits for the button to release before it resumes execution of the first program. This prevents immediately executing the push button program again.

Notice there is no easy way to go back to exactly the point you were at when the polled event occurs. In this case, I simply restart the main program which is okay.

I didn't use the automatic assignment of variables because that makes it difficult to share variables between banks. Instead, I name variables manually. For example:

```
adctr    var    w0
```

This causes the program to use w0 as the adctr variable. This makes it easier to share and also to keep from clobbering another program's variables. Just remember that if you use w0 is the same as b0 and b1. The w1 register is the same as b2 and b3, and so on. You can't use w0 and, for example, b1 without causing problems.

Listing 6.4 Program for bank 0.

```
' This is program 0 of the logging example
' Requires a BS2P -- Williams
WCTL CON 15 ' EEPROM WCTL (not actually necessary)
RC CON 7 ' RC input
DUMP CON 11 ' Button goes low for data dump

low WCTL ' Enable EEPROM write (could tie low)

adctr var w0 ' address counter (shared with pgm 1)
state var w1 ' current state
i var b4 ' gp counter

adctr = 0 ' start at 0

' When DUMP button goes low, run program 1 right away
POLLIN DUMP,0
POLLRUN 1
POLLMODE 4
debug "Running...",cr
```

Listing 6.4 Program for bank 0. (Continued)

```
loop:
' Read sensor
HIGH RC
PAUSE 10
RCTIME RC,1,state
' Write to EEPROM
i2cout 0,$A0,adctr>>8\adctr&$FF,[state>>8,state&$FF]
adctr=adctr+2
for i=1 to 60 ' delay 30 seconds in 1/2 second ticks
   pause 500
next
goto loop
```

Listing 6.5 Datalogging program part 2 (bank 1).

```
' Program 1 for the Logging Example
' Requires BS2P -- Williams
DUMP CON 11 ' DUMP button
adctr var w0 ' shared address counter
state var w3 ' non-shared state variable
ad var w4 ' non-shared address counter

' Dump all data
for ad=0 to adctr-2 step 2
   ' read from EEPROM
   i2cin 0,$A1,ad>>8\ad&$FF,[state.highbyte,state.lowbyte]
   debug hex ad, ":", hex state,cr
next
waiting: if in11=0 then waiting ' wait for button release
pause 1 ' wait for bounce
run 0 ' restart log
```

Asynchronous Communications

The chief reason you'll want to do RS232 communications is to talk to a PC via its standard serial port. However, there are other devices that require RS232 (for example, the serial LCDs in Chapter 7). Another reason you might want to perform serial I/O is to communicate with another Stamp over a wire, or a modem. You can also use a modem to dial a phone — perhaps to dial a pager number when some condition occurs.

RS232 Basics

RS232 ports usually use a male 25-pin connector (a DB25) or a male 9-pin connector (a DB9). Therefore, you'll need to connect using a female connector. Terminals, PCs, and similar devices are usually wired as Data Terminal Equipment (DTE). This means that they transmit on pin 2 and receive on pin 3 (at least, on a DB25). DCE (Data Communication Equipment) transmits on pin 3 and receives on pin 2. Every pin that is an input for a DTE device is an output on a DCE device and vice versa (see Table 6.1). Modems and printers are often DCE devices.

You can easily connect a DTE and a DCE device with a straight cable — that is, one that connects every pin to the same pin on the other side of the cable. A cross cable crosses the inputs and outputs.

Table 6.1 shows that there are many signals on the 25-pin connector. However, most devices don't use all of the pins. Commonly, pins 1–8 and pin 20 will take care of nearly all devices. Many cheap cables only wire these pins anyway.

Some devices really don't require any pins except for the send and receive (along with a ground). This leads to three-wire cables (which may be straight or crossed, of course). Some three-wire cables have fake handshaking connections — they just short each side's inputs to the same side's outputs.

If it sounds like getting a good connection between devices is difficult, it is! There are lots of combinations of cables and connections you may have to try. This isn't a problem peculiar to the Stamp — it is part of using RS232. You can purchase a *break out box* (sometimes known as a *BOB*) that lets you monitor the signals on the cable. This can be a big help in troubleshooting. Many BOBs also let you selectively reroute signals so you can experiment with various cross cable configurations.

For RS232 data transfer to work, you must meet several conditions:

- The baud rates must match.
- The number of stop bits, data bits, and parity must match.
- The signal polarity must match.
- The transmitter's voltage levels must be acceptable to the receiver.

- The transmitter's output must connect to the receiver's input and vice versa.
- Any necessary handshaking signals must be enabled.

Table 6.1 RS232 connections.

Signal	Pin (25)	Pin (9)	DTE Direction	Description
PG	1	N/A	N/A	Frame or protective ground
TX	2	3	Out	Transmit data
RX	3	2	In	Receive data
RTS	4	7	Out	Request to send
CTS	5	8	In	Clear to send
DSR	6	6	In	Data set ready
GND	7	5	N/A	Signal ground
CD	8	1	In	Carrier detect
DTR	20	4	Out	Data terminal ready
RI	22	9	In	Ring indicator

The Stamp has a limited number of baud rates, stop bits, data bits, and parity settings. That means you'll usually have to adjust the other device to match the Stamp. Voltage and polarity, however, are more of a hardware issue.

A correctly-designed RS232C device (RS232C is the most common RS232 variant) will produce –12V for a logic `1` and +12V for a logic `0`. Of course, the Stamp doesn't usually have a supply of ±12V so this could present a problem.

The Stamp II's built-in RS232 port assumes that when it is sending data, the other side will not be talking (a good assumption because the Stamp can't listen and talk at the same time anyway). It, therefore, feeds the –12V from the serial input back to the output for a logic `1`. When it wants to generate a 0, it forces the serial output to 5V.

Of course, 5V isn't the normal value for an RS232 0, but most receivers will accept it and it is within the RS232 spec. In fact, most RS232 receivers will work with 5V and 0V, so you can directly connect a Stamp pin to an RS232 input and it will almost always work. The operative word, is almost. If it doesn't work, the equipment manufacturer is not at fault because this is not a standard value for RS232.

Of course, if you do connect 0 and 5V directly to an RS232 input, you'll need to invert the output bits. The Stamp allows you to do this by changing the `baudmode` parameter to the `SEROUT` command (see Chapter 2). For example, using 84 on the Stamp II selects 9,600 baud normal mode and 16,468 selects 9,600 baud inverted.

This is somewhat confusing because normal RS232 data is inverted (–12V is a 1). However, line drivers typically invert the logic signal for you. So a direct connection is inverted, while a connection through a line driver (which inverts the signal) is non-inverted (because the two inversions cancel each other out).

You can receive data directly from an RS232 port, too. The Stamp's input protection diodes will bleed off the excess voltage from the RS232 port. Of course, the diodes will look like dead shorts, so you'll want to put a series resistor (22K or so) between the Stamp pin and the RS232 output. This prevents the short circuit from damaging the RS232 port. Again, you need to invert the bits in software with this method (using an inverted baudmode parameter to `SERIN`).

If you don't want to resort to these kind of tricks, you can use a level converter. The classic converters are the 1488 and 1489 ICs. However, these require ±12V supplies. A more popular alternative is the MAX232 from Maxim. This chip requires four capacitors and uses them in a charge pump to generate ±12V from the 5V supply. The Dallas Semiconductor DS275 is another driver for 5V systems. This chip uses circuitry similar to what you'll find on the Stamp II to route the –12V from the RS232 port back to the input. It then supplies 5V for a logic `0`.

For production designs you really should use a chip like the 1488/1489, MAX232, or DS275. However, for your own purposes, it is often very useful to use direct connection or other methods that don't generate the appropriate voltage levels.

The final piece to the RS232 puzzle is handshaking (sometimes called flow control). Suppose you are trying to send data to a PC via the RS232 port. You start Hyperterminal (the standard Windows terminal program) and you don't see any data. Of course, if your wiring or programming is wrong, you won't see anything. But what if everything seems to be correct (including the points already mentioned like baud rate), but you still get no results? Perhaps Hyperterminal is expecting hardware handshaking. That means that it expects certain voltage levels on RS232 pins (most hardware using DTR, CTS, RTS, and DSR for handshaking). If the program doesn't see the correct signals, it won't send or receive data.

You can fake the correct signals, of course (and some cables do this automatically by shorting, for example, CTS to RTS and DSR to DTR). However, the easiest answer is to simply tell Hyperterminal to use no handshaking. Exactly how you do this depends on your terminal software. For Hyperterminal, bring up the properties for the connection and select `None` in the `Flow Control` section.

Software handshaking (also known as `XON/XOFF` handshaking) relies on a special character to tell the other device to stop sending (`XOFF`). Later, the device will send an `XON` to enable the device to resume transmission. The Stamp is poorly suited for using this type of flow control. In real life, when you send an `XOFF`, there may be several characters still on the way to you before the other device stops sending. Because

the Stamp can't handle input asynchronously, you can only use XON/XOFF in certain situations.

Hardware handshaking fundamentals

Different devices may use hardware handshaking lines differently. For example, a simple serial printer may only use DSR to indicate if it is ready or not. However, you can usually expect handshaking to work something like this:

Action	Meaning
DTE asserts DTR	I am on-line and ready
DCE asserts DSR	OK I'm ready too
DTE asserts RTS	I'm ready to go
DCE asserts CTS	I'm ready to go too
DCE/DTE exchange data	
DCE lowers CTS	Buffer is full — DTE must stop sending
DCE asserts CTS	Ready again
DCE/DTE exchange data	

Use a constant for baudmode and pin numbers

You should rarely if ever specify a fixed number for the baudmode parameter in the SEROUT and SERIN commands. Instead, use a constant value. That way if you have to change how your program works, you can make one or two easy changes instead of having to edit every serial I/O command. So instead of:

```
SEROUT 16,84,[w1]
```

Try:

```
SERPIN con 16
BAUDRATE con 84
SEROUT SERPIN, BAUDRATE,[w1]
```

Open Collector Async

You'll notice that the Stamp's baudmode parameters can be set to an open collector mode. This can be useful where you want to network multiple Stamps together. Of

course, all the Stamps can't talk at one time, so you need some software control. However, with open collector outputs, you can connect all the pins together (along with a single pull-up resistor) and not worry about short circuits damaging the Stamp pins.

As an example of software control, suppose you had one Stamp acting as a master. It might transmit a byte that selects an active slave Stamp. The slave can then send data back to the master until the master selects another slave.

A PC Frequency Counter

There are many ways to measure frequency with a Stamp. Why measure frequencies? Many real-world sensors provide frequency outputs (tachometers, for example). You also might need to measure tones for data communications or control purposes.

You can measure frequency with the `COUNT` command or the `PULSIN` statement. `COUNT` counts the number of pulses in a given amount of time. `PULSIN` measures the width of a single half-cycle.

`PULSIN` measures the amount of time a pulse remains high or low. On the Stamp II, the result of the `PULSIN` command is the length of the pulse (in 2S units). So if the command returns 10, the pulse width is really 20S. This is just the width of a half-cycle. If the duty cycle is 50%, you can multiply the answer by 2. Otherwise, you need to measure a positive and negative cycle and add the results. Of course, this assumes the signal is repetitive (which is usually true).

If you want to convert the raw counts to Hertz (cycles per seconds), you need to multiply by 2 (to get S) and divide by 1,000,000 (to convert to seconds), then take the reciprocal. The problem is that the Stamp is not good at doing this kind of math. The Stamp only does integer math and can only handle numbers to 65,535.

Of course, the reciprocal of $^{x}/_{1{,}000{,}000}$ is $^{1{,}000{,}000}/_{x}$ no matter what x is. Because the Stamp can't divide by 1,000,000, you need to factor the expression. Rewrite $^{1{,}000{,}000}/_{x}$ as $50(^{20{,}000}/_{x})$ and you are in business (because 50 and 20,000 are both numbers that can fit in a word variable). You can find the code for both Stamps in Listing 6.6 through 6.8. The Stamp I code is very similar to the Stamp II code except that the Stamp I measures pulses in 10S units.

Listing 6.6 Basic Stamp I frequency counter.

```
input 8
loop:
pulsin 8,1,w1   ' measure 1/2 cycle
pulsin 8,0,w2   ' measure 1/2 cycle
w3=(w1+w2)*10     ' total cycle time
```

Listing 6.6 Basic Stamp I frequency counter. (Continued)

```
' You could do a variety of things here
' including calculate the resistance of R
' or the capacitance of C (if you are using
' the sample oscillator)
' We will calculate the frequency in Hz
w4=(50000/w3)*20  ' convert to engineering units
      ' obviously the lowest we can resolve
      ' is 20hz this way
      ' although the basic precision is much higher
      ' however with integer only math this is a problem
debug #w4, "Hz", cr
goto loop
```

Listing 6.7 Basic Stamp II frequency counter.

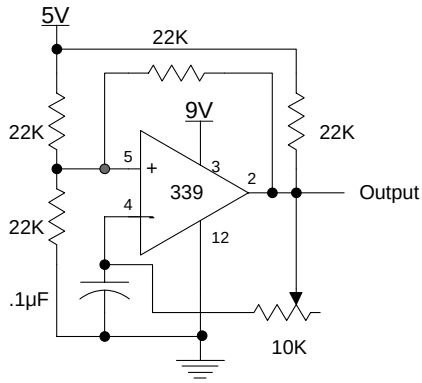
```
v1 var word ' time for 1 cycle
v0 var word ' time for 0 cycle
v var word  ' total time of cycle
f var word  ' computed frequency
input 8
loop:
pulsin 8,1,v1   ' measure 1/2 cycle
pulsin 8,0,v0   ' measure 1/2 cycle
v=(v1+v0)*2     ' total cycle time
' You could do a variety of things here
' including calculate the resistance of R
' or the capacitance of C (if you are using
' the sample oscillator)
' We will calculate the frequency in Hz
f=(50000/v)*20  ' convert to engineering units
      ' obviously the lowest we can resolve
      ' is 20hz this way
      ' although the basic precision is much higher
      ' however with integer only math this is a problem
debug dec f, "Hz", cr
goto loop
```

Listing 6.8 PC interface frequency counter.

```
v1 var word ' time for 1 cycle
v0 var word ' time for 0 cycle
v var word  ' total time of cycle
f var word  ' computed frequency
input 8
loop:
pulsin 8,1,v1   ' measure 1/2 cycle
pulsin 8,0,v0   ' measure 1/2 cycle
v=(v1+v0)*2     ' total cycle time
serout 16,84,[dec v, 13,10]
goto loop
```

If you don't have a frequency source handy you can easily build one with simple parts (for example the LM339 comparator used in the A/D converter in Chapter 5). You can find an example oscillator in Figure 6.4. Just connect pin 2 of the comparator to the Stamp input (pin 8 in the example code). You can vary the resistor or capacitor to vary the frequency. Try touching the capacitor leads and then observe the effect. You could use this circuit as a touch switch. Try replacing the resistor with a sensor that changes with temperature, light, or some other external stimulus.

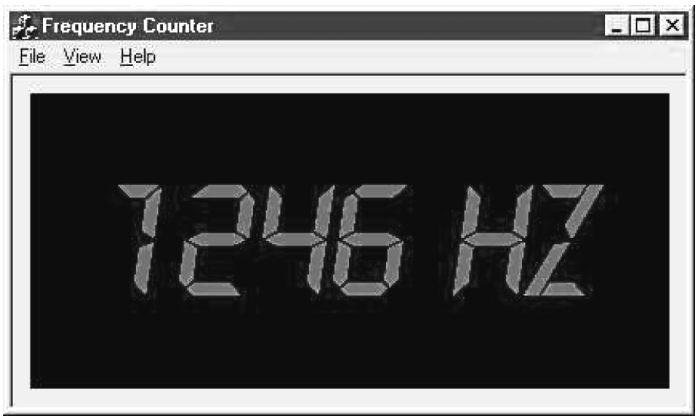
Figure 6.4 Sample oscillator.



If you want to read frequency on a PC, consider the code in Listing 6.8. This is more or less the same code from the previous listing, but it uses SEROUT instead of DEBUG. Also, this code doesn't convert the raw counts to Hertz because the PC does a better job of this. You can find a VB program that reads the frequency in Listing 6.9

and a Visual C++ program in Listing 6.10. Either program will display the frequency in a digital display (see Figure 6.5).

Figure 6.5 The digital display.



Both programs use the MSCOMM ActiveX control from Microsoft. This ActiveX control makes it very simple to use the RS232 port from your programs. When data arrives, the MSComm1_OnComm event fires and it is simple to read the data using the control's Input property. (Discussion continues on page 294.)

Listing 6.9 VB program for the frequency counter.

```
VERSION 5.00
Object = "{648A5603-2C6E-101B-82B6-000000000014}#1.1#0";"Mscomm32.ocx"
Begin VB.Form Form1
   Caption         =   "Frequency Counter"
   ClientHeight    =   3195
   ClientLeft      =   60
   ClientTop       =   345
   ClientWidth     =   5175
   LinkTopic       =   "Form1"
   ScaleHeight     =   3195
   ScaleWidth      =   5175
   StartUpPosition =   3  'Windows Default
   Begin MSCommLib.MSComm MSComm1
      Left            =   3720
      Top             =   2400
```

Listing 6.9 VB program for the frequency counter. (Continued)

```
      _ExtentX        =   1005
      _ExtentY        =   1005
      _Version        =   327680
      CommPort        =   2
      DTREnable       =   -1  'True
      RThreshold      =   250
   End
   Begin VB.Label Label2
     Caption        =   "You have to change the COM port using VB for this Demo.
By default, it uses COM2"
      Height          =   735
      Left            =   240
      TabIndex        =   1
      Top             =   2280
      Width           =   2895
   End
   Begin VB.Label Label1
      BackColor       =   &H00000000&
      Caption         =   "0000HZ"
      BeginProperty Font
         Name            =   "Crystal"
         Size            =   72
         Charset         =   0
         Weight          =   400
         Underline       =   0   'False
         Italic          =   -1  'True
         Strikethrough   =   0   'False
      EndProperty
      ForeColor       =   &H0000FF00&
      Height          =   1815
      Left            =   240
      TabIndex        =   0
      Top             =   240
      Width           =   4575
   End
End
```

Listing 6.9 VB program for the frequency counter. (Continued)

```
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Private Sub Form_Load()
MSComm1.PortOpen = True
End Sub

Private Sub MSComm1_OnComm()
If MSComm1.InBufferCount <> 0 Then
   InpString = MSComm1.Input
   For i = 1 To Len(InpString)
      c = Mid(InpString, i, 1)
      If c = Chr(13) Or c = Chr(10) And aValue <> "" Then
        If aValue <> 0 Then
          Label1.Caption = Int(1000000# / aValue) & "HZ"
        Else
          Label1.Caption = "-0-"
        End If
        aValue = ""
      Else
        aValue = aValue & c
      End If
    Next
End If
End Sub
```

Listing 6.10 Excerpts from a C++ program to read frequency.

```
// FreqCtrView.cpp : implementation of the CFreqCtrView class
//

#include "stdafx.h"
#include "FreqCtr.h"
```

Listing 6.10 Excerpts from a C++ program to read frequency. (Continued)

```
#include "FreqCtrDoc.h"
#include "LedDisplay.h"
#include "FreqCtrView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////
// CFreqCtrView

IMPLEMENT_DYNCREATE(CFreqCtrView, CFormView)

BEGIN_MESSAGE_MAP(CFreqCtrView, CFormView)
//{{AFX_MSG_MAP(CFreqCtrView)
ON_WM_TIMER()
ON_COMMAND(IDM_COLOR, OnColor)
//}}AFX_MSG_MAP
// Standard printing commands
ON_COMMAND(ID_FILE_PRINT, CFormView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_DIRECT, CFormView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CFormView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////
// CFreqCtrView construction/destruction

CFreqCtrView::CFreqCtrView()
: CFormView(CFreqCtrView::IDD)
{
//{{AFX_DATA_INIT(CFreqCtrView)
m_freq = _T("");
```

Listing 6.10 Excerpts from a C++ program to read frequency. (Continued)

```
//}}AFX_DATA_INIT
// TODO: add construction code here

}

CFreqCtrView::~CFreqCtrView()
{
}

void CFreqCtrView::DoDataExchange(CDataExchange* pDX)
{
CFormView::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CFreqCtrView)
DDX_Control(pDX, IDC_MSCOMM1, m_comm);
DDX_Text(pDX, IDC_FREQ, m_freq);
//}}AFX_DATA_MAP
}


CFreqCtrDoc* CFreqCtrView::GetDocument() // non-debug version is inline
{
ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CFreqCtrDoc)));
return (CFreqCtrDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////
// CFreqCtrView message handlers
BEGIN_EVENTSINK_MAP(CFreqCtrView, CFormView)
    //{{AFX_EVENTSINK_MAP(CFreqCtrView)
//}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()
```

Listing 6.10 Excerpts from a C++ program to read frequency. (Continued)

```
void CFreqCtrView::OnInitialUpdate()
{
int port;
CFormView::OnInitialUpdate();
LED.SubclassDlgItem(IDC_FREQ,this);
port=((CFreqCtrApp *)AfxGetApp())->m_comport;
m_comm.SetCommPort(port);
m_comm.SetSettings("9600,N,8,1");
m_comm.SetInputLen(0);
m_comm.SetHandshaking(0); // no handshaking
m_comm.SetRThreshold(0);  // no events
m_comm.SetInBufferSize(4096);
m_comm.SetPortOpen(TRUE);
SetTimer(1,50,NULL);
ResizeParentToFit(TRUE);
}


void CFreqCtrView::OnTimer(UINT nIDEvent)
{
if (m_comm.GetInBufferCount())
{
  VARIANT inputv;
  char* input;
  inputv=m_comm.GetInput();
  int ct=m_comm.GetInBufferCount();
  // got to convert BSTR to CString here
  int len=m_comm.GetInBufferCount();
  input=(char *)inputv.bstrVal;


  while (len--)
  {
  if (*input=='\x0d' || *input=='\x0a')
```

Listing 6.10 Excerpts from a C++ program to read frequency. (Continued)

```
{
if (!value.IsEmpty())
{
unsigned v=strtoul(value,NULL,10);
if (v!=0)
{
float f=1000000.0f/v;
m_freq.Format("%d HZ",(int)f);
}
else
m_freq="-0-";
}
  value.Empty();
}
else
{
  if (*input>='0' && *input<='9') value+=*input;
}
  input+=2; // skip hi byte of UNICODE characters
  }
}
UpdateData(FALSE);
CFormView::OnTimer(nIDEvent);
}



// Set colors
void CFreqCtrView::OnColor()
{
CColorDialog dlg(LED.m_Color);
```

Listing 6.10 Excerpts from a C++ program to read frequency. (Continued)

```
if (dlg.DoModal()==IDOK)
LED.m_Color=dlg.GetColor();
LED.Invalidate(NULL);
}
```

More Power Supply

The last two chapters have contained a Stamp-controlled power supply project. It might seem simple to add RS-232 control to the power supply. However, since the Stamp can only do one thing at a time, including listening for RS-232 input, this takes careful design.

Consider the simple program that appears in Listing 6.11. This is more or less the same power supply program that appears in Chapter 4 but instead of using push buttons, it looks for a + or – character from an RS-232 terminal.

If you run this program you'll notice that not all of the + or – characters you enter will have any effect. That's because while the Stamp is executing the `PWM` command it is not listening to the serial port. You can reduce the time the `PWM` command executes, but that is only useful to a point. Even if you don't build the hardware, try running this program and observe the results on the debug terminal. Then you can experiment with the effect different PWM durations and `SERIN` timeouts have on the program.

Listing 6.11 This RS-232 controlled power supply has limitations.

```
volt var byte
btn1 var byte
btn2 var byte
cmd var byte

volt=0
btn1=0
btn2=0



top:
pwm 0,volt,200
serin 16,84,200,top,[cmd]
if cmd="+" then voltup
if cmd="-" then voltdn
```

Listing 6.13 The VB class module for the I/O extender. (Continued)

```
Stamp_Read = Asc(Mid(RawResult, 3, 1))
End Function
Public Sub Stamp_HiSet(x, y)
comm.Output = Chr(30) + Word(x) + Word(y)
StampResult
End Sub
Public Function Stamp_Alive() As Boolean
comm.Output = Chr(31) + Word(0) + Word(0)
StampResult
Stamp_Alive = RawResult = Chr(0) & Chr(&HAA) & Chr(&H55)
End Function
```

Stamps on the Net

Can a Basic Stamp connect to the Internet? Perhaps not in the traditional sense, but with another computer acting as an intermediary, the Stamp can interact with a network. There are small computers specifically designed for this type of service, but you might want to consider using a PC as a gateway.

Don't be too quick to dismiss using a PC as a slave to a Stamp. The PC doesn't have to be very modern. Older PCs are very inexpensive and plentiful. Also, one PC can handle many Stamps, so the cost per Stamp could be less than using a dedicated Internet connection solution.

There are commercial programs that allow the PC to interpret data and send it over the Internet (like my company's NetPorter software, for example). However, it is simple enough to write a dedicated program using Visual Basic.

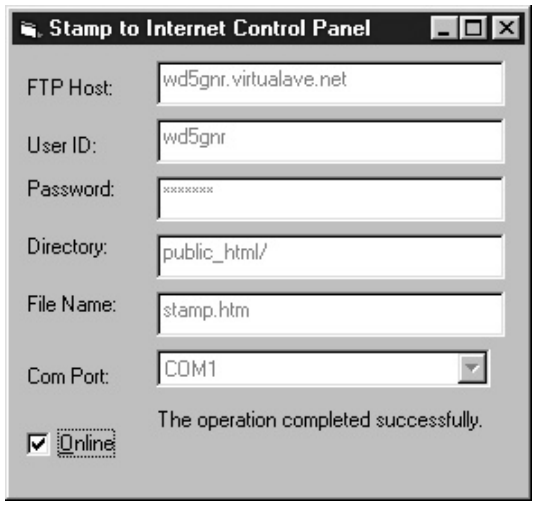
There are two Visual Basic components that make this relatively easy. First, you can use the `MSCOMM` control to read data from a serial port. You've seen this component in the previous project. The other component is the Microsoft Internet Transfer Control. This control can handle HTTP and FTP transfers with a minimum of fuss.

Neither of these controls show up automatically. You have to use the Project | Components menu item to add them to your tool palette. When you place them on your VB form, you'll only see a small icon. The user sees nothing. These components are for your internal use — they don't present a user interface.

Figure 6.6 shows the program in action. You simply set your FTP user ID and password along with the other parameters. The program assumes 9,600, 8, *n*, `1` for

communications parameters. When you make entries, the program remembers them for next time in the registry (except for the password).

Figure 6.6 The Internet bridge program in action.



Once everything is set up, you click Online and the program starts its work. Internet transfers can be slow, so if any errors occur, the program keeps working, but displays any error messages near the bottom of the window. The test Stamp program sends data every five seconds and I didn't have any problems but your mileage may vary.

The only parts of the VB code (Listing 6.13) that may not be obvious are the parts that read the Stamp's data and the FTP code. The way the COM port component is set up, the `MSCOMM1_OnComm` function fires when characters arrive. The logic in this function stores partial strings and uses a carriage return (chr(13)) to mark the end of a data packet. Each data packet is processed through `WriteHTML`.

The `WriteHTML` function creates a temporary file (using the same name your provide on the main form) and writes an HTML file using the data from the Stamp. You could do any post processing you want here and you can add (as I did) server-specific code, formatting, etc. You could even remember, for example, the last 10 values, and show more than one value here.

Once the temporary file is constructed, the program uses the transfer control's Execute method to FTP the data to the server. Error handling is critical here because it is possible for the last operation to still be pending. (Discussion continues on page 311.)

Listing 6.14 The Internet bridge software (Visual Basic).

```
Dim regName As String ' name for registry settings

Private Sub Form_Load()
regName = "StampNet"
MSComm1.Settings = "9600,n,8,1"
MSComm1.RThreshold = 1
Host.Text = GetSetting(regName, "FTP", "Host", "")
UID.Text = GetSetting(regName, "FTP", "UID", "")
' do not save password for security reasons
dir.Text = GetSetting(regName, "FTP", "DIR", "public_html/")
FN.Text = GetSetting(regName, "FTP", "FN", "stamp.htm")
Combo1.ListIndex = GetSetting(regName, "Comm", "Port", 1) - 1
End Sub

Private Sub WriteHTML(s As String)
' If things are still executing, we will get errors here
On Error GoTo WriteErr
' Open temporary file
Open FN.Text For Output As #1
Print #1, "<HTML><HEAD><TITLE>Basic Stamp Data</TITLE>"
Print #1, "</HEAD>"
Print #1, "<BODY BGCOLOR=GRAY>"
' specific for VirutalAve
Print #1, "<!--virtualavenuebanner-->"
Print #1, "<P><FONT COLOR=RED>"
Print #1, "Value="; s
Print #1, "</FONT></P>"
Print #1, "</BODY></HTML>"
Close #1
Inet1.Execute "FTP://" & UID.Text & ":" & PW.Text & "@" & Host.Text, "PUT " &
FN.Text & " " & dir.Text & FN.Text
GoTo EndWrite
WriteErr:
  ' don't worry about it, but try to close file
```

Listing 6.14 The Internet bridge software (Visual Basic). (Continued)

```
  On Error Resume Next
  Close #1
EndWrite:
End Sub


Private Sub Inet1_StateChanged(ByVal State As Integer)
Response.Caption = Inet1.ResponseInfo
End Sub

Private Sub MSComm1_OnComm()
Static inraw As String
Dim inline As String
Dim n As Integer
' when characters are available...
If MSComm1.CommEvent = comEvReceive Then
   Do
    inraw = inraw & MSComm1.Input
    ' find end of line
    n = InStr(inraw, Chr(13))
    If n <> 0 Then
      inline = Left(inraw, n - 1)
      inraw = Mid(inraw, n + 1)
      WriteHTML inline
    Else
      Exit Do
    End If
  Loop While Len(inraw) <> 0
End If

End Sub

Private Sub Online_Click()
If Online.Value = 1 Then
```

Listing 6.14 The Internet bridge software (Visual Basic). (Continued)

```
  Combo1.Enabled = False
  Host.Enabled = False
  UID.Enabled = False
  PW.Enabled = False
  FN.Enabled = False
  dir.Enabled = False
  Response.Caption = ""
  MSComm1.PortOpen = True
Else
  MSComm1.PortOpen = False
  Combo1.Enabled = True
  Host.Enabled = True
  UID.Enabled = True
  PW.Enabled = True
  FN.Enabled = True
  dir.Enabled = True
End If
End Sub

Private Sub Host_Change()
SaveSetting regName, "FTP", "Host", Host.Text
End Sub
Private Sub UID_Change()
SaveSetting regName, "FTP", "UID", UID.Text
End Sub

Private Sub dir_Change()
SaveSetting regName, "FTP", "DIR", dir.Text
End Sub

Private Sub FN_Change()
SaveSetting regName, "FTP", "FN", FN.Text
End Sub
```

Listing 6.14 The Internet bridge software (Visual Basic). (Continued)

```
Private Sub Combo1_Change()
MSComm1.CommPort = Combo1.ListIndex + 1
SaveSetting regName, "Comm", "Port", Combo1.ListIndex + 1
End Sub
```

Of course, you also need a Stamp program to generate data for the Internet. You could use the circuit in Figure 6.3 and the simple program in Listing 6.15.

Listing 6.15 A test program for the Internet bridge.

```
top:
high 7
pause 1
rctime 7,1,w1
serout 16,84,[dec w1,cr]
pause 5000
goto top
```

Summary

You'll use serial I/O to communicate with PCs, RS232 devices, and other peripherals. Making RS232 work need not be a black art, although you do need a logical, methodical approach when things don't go right.

Using serial communications can open the door to complex networked designs as well as designs that use coprocessors, modems, EEPROMs, or those that communicate with a PC. PC communication is especially useful. Imagine a Stamp reading sensors and passing the readings to a PC program which updates a Web page. Of course, you could also store data in a database or a comma-delimited file for input to a database.

Exercises

1. Change the code in Listing 6.1 so that it doesn't use `SHIFTIN` and `SHIFTOUT`. Instead, replace it with the same logic using `HIGH`, `LOW`, and `PULSOUT`.

2. Change the PC Frequency Counter in this chapter to connect directly to an ordinary Stamp pin. What changes do you need to make to the Stamp code? What changes, if any, does the PC software require?
3. Design a PC-controlled pattern generator using the I2C EEPROM circuit in this chapter. The Stamp accepts commands from a terminal program on the PC. Allow commands to set specific EEPROM addresses and the maximum address you want to use. When the PC user issues a `RUN` command, the Stamp will cycle through the EEPROM from address 0 to the maximum address transferring the byte in EEPROM to eight output pins (monitor these with LEDs or a scope). After the Stamp reaches the maximum EEPROM address, it starts over at address 0. If you don't have an external EEPROM, you could use a portion of the Stamp's built-in EEPROM to complete this exercise. This could be the basis for a useful test instrument that could simulate various input sequences for a system under test.

For answers to the exercises, see the Answer Key, page 435.