Column #41, July 1998 by Jon Williams:

Remote Control Stamping

Microcontroller networking is not new — I'm just a little late to the party. Honestly, I've been wanting to control one (or more) Stamps from my PC for a long time. Inspired by Ryan Sheldon's terrific articles on PC-based RS-232 control, I decided to stop thinking about networking Stamps and start doing it. The cool thing about using the BASIC Stamp is that we can design our network nodes to do anything we need or desire; we are not restricted to off-the-shelf components.

What I'm going to present this month is the front-end of a project that I generically call StampNet. And I really should correct my last statement. This isn't a singular project as much as a series of software, firmware, and hardware building blocks designed to guide and inspire you to build your own Stamp-based network projects. Sound good? Okay then, let's dig in.

The Big Picture

Since we're not going to squeeze all of this into one installment, let's take a look at the big picture. The overall concept is divided into three major elements: the "master" control software, a communication module, and "slave" (remote) nodes. Of course, there is software in all three elements, and hardware at the communication module and slave nodes.

Like Ryan, we're going to create our master control software using Visual Basic. Our slave nodes will be custom-built around the BASIC Stamp. As I've already pointed out, we're not going to connect the master software directly to the slave nodes; there will be an intermediary that I call the communication module or "master node." The purpose of the master node is two-fold: it buffers communication between the fast PC and the relatively slow Stamp-based slave nodes, and it gives us the opportunity to handle local I/O and other network-type (X-10) control.

The Master Node

Based on the last sentence, you've probably guessed that we're going to use a BS2 for the master node. Why? Several reasons, actually. Using the BS2 gives us the opportunity to do local I/O, X-10 control and connects directly to the PC through a serial cable. All this for about the same price as an RS-232 to RS-485 adapter. And, unlike a standard adapter, we can modify the BS2's behavior to suit our particular needs.

Stage 1

This month, we'll work with the heart of our communications network by building a small demo program. The purpose of this program — PCTOBS2 — is to establish a serial link between the PC and the master node. With the link established, we'll be able to pass information back and forth.

Since I didn't want to fuss with hardware at this point, my temporary master node is the BASIC Stamp Activity Board (BSAB). If you don't have one, download the documentation (PDF format) from Parallax for the I/O schematics. The demo master node program uses the four LEDs, the POT input, and the analog (PWM) output circuitry. It's all very straightforward stuff. You can easily construct the circuit on a solderless breadboard.

For convenience sake, the master node communicates with the PC through the programming port (defined as pin 16 in PBASIC). You can change this if you like, but with the pin's built-in level shifters, it's really the easiest route to take. If you do build your own hardware, be sure to put a 0.1 uF capacitor inline with the ATN pin on the Stamp. The capacitor blocks the steady state of the PC's DTR output and allows the "attention" pulse generated by the BS2 programming software. The capacitor is already installed in the BSAB and I've updated my BS2 carrier boards with the same arrangement. I recommend that you do the same.

Alternately, you can install an in-line switch, but you'll have to remember to close it for programming and to open it when running your VB program(s).

Visual Programming

Since Ryan Sheldon has done such an excellent job explaining the construction of a Visual Basic program, I'm not going to cover all the gory details. I will cover those things that I consider good practice or that will bite you when you're not looking (ask me how I know they'll bite …). If you skipped Ryan's articles in the March and April ’98 issues, please go back and read them now.

The most important thing to keep in mind is that Visual Basic is an event-driven language, unlike PBASIC, which is essentially linear in nature. Okay, what do we mean by "event-driven?" In the simplest terms, nothing happens in an event-driven program until something (an event) happens! Sounds a bit like the chicken and the egg, doesn't it? It's really not that bad.

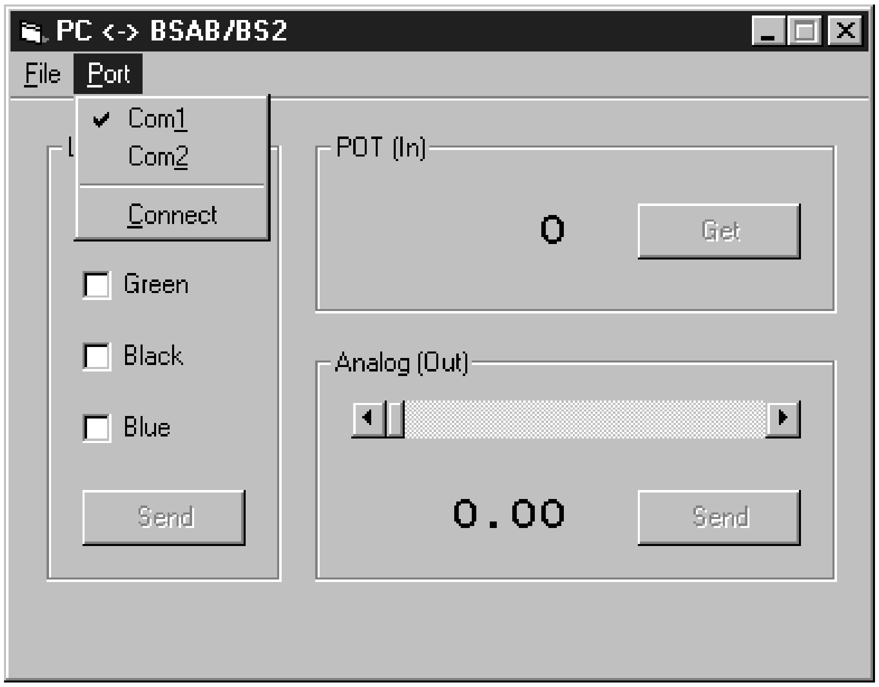
Events are generated in myriad ways: the program starts, a button is clicked, a character is received from the serial port, etc. As VB programmers, it is our job to write the handlers (subroutines) for the events that affect us. Since most of it is very simple coding, I'm not going to analyze the entire VB program; I'm just going to cover the serial communications stuff necessary to talk with the Stamp.

Getting Connected

Figure 41.1 shows our VB program just after start-up with the Port menu selected. In my opinion, it's good practice to allow your program to open and close the serial port when needed. The purpose of the Port menu, therefore, is to allow the user to select which port to use and then to connect when ready. Listing 1 is the code for the subroutine that handles the connection.

In operation, the "Connect" menu item changes to "Disconnect" when a successful connection is made. I could have used a separate menu item, but this just seemed nicer and more intuitive. As you can see, the code checks the menu caption to determine the current connection state. If the caption is "&Connect" (the leading ampersand underlines the "C" to define a keyboard shortcut), we know that we're not connected and can open the serial port.

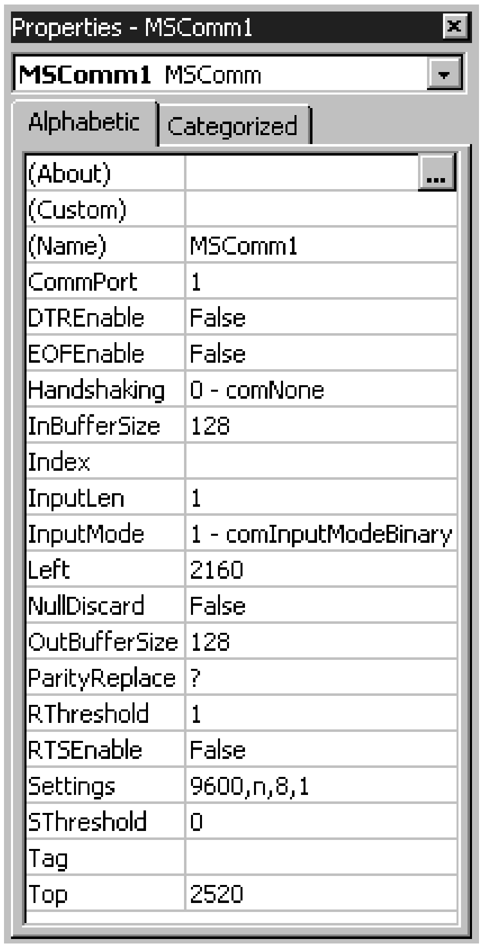
Figure 41.1: Visual BASIC Program started with COM port selected



It attempts to open the serial port, I should say. Remember that, in our multi-tasking environment, several programs could be running and any one of them might be using our target serial port. It's a good idea to notify the user (especially yourself) when there is a problem. Notice that just before we try to open the port, we define an error trap. What this line (On Error GoTo NoConnect) does is force the program to branch to the label "NoConnect" if there is an error. Since we've defined the error trap just before attempting to open the serial port, it's pretty safe to assume that any error would have been caused by that attempt. Advanced programmers may want to check the actual error code before proceeding.

For the moment, let’s assume that the port was in use. Since our program can't open the selected port, an error is generated and the program branches to "NoConnect." You can see that, in this section, we display an error message and then turn off the error trapping. This prevents other errors from bringing us back to the "NoConnect" label. Pretty simple and very useful. This is particularly important if you plan to distribute your programs. Okay, now let's go the other way. When the port is opened, we turn off the error trapping and then tell the user we've made a successful connection by changing the caption of our form (you can see this in Figure 41.2).

Figure 41.2: Successful COM port connection



(About)	
(Custom)	
(Name)	MSComm1
CommPort	1
DTREnable	False
EOFEnable	False
Handshaking	0 - comNone
InBufferSize	128
Index	
InputLen	1
InputMode	1 - comInputModeBinary
Left	2160
NullDiscard	False
OutBufferSize	128
ParityReplace	?
RThreshold	1
RTSEnable	False
Settings	9600,n,8,1
SThreshold	0
Tag	
Top	2520

Since we're connected, we change our Port menu item to "Disconnect" and disable the port selection items. This is done for safety. We don't want the user changing serial port properties while the port is in use. The last step in the connection process is to enable our transmit buttons (labeled "Send" and "Get" on the form). Since this happens in several places in the program, I created a custom subroutine called "SetButtons()" to do the work in one place.

Disconnecting is just a matter of undoing everything that we just did. Notice the "Exit Sub" in the line just after the connect/disconnect section. This is important. It forces the subroutine to end so that the error trapping code is not executed under normal conditions.

Talking To The Stamp

Let's move on. Now that we're connected, we have access to the transmit buttons. There is one transmit button for each of the respective areas of the program: setting the state of the LEDs, reading the POT, and outputting an analog voltage. When we click one of these buttons, a message — and sometimes data — is sent to the Stamp for processing. If the message was a request for information (like the POT reading), the Stamp will return data for us to process and display.

Since I'm a simple guy at heart, I like to keep things simple. Along that theme, I decided on fixed-length messages between the Stamp and the PC. Sending messages from the PC is done with a subroutine called "SendMsg." Receiving information from the Stamp happens in the background (automatically, based on the arrival of characters at the serial port) in the less cleverly named "MSComm1_OnComm." I'll explain the name and its operation in a minute. First things first. Let's send a message to the Stamp.

Take a look at Listing 41.2, the code for sending a message to the Stamp. As I already pointed out, our message will have a fixed length; in this case, four bytes, a header ($55), the node address, the message and a data byte that might be needed for the particular message. The header gives the Stamp something to look for before reading in the rest of the message. This helps keep things in sync. The address byte specifies the target node for this message. We will use 1 as the address for the master node. Next comes the command and data (i.e., LED status). Now that we know what it does, let's see how it's done.

The first thing we do before actually sending the message is to disable our transmit buttons. In practice, you'll probably never notice this as it takes less than five milliseconds to transmit four bytes at 9600 baud. I think it's good practice, however, just in case something goes wrong. We don't want to stuff any more messages into a transmit queue that's not functioning properly.

Since the MSComm object deals with strings, a string (msgQueue) is built from our message bytes. To aid in the development, I created a hex version of this string (prefixed with "TX: ") and put it on the form. As it turns out, I never had any trouble sending information to the Stamp. With the message string built and displayed, we send it to the Stamp by copying it to the output buffer of the MSComm object.

After placing the message in the buffer, the program waits for it to be sent by monitoring the number of bytes left in the output buffer. The "DoEvents" keyword embedded in this Do-Loop is very important as it lets the program handle other events while in the loop. If you left this out, and the transmission got hung up, you could end up having to terminate your program forcefully.

With the output buffer empty — meaning our message has been sent — we clear the input buffer of any garbage and wait for a message to come back. We also save the message number we sent (for later processing of any returned data) and re-enable the transmit buttons.

Listening To The PC

The PC code wasn't too tough, was it? Now let's look at the BASIC Stamp end of things — that's even easier. Listing 41.3 is the BS2 program that processes the messages from the PC. Of course, we'll add to it in the future, but you can plainly see that the structure is set up for easy adaptation.

There's really nothing to it. We wait for a message that begins with $55, then store the address, message number, and data. Since this demo doesn't deal with slave nodes, it throws away any messages not destined for the master node. Next month, we'll update this code to pass the message on to the target slave.

You may remember we talked about the BRANCH command a couple of months ago. This program demonstrates a very typical use of BRANCH. Now, I know that the astute — and that's all of you — will observe that this may be just a little too convenient since we're only using three messages and the numbering is contiguous (1, 2, and 3).

Later, when we get a little more sophisticated and add messages, it may not be convenient to number them in this fashion. No problem. We can use LOOKDOWN to sort things out for the BRANCH command.

The subroutines are trivial, so I'm not going to discuss them except to say that each is responsible for setting the value of sData that will be returned to the PC. The last line of each subroutine is a GOTO that routes the program to the response code. Like at the PC, we start the message with $55, send our address, the return data, and finish up with a carriage return. The carriage return lets the PC program know that the message is complete. This works well since the message is sent back as a string in hex format.

The last thing we need to do to complete this phase of our project is receive the message at the PC and process the data. Unlike the Stamp, the PC does not sit and wait for the response; it goes happily about its business until characters show up at the serial port.

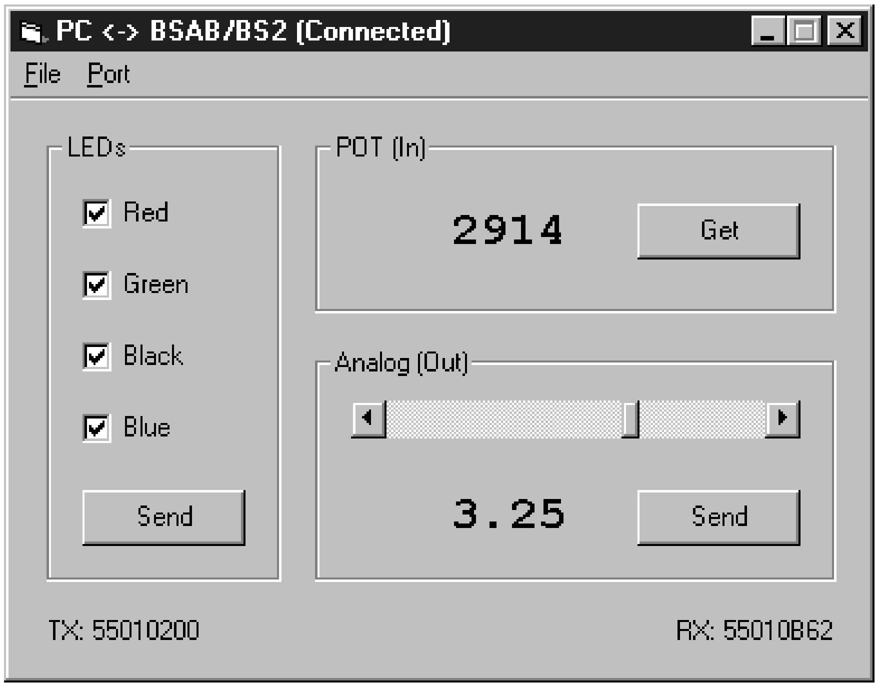
Listening For The Stamp

Remember just a few minutes ago we talked about event-driven programming? Well, here it is. I commented earlier that an event can be generated upon the arrival of a character at the serial port. We're going to take advantage of this behavior to deal with the data as it shows up. You may be wondering why I chose event-driven code over polling (waiting for the data). I did this because the same code can be used in projects where the PC asks for data and in projects where a Stamp just freewheels information into the PC as it pleases. We will not be freewheeling in a multi-Stamp network, but you may want to build a Stamp-based data collector, for example, that sends data to the PC without being prompted. This is where event-driven programming can be very useful.

Just a quick note on set-up. Figure 41.2 shoes the settings I used for the various MSComm properties. Notice that the property RThreshold (receive threshold) is set to one. This forces a "receive" event to be generated each time a character shows up. You'll need to set this as the default value of zero (no event).

"OnComm" is a generalized handler that is called when any event is generated by the MSComm object. Listing 41.4 is the code for this subroutine. Since we only care about receiving characters at the moment, the first thing the subroutine does is check for that event (comEvReceive). If it is a receive event, we grab the character from the input buffer. If the character was a carriage return, we know that the message is complete and we can process it. If the character was anything else, we add it to our holding string and wait for another character.

Figure 41.2: VB program after successful transaction with the Stamp



2914

3.25

TX: 55010200

RX: 55010B62

And remember, all this happens while the program is doing other things.

The first step in checking the message is to look for our header ($55) at the beginning of the buffer. Theoretically, it should always occupy the first two characters of the received string, but I found when writing the program that this wasn't always the case. For reasons that I still don't understand, junk seemed to precede the message from time to time. The Mid$ function takes care of removing any junk characters that may have found their way into the front end of our message.

If the "cleaned" message contains the correct number of characters, we put it on screen and call the subroutine "ProcessInput" to display the returned data.

Figure 41.3 shows the program after a successful transaction with the Stamp. Notice that the transmitted message was "55010200." Broken down, we see our header ($55), the node number (01), the "Get POT" command (02), and a data byte of $00. You can see that the Stamp returned "55010B62." This string tells us that the message came from node 01 and the data passed back was $0B62.

Since the transmitted command was "Get POT," the data is converted to a decimal (2914) and displayed on the form.

If the message is not eight characters in length, we assume that it's bad and display "RX: Bad Packet" on the form. Here's another one of those voodoo programming things. I wrote this error handling code because I was getting bad messages about 25% of the time. That's a problem. And when I modified my Stamp program to deal with single-character commands and hooked it up to a terminal program, I got no such junk. I was so frustrated that I started saying words we can't print here and I even resorted to asking my goldfish for advice! He was no help, so I'm going to buy a new one as soon as I can get to the pet store (just kidding, I like my goldfish — even if he doesn't know duff about computer programming!).

Okay, here's the truth. In a moment of desperation, I did a search on the term "mscomm" using Windows Explorer and found a demo program buried deep in my hard drive. Low and behold, it was a terminal program written in VB. When I ran it, no junk! What I noticed was something very interesting in the MSComm settings: the input mode was set to binary (versus text mode that I had been using).

I tried it. It worked. No more junk. Hooray! Why did it work? Beats the tar out of me. If one of you VB gurus will explain it to me, I'll share it with everyone else.

The code works, so play with it — as is — as a starting point before getting adventurous, particularly with the MSComm object properties. While it may not seem like much, we've actually got quite a lot of power with the code we have in hand. The PC has just become a display device for the Stamp. This leads to all kinds of possibilities with graphics, long-term data storage, sharing with other programs; the sky is really the limit.

What's Next?

Next time, we'll create programs that listen to a free-wheeling stamp, send X-10 commands, and send messages to remote Stamp nodes. For reference, you should probably go back and read the June 1997 installment of this column (written by Scott Edwards) on RS-485 networking with Stamps. Have fun. I'll see you next time!

```
' Listing 41.1
Private Sub mnuPortConnect_Click()
  If mnuPortConnect.Caption = "&Connect" Then
    ' connect
    ' trap comm port error
    On Error GoTo NoConnect
    MSComm1.PortOpen = True
    ' connected - remove error trapping
    On Error GoTo 0
    frmMain.Caption = "PC <-> BSAB/BS2 (Connected)"
    ' update Port menu
    mnuPortConnect.Caption = "&Disconnect"
    ' disable comm port selection while connnected
    mnuPortCom1.Enabled = False
    mnuPortCom2.Enabled = False
    SetButtons (True)
  Else
    ' disconnect
    MSComm1.PortOpen = False
    frmMain.Caption = "PC <-> BSAB/BS2"
    mnuPortConnect.Caption = "&Connect"
    mnuPortCom1.Enabled = True
    mnuPortCom2.Enabled = True
    SetButtons (False)
  End If
Exit Sub

NoConnect:
  ' display port connection problem
  lblTransmit.Caption = "Error: Could not connect"
  On Error GoTo 0
End Sub

Public Sub SendMsg(addr As Byte, msg As Byte, data As Byte)
  Dim msgQueue As String * 4
  Dim msgData As String
  Dim x As Byte
  Dim temp As Byte

  ' disable buttons while sending message)
  SetButtons (False)

  ' build message string
  msgQueue = Chr$(&H55) & Chr$(addr) & Chr$(msg) & Chr$(data)

  ' display transmit message (in hex)
  msgData = "TX: "
  For x = 1 To 4
    temp = Asc(Mid$(msgQueue, x, 1))
    msgData = msgData & HexStr(temp)
  Next
  lblTransmit.Caption = msgData

  ' send message to Stamp
  MSComm1.Output = msgQueue
  ' wait until output buffer is empty
  Do
    DoEvents
  Loop Until MSComm1.OutBufferCount = 0
  ' clear the input buffer
  MSComm1.InBufferCount = 0
  ' save the message number
  lastMsg = msg

  ' re-enable buttons
  SetButtons (True)
End Sub
```

```
' Listing 41.2
Private Sub MSComm1_OnComm()
  Dim temp As String
  Dim i As Integer

  If MSComm1.CommEvent = comEvReceive Then
    ' get a character from the buffer
    temp = StrConv(MSComm1.Input, vbUnicode)
    ' if CR, then process the data
    If temp = Chr(13) Then
      ' look for header
      i = InStr(1, buffer, "55")
      ' remove *trash* preceeding header
      If i > 0 Then
        buffer = Mid$(buffer, i)
      End If
      If Len(buffer) = 8 Then
        ' go process the input
        lblReceive.Caption = "RX: " & buffer
        ProcessInput (lastMsg)
      Else
        lblReceive.Caption = "RX: Bad Packet"
      End If
      ' clear the  buffer
      buffer = ""
    Else
      ' add character to buffer
      buffer = buffer & temp
    End If
  Else
    ' process other events here
  End If
End Sub
```

```
' Listing 41.3
' Nuts & Volts: Stamp Applications, July 1998


' -----[ Title ]-----------------------------------------------------------
'
' File...... PCTOBS2.BS2
' Purpose... PC To BSAB/BS2 Demo
' Author.... Jon Williams
' E-mail.... jonwms@aol.com
' WWW....... http://members.aol.com/jonwms
' Started... 31 MAY 1998
' Updated... 31 MAY 1998


' -----[ Program Description ]---------------------------------------------
'
' This program receives a command+data message from a corresponding PC
' program. The PC message is structured thusly:
'
'   $55, address, message, data
'
' The Stamp will respond with:
'
'   $55, address, high_byte, low_byte, CR
'
' Note: Not all messages will return data, but the Stamp will always send
' four elements to the PC. All bytes are returned in hex format, followed
' by a carriage return that serves as an end-of-packet marker.
'
' BASIC Stamp Activity Board jumper settings:
'
'   X2: Pos 2
'   X6: In


' -----[ Revision History ]------------------------------------------------
'
' 31 MAY 98 : Rev 1


' -----[ Constants ]-------------------------------------------------------
'
Baud96  CON     84                      ' serial baud rate (to PC)
SIOPin  CON     16                      ' use programming port
                                        '  - couple ATN with capacitor
PotPin  CON      7                      ' pot input on BSAB
```

```
AnPin   CON     12                    ' analog (PWM) pin on BSAB


' -----[ Variables ]----------------------------------------------------
'
addr    VAR     Byte                  ' node address
msg     VAR     Byte                  ' message
pcData  VAR     Byte                  ' data byte
sData   VAR     Word                  ' return data
sDHigh  VAR     sData.HighByte
sDLow   VAR     sData.LowByte


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


' -----[ Initialization ]-----------------------------------------------
'
Init:   OUTC = %1111                  ' clear LEDs
        DIRC = %1111                  ' make LED pins outputs


' -----[ Main Code ]----------------------------------------------------
'
Main:   ' wait for message from PC
        SERIN SIOPin, Baud96, [WAIT ($55), addr, msg, pcData]

        ' ignore message not addressed to this node
        IF addr <> 1 THEN Main

        ' jump to process called by msg
        ' 0 is not a valid message
        BRANCH msg, [Main, DoLEDs, GetPot, AnOut]

        GOTO Main


' -----[ Subroutines ]--------------------------------------------------
'
DoLEDs:' invert bits for active low outputs on BSAB
        OutC = pcData ^ $0F
        ' clear response data
        sData = $00
        GOTO Rspnd

GetPot:' read pot and return value to PC
        HIGH PotPin
        PAUSE 5
        RCTIME PotPin, 1, sData
        GOTO Rspnd
```

```
AnOut:  ' output the analog value
        PWM AnPin, pcData, 500
        ' clear response data
        sData = $00
        GOTO Rspnd

Rspnd:  ' output to PC
        SEROUT SIOPin, Baud96, [hex2 $55, hex2 addr, hex4 sData, 13]
        GOTO Main
```