



Column #44, November 1998 by Jon Williams

Timing Is Everything

One of the regular pokes taken at the Stamp is its lack of interrupts. A fair complaint perhaps, but not all micros have them and quite a few products are built with micros that don't. With a little bit of programming discipline — and a helpful second Stamp — we can write code that emulates a software interrupt.

I recently converted a PIC-based (16F84) project to the BS2. The trickiest part of the conversion was the way I monitored the four sensor (switch) inputs. In the PIC, I have a software interrupt that causes the inputs to be scanned several times per second. With no interrupts in the BS2, I had to use a different method. It's simple, but not necessarily easy. The main code loop has to be fine-tuned so that no matter what path it takes, its execution time is always the same. Geez, how do we do that? The truth is that the process can be very tricky, especially with complex code (so we'll keep our demo simple). I believe the easiest way to write such a program is by creating a state machine. PBASIC makes this very easy with the BRANCH command.

Let's say we want to monitor a couple of inputs and, if each is active beyond some threshold, we take an action. This doesn't sound like a big request, does it? So how would you pull it off? The way that I chose to do it was by monitoring the inputs at a regular interval. If the input is active, a counter is incremented; otherwise, it's cleared. If

Column #44: Timing is Everything

the counter exceeds a given threshold, then some action is taken. The upside of this technique is that the inputs are debounced by the process.

Program Listing 44.1 is a very simple demo using a Stamp II and the Parallax BASIC Stamp Activity Board (BSAC). Two switches (on pins 8 and 9) are monitored. If either of the inputs is held long enough, a corresponding LED is lit (in addition to the input LED). If both are lit at once, they are flashed.

Take a look at the code. What you should notice is that the subroutine ChkSw is always run through each execution of the loop. By trimming the loop timing so that it's constant, we're able to write our processing code based on predictable behavior. In this case, each run through the loop is about 100 milliseconds. So, to alarm on a two-second input, the input would have to be active for 20 iterations through the loop ($20 * 100 \text{ ms} = 2 \text{ seconds}$).

You might be wondering why I chose to use a loop in ChkSw with only two inputs. The reason is that it makes the code very easy to expand, even if it does make it a little tricky to time. There are a couple of PAUSE commands inserted in this code to make the timing as even as possible, regardless of the state of the inputs.

Another thing that I did was use a nibble-sized variable for the alarm state. This technique will give you the ability to have differing alarm levels. As an example, a BS2-based product I developed (and upon which this demo is based) has a secondary threshold for each channel. By monitoring the alarm state for each channel, I was able to set the alarm threshold accordingly.

Okay, the code works, so how do we fine-tune the timing? Some assembly language programmers actually go through the arduous process of counting machine cycles. Thankfully, we don't have to do that. What we're going to do is use a spare pin on our project to talk to another Stamp. The second Stamp will measure our code and display the timing on a Scott Edwards serial LCD.

Listing 44.2 is a program that performs the measurement. The code measures a high pulse on pin 15. Since the resolution of PULSIN is two microseconds, dividing by 500 gives us milliseconds — a timing unit that's a little easier to deal with. Keep in mind that this program is designed to measure short evens. Due to the constraints of PULSIN, the maximum duration that can be measured is 131 milliseconds.

Once you've connected the timing Stamp to your project, you'll need to isolate the section under test. With the section isolated, place the line HIGH x at the beginning of the

section and LOW x at the end of the section (x is the pin number). So long as the fragment is within the limits of PULSIN, the time will be displayed on the LCD. In the demo, ChkSw takes about nine milliseconds to run. I added a couple of pads (PAUSE statements) to keep things as even as possible, regardless of the inputs. With that done, each state routine is checked and padded so that they're equal. With the state processes equal, the last thing to do is pad the main loop to achieve the desired loop timing (100 ms).

How long will trimming a program take? Obviously, that depends on the complexity of the program. And sometimes the process will seem like voodoo. I've found it helpful to make a copy of the program to extract code fragments for test. This prevents me from wrecking an otherwise working program.

Just remember that the purpose of this design style is to emulate a timed software interrupt process. If you need to deal with spurious events, stretch the input(s) with a one-shot (a 555 will do) so that you can see the event when you run your pseudo-interrupt process.

Serial Follow-Up

Every time I work on a project that involves the Stamp communicating with a PC, I'm reminded of Roger Arrick's (of Arrick Robotics) E-Mail tag line: It's harder than it looks. No kidding. I've done two Stamp designs recently that get set-up data from the PC. I thought I'd share what I've been through — perhaps I'll save you a bit of frustration in your projects.

I'm using Visual Basic to develop the PC side of my projects. One of the points that I want to reiterate is to use text mode (versus binary mode) with the MSComm control. Why? It's just plain easy. You take your data and convert it to a string (with Chr\$(x)) to send it. On receipt, you get a string back that can be parsed and converted with Asc(x). Simple, right? You bet. I'm sure there's somebody that will disagree with me, but I've found the binary mode a nightmare and just plain choose not to use it.

Recently, I had a short conversation about the merits of text versus binary with Jan Axelson. She expressed some concern that some bytes might not convert properly (since VB stores strings with Unicode). We independently ran experiments and concluded that this was not the case.

Column #44: Timing is Everything

Just to be safe, you should run your own test, especially if you run a version of Windows that is not a western language (i.e., Japanese or Chinese).

On my end, I created a simple application that sent the value of a spin control to the Stamp. The Stamp program simply waited for the byte and sent it back. A quick check on the transmitted versus received bytes was enough to convince me that text mode would work with my projects.

One of my designs uses the BS2 programming port as the serial connection. I feel a bit like a dope for only recently noticing that the Stamp was echoing anything sent to it. The echo is caused by the hardware design of the serial connection. Take a look at the BS2 schematic in the Stamp manual. Notice the resistor connected between the transmit and receive lines? That's the culprit. It's not a problem though, especially since VB includes some nice string functions.

For example, you want to send a two-byte command to the Stamp and expect a response that is eight bytes long. What you'll want to do is wait for 10 characters to show up and strip the first two (the echoed transmission) with the Mid\$() function. The resultant string can be parsed to extract the Stamp's data.

One of the nice aspects of the BS2 SEROUT function is the optional pacing parameter. Unfortunately, VB doesn't have such a facility. There will be times when you'll want to insert a delay between the characters sent to the Stamp. I borrowed some code from Jan's book, *Serial Port Complete*, to perform a delay. Delays are particularly important when the Stamp needs to do some sort of processing between PC transmissions. Just keep in mind that you might need to adjust your SERIN timeout value if the PC has a delay on its end.

Enough chitchat, let's look at a program (Listing 45.3). I developed this project — EEMover — as a test-bed for uploading and downloading parameters from the Stamp. It also gave me a chance to work with polling versus interrupt reception in VB (we talked about that in the StampNet 1 project).

Let's look at the Stamp side first. Set-up parameters are stored in the first 64 bytes of the Stamp's EEPROM (note that for this demo program the data is completely arbitrary). After this data block is a three-byte revision code that will not be overwritten. Its purpose is to identify the code and features of the attached controller.

The first thing the program does is check to see if the PC is connected. This is accomplished by looking at the state of the receive line. Take a look at Figure 45.1. As

you've seen in many Stamp projects, R1 limits the current into the Stamp from the RS-232 connection. The purpose of R2 is to pull the line high (1) when nothing is connected. As soon as you connect a PC, this line will go low (0). In my production project, I look to see if a PC is connected and jump the code that allows set-up parameters to be changed.

Figure 44.1: RS-232 connection to the BASIC Stamp 2

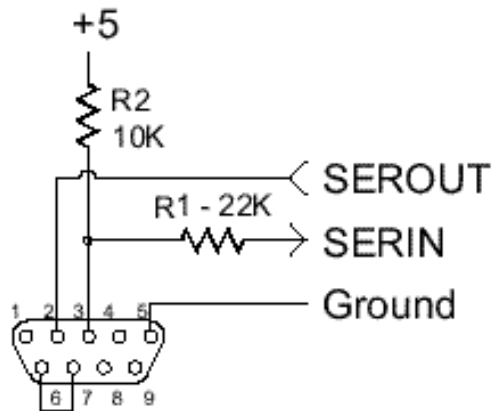
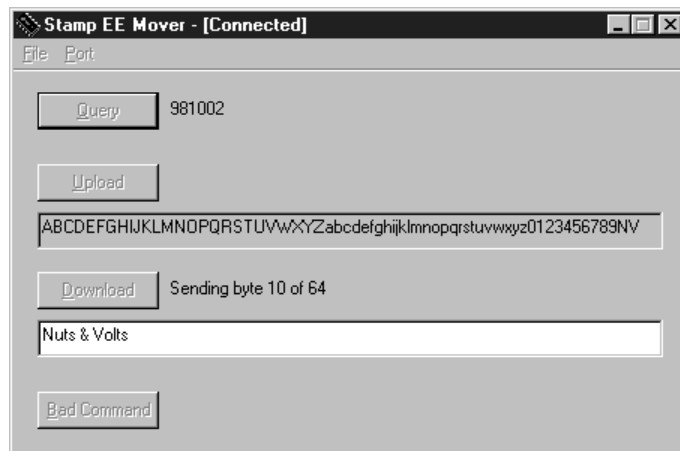


Figure 44.2: EEMover uploads/downloads parameters to a BASIC Stamp



Column #44: Timing is Everything

Even if you don't need to detect a PC, it's good practice to tie the input line to Vcc or ground. This will prevent noise from causing a false start bit on a floating input and corrupting your serial input.

Back to the demo. Once we see the PC, we wait on a start byte (\$FF) and a command. It's a good idea to use the timeout parameter so that your program doesn't hang up if something goes wrong. In this case, we simply loop back to the beginning and look for the PC again. After the start byte and command arrive, the command is converted to a cleaner format with LOOKDOWN, and the program BRANCHes to the code that's appropriate for the command received.

The first possible command is the general query. This causes the Stamp to return the embedded revision code. Once again, the code takes advantage of PBASIC's symbolic constants. Notice the loop control line:

```
FOR addr = RevCode TO (RevCode + 2)
```

The loop reads and sends the revision code bytes to the PC. The nice part about this is that I can move the revision code to any portion of the EE and not change the processing code. Since this is my last opportunity to preach (more later), I encourage you to write programs that are easy to read and maintain. Enough said. Processing the upload command is identical; it just sends more information to the PC.

The trickiest part of the program is downloading new parameters. Once the download command is received, we wait for 64 bytes to be sent from the PC. In order to keep things synchronized and in order, the PC actually sends a start byte (for framing), and the address and data to be written to the address. After reception, the Stamp writes the data to EE, reads it back, then sends the address and data back to the PC for confirmation. By reading back the just-written data, we're able to detect transmission and EE errors. As we'll see in just a minute, the PC will only tolerate a few errors before giving up.

Okay, let's look at the PC code for downloading (Listing 44.4). Here's a basic description before we get to the specifics: After sending the download command, the code will loop through 64 iterations. If there is no data for the loop iteration, a zero is downloaded. After sending the start, address and data byte, the PC waits for the address and data to come back. If the address and data do not come back correctly, another attempt is made. If five attempts are made to the same address and the data still comes back bad, the process is aborted. Whew! There's a lot going on.

The first thing we do is prevent a premature program shutdown so that the Stamp's EE doesn't get trashed and cause problems. This is done by disabling the buttons and menus and setting a flag called "downloading." This flag is checked in the QueryUnload event in case the operator clicks the form's close [x] button.

With the process safe from user-interruption, the command is sent and the program waits for 200 milliseconds for the Stamp to get ready. This is a case where the Stamp SERIN timeout for the downloaded data needs to be longer than 200 milliseconds.

A Do-Loop construct is used for the overall download structure. The control variable `addr` is initialized to 1 before starting the loop and checked for exceeding 64. This structure makes it easy to break out of the Do-Loop by setting the control value to greater than 64.

The first part of the loop core gets a piece of data to send. In this case, we're grabbing a character from a text input field. If you are dealing with non-string data, remember to convert the byte to a string with the `Chr$()` function. In this program, a null string (the input is less than 64 characters) is converted to 0. I used this because some of the elements in my Stamp production are zero-terminated strings. With the character in hand, the start byte, and address and data are sent. As I stated earlier, I decided to send the start byte through each iteration to keep everything in sync.

After the transmission, we wait for two bytes to come back from the Stamp. Notice the use of the `DoEvents` function in the reception loop. This is really important, especially if something goes wrong on the Stamp end. `DoEvents` allows Windows to do other things while processing the loop.

The received input is checked against the data sent. If it's the same, we move on to the next address. If it's not, we'll try the same address again. If the same address fails five times, the loop is terminated and the user is notified.

It all looks pretty easy now that it works, but it was a bear in the process. Good luck with your PC/Stamp projects. They really are a lot of fun when everything is working right.

Farewell, Friends

As those of you that monitor the Stamp list know, this article will be my final regular Stamp Applications column for Nuts & Volts. I stress "regular" because I am not walking away from Stamping and I really do enjoy writing about my projects. It's just that time is scarce for me right now. With a full-time engineering job and a budding career as an

Column #44: Timing is Everything

actor (and all the craziness that goes along with that ...), I honestly don't have the time to write a full column each month. I will submit articles as time permits — including the conclusion of StampNet. I promise.

I really do want to thank Larry Lemieux and his staff at Nuts & Volts for allowing me to write the column this past year and for giving me complete freedom to write about what I chose. Of course, I have to thank my buddy Scott Edwards for trusting me with his creation, for his support, his encouragement, and for his friendship. And I certainly couldn't have done what I've done without the support of Ken Gracey and the good folks at Parallax. Thank you, Parallax, for everything.

Finally, I must thank my dearest friend, Kim Jaech. Who's Kim? Kim is the miracle worker that translates my techno-gibberish into intelligible English. And all along you folks thought I was a pretty good writer, didn't you? Well, not without Kim. Everything that I've ever had published has passed through her capable hands and so long as I have my way, everything I ever write in the future will too. Thanks, Kim. I love you.

For those of you in the Metroplex, perhaps I'll see you at a DPRG meeting or at Tanner's on one of my Saturday afternoon parts runs. For those that I've met via E-Mail, snail mail, or the telephone, thanks for your kind words and valuable suggestions for the column. I sincerely appreciate your comments and your input. May God bless each of you always.

Happy Stamping. I'll see you in the movies!

```
' Program Listing 44.1
' Nuts & Volts: Stamp Applications, November 1998

' ----[ Title ]-----
'
' File..... STATE.BS2
' Purpose... State machine framework
' Author.... Jon Williams
' E-mail.... jonwms@aol.com
' WWW..... http://members.aol.com/jonwms
' Started... 03 OCT 98
' Updated... 04 OCT 98

' ----[ Program Description ]-----
'
' This program demonstrates the general operation of a state machine. Its
' purpose is to monitor and debounce several inputs with near real-time
' precision (emulating a software interrupt-based process).
'
' Run the program on a Parallax BSAC

' ----[ Revision History ]-----
'

' ----[ I/O Definitions ]-----
'
SW0_   VAR    In8           ' switch inputs
SW1_   VAR    In9
LED0_  VAR    Out10        ' LED outputs
LED1_  VAR    Out11

' ----[ Constants ]-----
'
On      CON    0           ' active low output
Off     CON    1

' ----[ Variables ]-----
'
state  VAR    Byte
tics   VAR    Byte(2)
alarm  VAR    Byte(2)
swMax  VAR    Byte(2)
x      VAR    Byte
test   VAR    Byte
```

Column #44: Timing is Everything

```
' ----[ EEPROM Data ]-----  
,  
  
' ----[ Initialization ]-----  
,  
Init:  DirC = %1100  
       state = 0  
  
       LED0_ = Off           ' start with LEDs off  
       LED1_ = Off  
  
       tics(0) = 0          ' clear the input counters  
       tics(1) = 0  
  
       alarm(0) = Off       ' alarms clear  
       alarm(1) = Off  
  
       swMax(0) = 20        ' alarm on 20 "tics" (2 sec)  
       swMax(1) = 50        ' alarm on 50 "tics" (5 sec)  
  
' ----[ Main Code ]-----  
,  
Main:  ' HIGH 0             ' start of timing pulse  
       GOSUB ChkSw          ' check the inputs  
  
       ' process the current state  
       BRANCH state, [Proc0, Proc1, Proc2]  
       GOTO Main  
  
Proc0: LED0_ = alarm(0)     ' update the LEDs  
       LED1_ = alarm(1)  
       PAUSE 50             ' pad to simulate other code  
       GOTO NxtSt  
  
Proc1: test = alarm(0) + alarm(1)  
       IF test > 0 THEN P1x ' if both on, flash  
         LED0_ = Off  
         LED1_ = Off  
P1x:   PAUSE 50  
       GOTO NxtSt  
  
Proc2: ' do something else  
       PAUSE 51  
       GOTO NxtSt  
  
NxtSt: PAUSE 38             ' main loop timing trim  
       state = (state + 1) // 3 ' update the state  
  
       ' LOW 0             ' end of timing pulse
```

```

        GOTO Main

' ----[ Subroutines ]-----
'
ChkSw:  ' check the state of the input switches
        FOR x = 0 TO 1          ' loop through inputs
          BRANCH x, [CS0,CS1]
CS0:    ' IF Sw0_ = OFF THEN SwNo
        GOTO SwYes
CS1:    ' IF Sw1_ = OFF THEN SwNo
        GOTO SwYes
SwYes:  ' PAUSE 1          ' timing trim
        tics(x) = tics(x)+1    ' update input counter
        IF tics(x) < swMax(x) THEN SwNext
          alarm(x) = On        ' set alarm for input
          tics(x) = 0          ' start new counter
          GOTO SwNext
SwNo:   ' PAUSE 2          ' timing trim
        alarm(x) = Off        ' clear the alarm bit
        tics(x) = 0          ' reset the input counter
SwNext: NEXT
        RETURN

```

```

' Program Listing 44.2
' Nuts & Volts: Stamp Applications, November 1998

' ----[ Title ]-----
'
' Program... SPDTEST.BS2
' Purpose... Speed monitor for code sections
' Author.... Jon Williams
' E-mail.... jonwms@aol.com
' WWW..... http://members.aol.com/jonwms
' Started... 23 SEP 1998
' Updated... 23 SEP 1998

' ----[ Program Description ]-----
'
' This program is designed to test code fragments for execution speed. The
' speed is measured in milliseconds in order to allow PAUSE commands into
' the code for speed tuning.
'
' The program monitors a high-going pulse on Pin 15 and displays the
' result (in milliseconds) on a SEETRON LCD. To use, connect a spare pin
' on your project to pin 15. At the beginning of your test section, inser
' HIGH x (where x is your output pin). At the end of the section place
' LOW x to stop the timing.

```

Column #44: Timing is Everything

```
' -----[ Revision History ]-----
'
' -----[ I/O Definitions ]-----
'
LCDpin CON      0          ' SEETRON LCD serial line
PInput CON     15         ' PULSIN pin

' -----[ Constants ]-----
'
N2400 CON      $418D      ' 2400b baud, inverted
Inst  CON      254       ' backpack LCD instruction byte
ClrLCD CON     1         ' clear the LCD
CrsrHm CON     2         ' move to line 1, col 1

' -----[ Variables ]-----
'
pTime  VAR      Word      ' pulse width (2 us units)
msTm1  VAR      Word      ' ms - whole portion
msTm2  VAR      Word      ' ms - fractional portion

' -----[ EEPROM Data ]-----
'

' -----[ Initialization ]-----
'
Init:  PAUSE 1000          ' let LCD intialize

      SEROUT LCDpin,N2400,[Inst,ClrLCD,"Ready..."]
      PAUSE 200

' -----[ Main ]-----
Main:  PULSIN PInput, 1, pTime
      msTm1 = pTime / 500
      msTm2 = pTime / 5 // 100
      SEROUT LCDpin,N2400,[Inst,CrsrHm,DEC msTm1,".",DEC2 msTm2," ms
"]
      GOTO Main

' -----[ Subroutines ]-----
```

```
' Program Listing 44.3
' Nuts & Volts: Stamp Applications, November 1998

' ----[ Title ]-----
'
' File..... EEMOVER.BS2
' Purpose... Stamp to PC Communications Experiments
' Author.... Jon Williams
' E-mail.... jonwms@aol.com
' Started... 30 SEP 98
' Updated... 2 OCT 98

' ----[ Program Description ]-----
'
' This program awaits a PC command and processes it accordingly: send the
' revision code, upload the EE-stored parameters or allow the download of
' a new (64 byte) parameters block.
'
' For convenience, this project was built on a Parallax BSAC. A SEETRON
' 4x40 LCD module was used in debugging.

' ----[ Revision History ]-----
'
' 2 OCT 98 : Finally figured it all out and everything works!

' ----[ I/O Definitions ]-----
'
TXpin  CON    0           ' serial output to PC
RXpin  CON    1           ' serial input from PC
LCDpin CON    15          ' debugging pin (SEETRON LCD)
PGMpin CON    16          ' uses BS2 programming port

PCcon  VAR    In1         ' pc connected?
LEDrx  VAR    Out8        ' receive LED
LEDtx  VAR    Out9        ' transmit LED

' ----[ Constants ]-----
'
N9600  CON    84           ' non inverted (through driver)
T9600  CON    $4054        ' inverted (direct)

Off    CON    1           ' active low
On     CON    0

Yes    CON    0
No     CON    1
```

Column #44: Timing is Everything

```
Start  CON    $FF          ' start byte
CmdQr  CON    $A7          ' query - return date code
CmdUp  CON    $A0          ' upload ee
CmdDn  CON    $A1          ' download new ee

' ---- [ Variables ]-----
'
cmd     VAR    Byte        ' command byte
addr   VAR    Byte
dByte  VAR    Byte
tries  VAR    Byte

' ---- [ EEPROM Data ]-----
'
Setup1 DATA  "ABCDEFGHJKLMNOPQRSTUVWXYZ"
Setup2 DATA  "abcdefghijklmnopqrstuvwxy"
Setup3 DATA  "0123456789"
Setup4 DATA  "NV"

RevCode DATA  98,10,02      ' embedded revision code

' ---- [ Initialization ]-----
'
Init:   DirA = %0001
        DirC = %0011

' ---- [ Main Code ]-----
'
Main:   LEDtx = Off          ' clear the TX/RX LEDs
        LEDrx = Off

        ' look for PC
        IF PCcon = Yes THEN M1      ' look for low on input line
            SEROUT LCDpin,T9600,[12,"PC is not connected"]
            PAUSE 250
            GOTO Main

M1:     SEROUT LCDpin,T9600,[12,"Connected. Waiting..."]
        SERIN  RXpin,T9600,5000,Main,[WAIT (Start), cmd]

        LEDrx = On            ' show transmission received
        PAUSE 50
        LEDrx = Off

        LOOKDOWN cmd,[CmdQr,CmdUp,CmdDn],cmd ' decode command byte
        BRANCH cmd,[CQry,CUpLd,CDnLd]      ' branch to processing
```

```

        GOTO CBad

CQry:  SEROUT LCDpin,T9600,[12, "Query "]
        GOSUB RCode
        PAUSE 1000
        GOTO Main

CUpld: SEROUT LCDpin,T9600,[12, "Upload"]
        GOSUB UpLoad
        PAUSE 1000
        GOTO Main

CDnld: SEROUT LCDpin,T9600,[12, "Download"]
        GOSUB DnLoad
        PAUSE 1000
        GOTO Main

CBad:  SEROUT LCDpin,T9600,[12, "Invalid Command"]
        PAUSE 1000
        GOTO Main

' ---- [ Subroutines ] -----
'

RCode: ' send revision code to PC
        LEDtx = On
        FOR addr = RevCode TO (RevCode + 2)
            READ addr, dByte
            SEROUT TXpin,T9600,[dByte]
        NEXT
        LEDtx = Off
        RETURN

UpLoad: LEDtx = On
        FOR addr = 0 TO 63
            READ addr, dByte
            SEROUT TXpin,T9600,[dByte]
        NEXT
        LEDtx = Off
        RETURN

DnLoad: ' download new set-up data
        LEDrx = On
        SERIN  RXpin,T9600,2000,DLx,[WAIT (Start),addr,dByte]
        LEDrx = Off

        ' show the transmission
        SEROUT LCDpin,T9600,[1,"Writing... ", DEC2 addr, "->",DEC3 dByte]

```

Column #44: Timing is Everything

```
SEROUT LCDpin,T9600,[" (",dByte, ") "]

' copy to ee
WRITE addr,dByte           ' write to ee
READ addr,dByte           ' read it back

' return address and data
LEDtx = On
SEROUT TXpin,T9600,[addr,dByte]
LEDtx = Off

IF addr < 63 THEN DnLoad           ' do 64 bytes
DLx: RETURN
```

```
' Listing 44.4
' Nuts & Volts: Stamp Applications, November 1998

Private Sub cmdDnload_Click()
Dim addr As Byte
Dim tries As Byte
Dim temp As String
Dim x As Byte
Dim response

Call EnableButtons(False)
mnuFile.Enabled = False
mnuPort.Enabled = False
Screen.MousePointer = vbHourglass
downloading = True

' send download command
Call SendHeader(CmdDn)
Call Delay(200)

' download the ee data
addr = 1
Do
lblByteCount.Caption = "Sending byte" & Str(addr) & " of 64"
tries = 0
Do
' get a byte
temp = Mid$(txtDnload.Text, addr, 1)
' if empty, program the location with 0
If (temp = "") Then temp = Chr$(0)
' build the buffer (adjusted address + byte)
txBuf = Chr$(addr - 1) & temp
' send start byte and buffer
MSComm1.Output = Chr$(Start) & txBuf

' wait for two-byte response
Do
```

```
DoEvents
  x = MSComm1.InBufferCount
Loop Until (x = 2)
rxBuf = MSComm1.Input

' increment tries counter
tries = tries + 1
Loop Until (tries = 5) Or (rxBuf = txBuf)

' exit if bad chip
If (tries = 5) And (rxBuf <> txBuf) Then
  response = MsgBox("Error: Cell " & Str(addr), vbOKOnly)
  addr = 65
End If

  addr = addr + 1
  Call Delay(150)
Loop While (addr <= 64)

lblByteCount.Caption = ""
Call EnableButtons(True)
mnuFile.Enabled = True
mnuPort.Enabled = True
Screen.MousePointer = vbDefault
downloading = False
End Sub
```

