Programming Microchip 28-DIP PICs through a USB port

My favourite microcontroller has been the 28-pin 16F872. For many years, I have used a home-built programmer connected to the parallel port of an old IBM ThinkPad. The ThinkPad had an Intel 80386 processor and the software was written in Turbo C. The 80386 processor could be configured to run as an 80286 processor, which was the last processor Intel ever made in which the user could get direct access to the I/O ports. And, at the time, Turbo C was the only version of "C" for which one could get a compiler for free on the internet. That old ThinkPad does not even have any USB ports!

But times have changed. Microchip has eliminated the 16872 and the nearest replacement, the 16F882, has two 14-bit configuration words. My old programmer will not burn the new chips. It is time for a new programmer.

How to convert a USB port to an old-school serial port

I want the programmer to be as simple as possible, both in software and hardware. Parallel ports have gone the way of the dodo bird, so the next simplest means of connecting the programmer to a laptop is through the serial port. Unfortunately, serial ports have also gone the way of the dodo bird. Modern laptops, including my current laptop do not have serial ports. They only have USB ports.

Building true USB functionality into a programmer is a lot of work, and completely unnecessary. There are USB-to-serial port adapter cables which save time and trouble. The cable I used is shown in the following photograph. The USB plug is at one end, and a DB9 plug (a 9-pin serial port male connector) is on the other. The cable is more complicated than it looks. Buried inside the DB9 plastic casing is a microprocessor which interfaces USB communication with a host computer on one side to serial port communication with a peripheral device (in this application, the programmer) on the other.

These USB-to-DB9 adapter cables come in a range of prices. The internet warned me off using cables that are too cheap. It may be that the cheapest ones do not implement all of the pins on the serial port. Prices vary with the length of the cable. Prices also vary with the version of USB used. I used a mid-priced cable, which cost about thirty Canadian dollars. I did not experience any problems.

How to tell the Host computer that a serial port connection has been made

From this point on, I will use the term "Host computer" to describe the laptop computer which will run the main program, which will be written in VisualBasic. How do we arrange things so that a VisualBasic form can get access to this new port?

Plug the USB end of the adapter cable into the Host computer, leaving the DB9 connector on the other end hanging free. The microprocessor inside the adapter cable will establish a communication link with the Host computer in the normal way that USB devices do. Among other things, it will tell the Host computer that it is a serial port. Go into Control Panel on the Host computer, and look at the hardware which is available. The list under the heading `Ports (COM & LPT)` will now include another port which was not there before. My Control Panel describes this new port as `Prolific USB-to-Serial Comm Port (COM5)`. (It looks like my adapter cable was manufactured by a company called Prolific.) What's important is the descriptor `COM5`. This is the name that my Host computer gives to the adapter cable. If you plug the USB end of the cable into a different USB port, it will be given a different name, like `COM3`.

We can use the SerialPort class in VisualBasic to carry out this identification programmatically. In fact, the SerialPort class contains all the methods and stuff that are needed to manage the serial port. The following VisualBasic code is a stripped-down main form which shows how to select and open a serial port.

```
Imports System.IO.Ports

Public Class Main
    Inherits System.Windows.Forms.Form

    Public WithEvents SerialPortToPIC As SerialPort
    Public ListOfSerialPorts() As String

    Public Sub New()
        Text = "PIC_Programmer"
    End Sub

    Private Sub Main_Load() Handles Me.Load
        ' This subroutine runs automatically when the form is loaded
        ' Detect all serial ports on the computer
        ListOfSerialPorts = IO.Ports.SerialPort.GetPortNames()
        ' List the serial ports in the comboboxCommPorts control
        For I As Int32 = 0 To UBound(ListOfSerialPorts) Step 1
            comboboxCommPort.Items.Add(ListOfSerialPorts(I).ToString)
        Next I
        ' Take the first item in the list as the default value
        comboboxCommPort.Text = comboboxCommPort.Items.Item(0).ToString
    End Sub

    Public comboboxCommPort As New System.Windows.Forms.ComboBox With
        {.Size = New Drawing.Size(75, 25), .Location = New Drawing.Point(5, 5),
        .Text = ""}

    Public WithEvents button_OpenPort As New System.Windows.Forms.Button With
        {.Size = New Drawing.Size(75, 25), .Location = New Drawing.Point(5, 100),
        .Text = "Open serial port", .TextAlign = ContentAlignment.MiddleCenter}
```

```
    Private Sub buttonOpenPort_Click() Handles buttonOpenPort.MouseClick
        ' Instantiate a serial port
        SerialPortToPIC = New SerialPort
        ' Store the name of the selected serial port
        SerialPortToPIC.PortName = comboboxCommPort.Text
        ' Open the selected serial port
        SerialPortToPIC.Open()
        ' Set the communication properties
        SerialPortToPIC.BaudRate = 9600
        SerialPortToPIC.DataBits = 8
        SerialPortToPIC.StopBits = StopBits.One
        SerialPortToPIC.Parity = Parity.None
        SerialPortToPIC.Handshake = Handshake.None
        SerialPortToPIC.ReceivedBytesThreshold = 1
        SerialPortToPIC.ReadTimeout = SerialPort.InfiniteTimeout
    End Sub

End Class
```

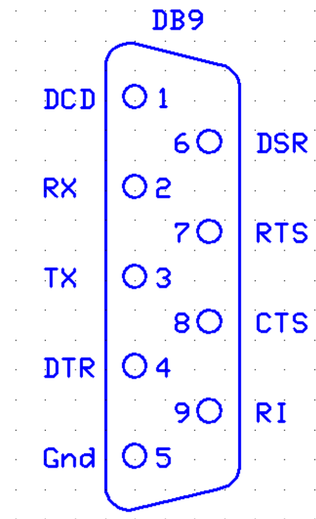
`SerialPortToPIC` is the name I chose for the variable which is the handle to the serial port. When the form loads, subroutine `Main_Load` runs automatically. It invokes the method `GetPortNames()` to place the names of all available serial ports into an array of strings `ListOfSerialPorts()`. These names are then listed in a combobox control named `comboboxCommPort`, where the user can peruse the list visually and click on the appropriate port name. The form also has a button control which displays the text `Open serial port`. When the user clicks on this button, the handler subroutine `buttonOpenPort_Click()` will run. It instantiates `SerialPortToPIC` as a new serial port pointing to whichever port the user selected in the combobox. The serial port is actually opened when the method `Open()` executes. The last eight lines of the handler subroutine show the properties which I assigned to this serial port. The first five properties are pretty obvious. I specified a transmission speed of 9600 bits per second, eight data bits per "information packet", one stop bit and no parity bits. As I will describe in more detail below, the most basic RS-232 communication protocol through a serial port uses only three of the nine lines available inside the DB9 cable. When the protocol has no handshaking, the other six lines are not used at all. The `ReceivedBytesThreshold` parameter is the number of "information packets" which the Host computer's input hardware will receive before it alerts the Host computer's system, and thus the VisualBasic application, that it has received some bytes. Because I developed this application in such a way that the Host computer and the programmer communicate by exchanging single bytes, it is necessary to alert the system after the arrival of even a single "information packet". Lastly, let me deal with the timeout feature. The SerialPort class allows the developer to specify a maximum interval of time the application should wait for incoming packets. I have ignored, or disabled, this feature by specifying an infinite timeout interval.

The six non-participating lines in the DB9 cable

Permit me to digress a bit by describing the lines which do not participate in communication between the Host computer and the programmer. The following diagram is a pin-out of a DB9 connector. The nine pins are arranged in two rows, one having five pins and the other four. For the purposes of this discussion, it doesn't matter whether this is a male or female connector. I have shown in the diagram the pin numbers and the mnemonic names by which they are usually identified.

The accompanying table sets out the full name of the pins. The names have historical significance, dating back to the days when users got their access to mainframe computers through what were called "dumb terminals". Datasets being ready and Clears to send were the types of notices that devices sent to each

other to let each other know about what to expect next. (To be more precise, it was voltage changes on these lines which gave notice.) What these lines used to be used for is not really important. What is more important is what they can do and could be used for if we so wished. The table includes a column which describes how the Host computer views these lines, either as voltage outputs or as voltage inputs.



Pin	Nmemonic	Full name	Host view
1	DCD	Data carrier detect	Input
2	RX	Receive data	Input
3	TX	Transmit data	Output
4	DTR	Data terminal ready	Output
5	Gnd	Signal ground	
6	DSR	Data set ready	Input
7	RTS	Request to send	Output
8	CTS	Clear to send	Input
9	RI	Ring indicator	Input

The most basic RS-232 communication protocol uses the receive and transmit pins (RX and TX) and the signal ground, which is the reference against which the voltage on all the other pins are compared. Of the other six pins, two (DTR and RTS) are outputs from the point-of-view of the Host computer and four (DCD, DSR, CTS and RI) are inputs. It turns out that the SerialPort class gives a VisualBasic application direct access to five of these six other pins. (The Ring indicator line cannot be accessed by SerialPort.)

Let me give an example. We could add the line `SerialPortToPIC.RtsEnable = True` to the VisualBasic code given above, perhaps as another line at the end of the handler subroutine. Execution of that statement will cause the voltage on the RTS line (pin 7) to go high. Executing the statement `SerialPortToPIC.RtsEnable = False` would cause the voltage on that line to go low. Similarly, setting the `DtrEnable` property True or False causes the voltage on the DTR line (pin 4) to go high or low, respectively.

Exactly what are the high and low voltages? They will be determined by the USB-to-DB9 adapter cable you use. The `Prolific` adapter cable I used produces +9V and -9V for the high and low voltages, respectively. I do not know how much current my cable can deliver, but I know from experiment that it is more than enough to light up an LED with a series 220Ω resistor.

I believe that the original specification for RS-232 devices required that they recognize as positive any and all voltages in the range +3V to +25V and as negative any and all voltages in the range -3V down to -25V. Later on, the voltage extremes were narrowed to ±15V.

Now let me give an example of how to control the lines which the Host computer sees as inputs. The VisualBasic expression `SerialPortToPIC.CtsHolding` will return a Boolean value of True or False depending on whether the voltage on the CTS line (pin 8) is high or low, respectively. `DcdHolding` and `DsrHolding` return the Boolean values for the DCD line (pin 1) and the DSR line (pin 6), respectively.

One could ask: how high or low must the input voltages be for the adapter cable to distinguish between them? My experiments showed that the `Prolific` adapter cable I used is actually much better than I expected. It can operate at TTL signal voltages. The RS-232 specification was set back in the days when

peripheral I/O devices were electro-mechanical and needed big currents (and correspondingly big voltages) to operate. Now that most devices are fully electronic, voltage extremes like ±25V are not needed. Furthermore, modern devices are much better at discriminating between different voltages. As I have said, my `Prolific` adapter cable interprets +5V as high (`CtsHolding` returns True) and 0V as low (`CtsHolding` returns False).

Why we cannot use these non-participating lines as a synchronous communication system

In principle, we could build our Host-to-programmer communication system using the two output lines (RTS and DTR) and two of the input lines (CTS and DCD, to pick two of the three), plus the ground line as a reference. The PIC inside the programmer would connect four of its I/O pins to the four signal lines. Each device would then have a dedicated pair of lines to the other device, one for control and one for data. They could send bits back-and-forth whenever they chose. That would be perfect. In fact, my first design for this application did just that. But it failed. Miserably.

The `RtsEnable` and `CtsHolding` commands, and the similar commands for the other I/O lines, work in the blink of an eye. For some applications, including this one, eye blinks are much too slow. It turns out that these commands have a built-in latency period between 11 milliseconds (for a single processor Host computer) and 16 milliseconds (for a multi-processor Host computer). For example, it will be 11ms after `RtsEnable` is set True before +9V appears on the RTS pin. This is thousands of times too slow. At a rate of 22ms per bit, the Host computer could send only five bytes to the programmer per second.

There is no way around these latency periods, which are not related to the time slicing Microsoft uses to allocate CPU time.

Fortunately, the RX and TX lines, which RS-232 uses, are not subject to this latency. Unfortunately, the SerialPort class does not have commands which give direct access to these two lines.

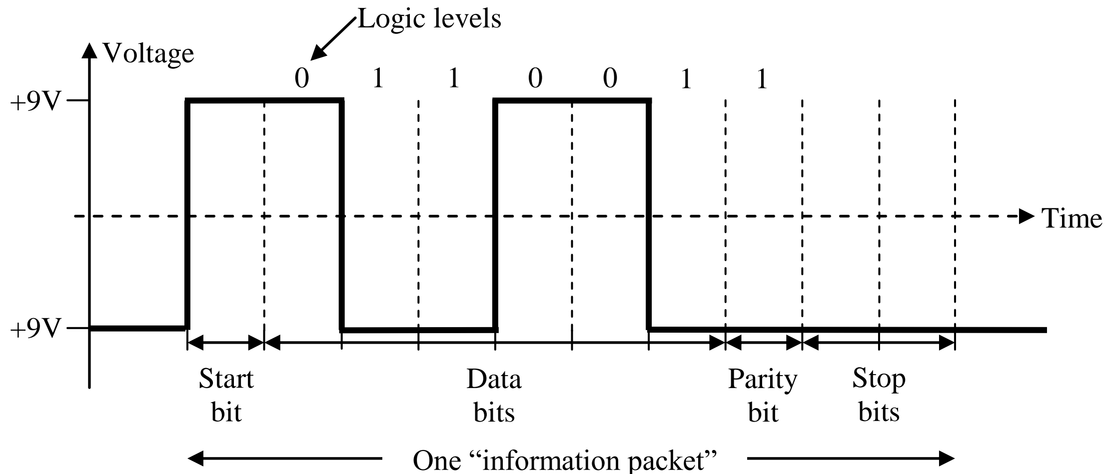
All I can say is that serial port communication has never been very high on Microsoft's to do list.

The RS-232 protocol

I developed the programmer so that communication between the programmer and the Host computer would take place at 9600 bits per second. I mention this simply to give some context for the timing events I will describe in this section. At 9600 bps, each bit has a duration of $1/9600^{th}$ of a second, or 104.167 microseconds.

RS-232 is unusual in that it has features of both unsynchronous and synchronous communication. It is unsynchronous in that an "information packet" can be sent at any random time. On the other hand, it is synchronous in that the timing of the bits inside the packet is very important. The principal virtue of this arrangement is that it requires only three physical lines: the TX (transmit) line on which the Host computer sends data out, the RX (receive) line on which the Host receives data from the programmer and the line at signal ground.

The TX and RX lines are symmetric. The Host computer's TX line is the programmer's RX line. And the programmer's TX line is the Host computer's RX line. Neither the Host computer nor the programmer "controls" the communication; the two parties are equals as far as exchanging data is concerned. The following diagram shows some parts of a typical "information packet". The vertical scale is the voltage on the line (either TX or RX) and the horizontal scale is time.



It is usual to think of an information packet as being "one byte" but that oversimplified. The packet always includes a start bit and at least one stop bit, and may include a parity bit as well. The information packet shown has a parity bit and two stop bits. Nor does the byte of data have to be eight bits; the one shown has only six bits. There are several things to note:

1. When at rest, between information packets, the voltage on the line is always low.
2. The start bit is always at the high voltage and the stop bits are always at the low voltage. It is the rising edge of the start bit which tells the receiving device that an information packet is beginning to arrive.
3. The receiving device must start timing things at the rising edge. It must rely on time to figure out when one bit ends and the next begins. Of course, the receiving end must know beforehand the speed at which the transmission will take place. It must also have agreed with the transmitting end on the number of data bits, parity bits and stop bits. At 9600 bps, the ten bits in the example will take $10 \times 104.167\mu s$, or 1.04167 milliseconds. The timer in the receiving device must be accurate enough to resolve ten bits. The accuracy of the timer is irrelevant after the instruction packet ends; nothing matters until the rising edge of the next start bit causes the process to begin all over.
4. The polarity of the data bits is inverted in RS-232. In the example, the byte being transmitted in the example is b'100110'.
5. I have assumed that the parity is "even", so that the sum of the ones in the byte, plus the parity bit itself, is an even number. Then, the parity bit must be b'1', and appears as the low voltage on the line.

You can see from the VisualBasic code set out above that this application uses eight data bits, no parity bit and one stop bit. The PIC used in the programmer (see schematics and so forth below) is run with a 20MHz ceramic resonator, so the instruction cycle time is 200 nanoseconds. The subroutine which transmits data to the Host computer has been coded so that each bit takes 521 instruction cycles, or 104.2 microseconds. That is adequate accuracy for this application. The subroutine which receives data from the Host computer has been coded to sample each bit at the mid-point of its duration.

A description of the hardware

I make reference to the schematic of the hardware, which is set out in Schedule "A" below.

The device is powered by a 12Vac wall wart. I would not normally use an alternating current wall wart but I had a nice small one sitting in the spare parts box crying out to be used. Its waveform is rectified by a small bridge rectifier B1 with a single 10μF filter capacitor C1. The filter capacitor must have a voltage high enough to handle the peak voltage, which will be approximately equal to 12Vrms $\times \sqrt{2} = 17$Vdc, less a couple of diode drops in the bridge rectifier.

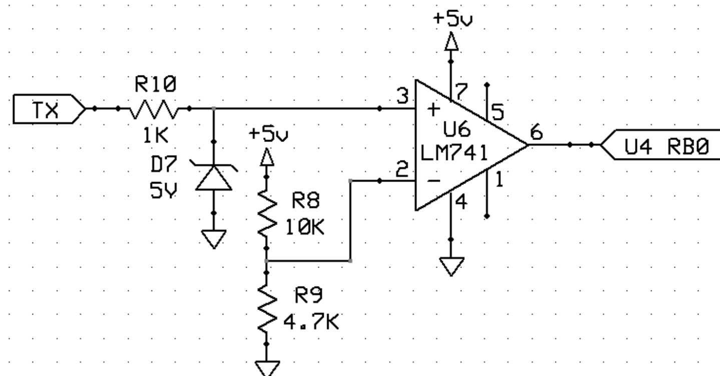
The voltage over capacitor C1 will be highly rippled, but consistent enough for regulation by a common 9V three-terminal regulator like an LM7809 (U1). The +9V output from U1 is further regulated/stored by another 10μF capacitor C2. This voltage is used in two ways.

Another three-terminal regulator, this one a 5V three-terminal regulator like an LM7805 (U2), produces +5V for use by the logic circuit. A third 10μF capacitor (C3) is a reservoir for this logic voltage. A red LED (D1) draws current directly from capacitor C3. It is a visual indicator to the user that the programmer is powered up.

The +9V supply is also used to power an LT1026 chip (U3), which is a combined voltage doubler and inverter. In the ideal no-load case, it would produce +18V on output pin 8 and -18V on output pin 4. As the chip is loaded, and current is drawn from the output pins, the output voltages drop quite quickly. Even so, the output voltages remain high enough for our purposes. 1KΩ resistors R2 and R3 drop the ±18V nominal output voltages down to ±13.6V. The 13.6V reference level is established by 13V zener diodes (D2 and D4) each of which is wired in series with a forward-biased 1N4001 diode (D3 and D5). (Aside: since very little current flows through these diodes, D3 and D5 drop slightly less than the 0.7V normally seen across a silicon p-n junction.)

Using an LT1026 to produce ±13.6V kills two birds with one stone. The +13.6V is applied to the /MCLR pin of the PIC being programmed, which will be programmed in High-Voltage Programming (HVP) mode. The dual ±13.6V levels are used as the high and low voltages for RS-232 transmissions to the Host computer.

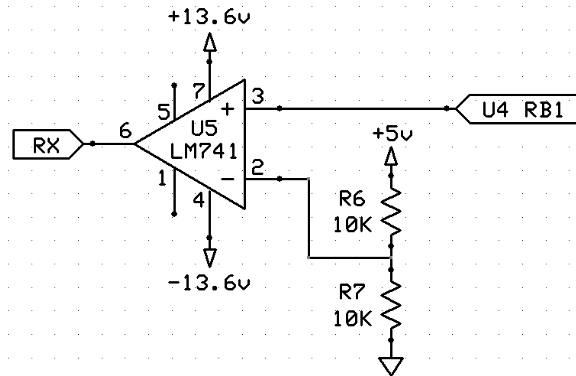
The input signal from the Host computer is processed by the following circuit. (In this paper, I will always refer to the TX and RX lines from the point of view of the Host computer. Thus, the TX line is the Host computer's outgoing transmission line, which is the programmer's input line.)



This circuit converts the ±9V RS-232 voltages on the TX line to logic levels (+5V and 0V). When the TX voltage is +9V, dropping resistor R10 and 5V zener diode D7 will cause the voltage at the opamp's positive, or non-inverting, to be +5V. When the TX voltage is -9V, the zener diode will be forward-

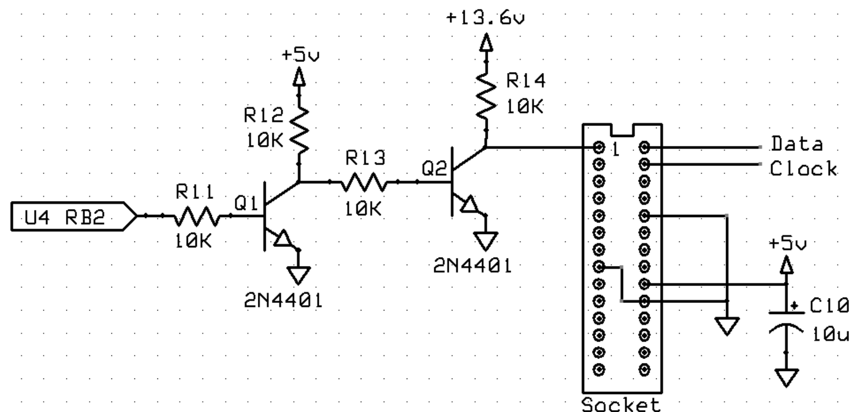
biased and will conduct like a normal diode, so the voltage at the positive terminal will be -0.7V. The reference voltage applied to the opamp's negative, or inverting, terminal is determined by the R8-R9 voltage divider. It will be a fraction 4.7 ÷ 14.7 of the logic voltage, or 1.4V. There is no feedback around this opamp, so its output voltage will be driven up or down as far as the opamp can take it. When the positive terminal voltage is greater than the negative terminal voltage, the opamp will amplify the positive difference, and the output voltage will be approximately +5V. In other words, this configuration does not invert the incoming signal. When the RS-232 voltage is high/low (+9V/-9V) the voltage delivered to the PIC's RB0 line (pin 21) will be high/low (+5V/0V). The opamp (U6) does not have to be anything special; I used a common LM741.

The circuit which generates transmissions from the programmer to the Host computer is as follows.



This circuit is similar to the previous one. It used another LM741 opamp (U5). The reference voltage applied to the opamp's negative terminal is developed by voltage divider R6-R7, so it is one half of the supply voltage, or 2.5V. Once again, the opamp is wired in a non-inverting open-loop configuration. Its output voltage will be close to the positive and minus supplies, being +13.6V and -13.6V.

The PIC's RB0 and RB1 lines (pins 21 and 22, respectively) are used for RS-232 communication with the Host computer. The PIC's RB2 line (pin 23) is used to control the /MCLR voltage applied to the PIC being programmed. The control circuit is as follows.



Two npn transistors (Q1 and Q2) scale the logic levels output on the RB2 line (+5V and 0V) to high-voltage programming levels (+13.6V and 0V) applied to the /MCLR pin of the device being programmed. I used 2N4401 transistors from the spare parts box, but any common npn transistors would be suitable. The accompanying resistors ensure that one of the transistors is in its saturation mode and the other is cutoff. Using two transistors ensures that the circuit is non-inverting: a high voltage at the RB2 pin gives a high voltage at the /MCLR pin of the PIC being programmed.

Also shown in this diagram is the 28-DIP socket, which is mounted on the lid of the programmed, and into which the PIC being programmed is inserted. The programming signals, being the Data and Clock lines, are pins 28 and 27 of the socket, respectively.

Note that pin 24 of the socket, being the RB3 line of the PIC being programmed, is hard-wired to ground. The RB3 pin is the LVP (Low Voltage Programming) pin. When a PIC is to be programmed at low voltage, the /MCLR line (pin 1) and the LVP line (pin 24) are brought up to +5V. Tying the LVP line low ensures that the PIC being programmed cannot ever enter Low Voltage Programming mode. However, that does not mean that the PIC being programmed can enter the High Voltage Programming mode! Whether or not the PIC being programmed can enter High Voltage Programming mode is determined by the setting of one of the bits in the Configuration Word(s). Historically, the factory-default setting of this configuration bit has been for High Voltage Programming. Starting not too long ago, Microchip has changed its policy, and the factory-default setting of this configuration bit is now for Low Voltage Programming. That means that a new Configuration Word(s) must be burned in before the chip can be High Voltage Programmed.

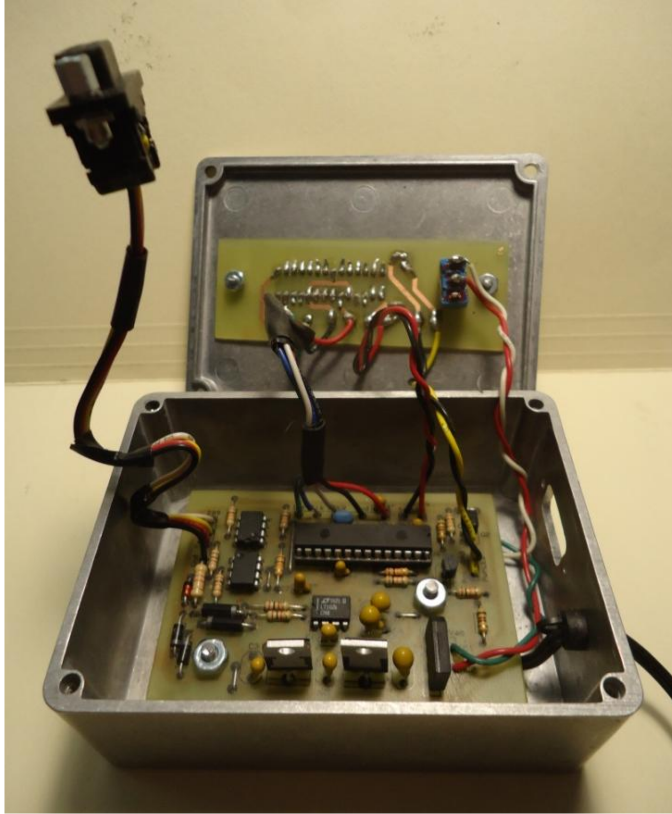
A few more notes about the hardware:

1. Resistor R5 and capacitor C9 form a poor-man's reset on power-up. The RC time constant of this pair is 10KΩ × 10µF = 100 milliseconds. The voltage at pin 1 of U4 will reach 2.5V about 50 milliseconds after power switch S1 is closed. This will give the power supply voltage 50ms to stabilize throughout the logic circuit before U4 resets.

2. A white LED (D6) is connected to the RA0 line (pin 2). I have called this the Status LED. The programmer uses this LED to notify the user about certain error conditions. The assembly listing set out below in Appendix "C" includes three error conditions and their displays.
   (a) If the PIC detects the beginning of a start bit in a transmission from the Host computer, and if the TX voltage at the mid-point of that start bit is no longer high (as it should be), then the Status LED will blink 500ms on and 1000ms off. (I have never encountered this error.)
   (b) The PIC will measure the TX voltage at the middle of a stop bit. If that voltage is not low (as it should be), then the Status LED will blink 1000ms on and 500ms off. (I have never encountered this error.)
   (c) If the PIC receives a single-byte command from the Host computer, and if the PIC does not recognize the command, then the Status LED will blink 200ms on and 200ms off, being a quick flash. (Ever since I completing debugging the PIC's program and the Host computer's program, I have not encountered this error again.)

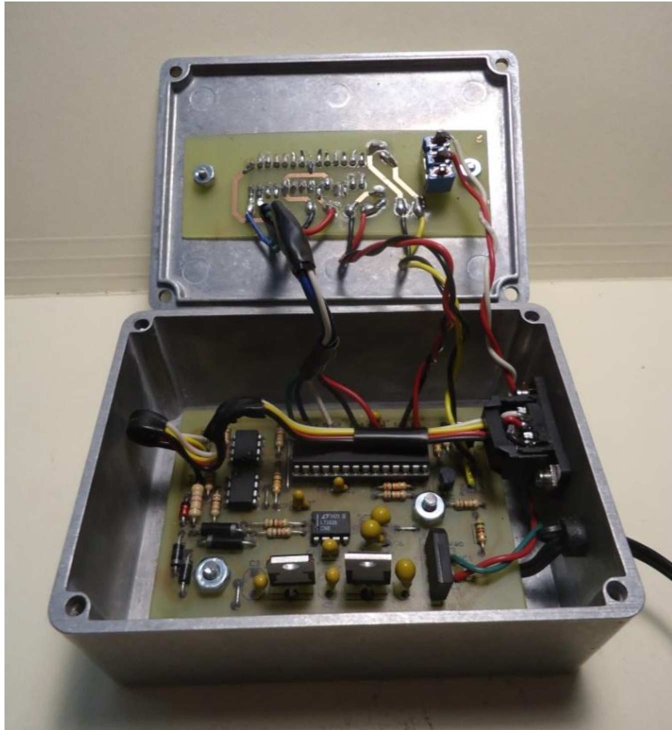
3. Despite my rant above about the latency delays suffered by the six non-participating RS-22 lines, I do make use of one of them. Note that the +13.6V supply voltage is wired directly to the CTS line of the DB9 connector. The VisualBasic program running on the Host computer uses this input during the start-up procedure of the programmer. The Host computer prompts the user to turn on the programmer. It tests `CtsHolding`, in the manner described above, until it goes True, at which time the Host computer knows that the programmer has been turned on. Since this is an operation which involves a human, the 11ms latency is not noticeable.

The following photographs show the finished programmer.



This is the inside of the project box, with the lid folded back. The printed circuit board with the programmer's PIC is bolted to the bottom of the box. The DB9 female connector is at the end of the pigtail at the top left. The cord from the wall-wart enters at the lower right.

A small printed circuit board, which I called the backing board, is bolted to the underside of the lid. The empty socket for the 28-DIP chip to be programmed is soldered to it. SO are the two LEDs. The power switch is a SPST toggle switch (although I had a surplus SPDT switch which I used here) whose throat passes through the backing board and the lid.



In this photograph, the pigtail has been folded down into the project box and the DB9 female connector bolted through a hole cut into the right-hand side.

This is the finished programmed. The empty socket is visible on the lid. The USB-to-serial port adapter cable has been wound around so the USB plug itself is lying on top of the project box. The tangle of wires underneath the box is the cord from the wall-wart, which is also shown here.

<u>How to split the programming task between the Host computer and the programmer?</u>

This is a question of design philosophy. Two computers are involved in this system: the Host computer and the PIC microprocessor inside the programmer. We have a choice about how much work each computer does.

In the usual case, the program to be programmed into a new PIC will have been developed on the Host computer using MPASM or some compiler. The output from that development process is a hex file to be burned into the new PIC.

At one extreme of the programming philosophy, the hex file would be transmitted to the programmer and the programmer would do everything. This would be a very “intelligent programmer”.

At the other extreme is the “dumb programmer”. The Host computer would control the programming process and, ideally, the PIC inside the programmer would do nothing more than relay commands from the Host computer to the PIC being programmed.

I have used the later philosophy. It has the following advantages:

1. It is much easier to change a VisualBasic program running on the Host computer than to change the program running on the programmer’s PIC.
2. As a subset of that advantage, it will be much easier to add the capability to program a different type of PIC to the Host computer than to the programmer’s PIC.
3. It is much easier for the Host computer to interact with the user to select the type of chip, etc.

Overview of the program running on the programmer's PIC

Appendix "C" below is the assembly listing for a PIC16F882. Appendix "D" is the assembly listing for a PIC16F872. Either PIC can be used as the brains of the programmer. I already pointed out that the 16F882 has two configuration words and the 16F872 has only one. But there are significant differences in the system registers. The 16F882 is a more sophisticated device than the 16F872, and more work is needed to set the system registers to get it to do what is wanted.

In operation, the PIC runs an infinite loop. It waits for the Host computer to send a one-byte command, which is stored in the user register Command. The PIC recognizes and processes the following commands:

Command	Description
0x00	Load configuration (2 bytes will follow)
0x01	Bulk erase setup1 [16F87X only]
0x02	Load data for program memory (2 bytes will follow)
0x03	Load data for EEPROM memory (2 bytes will follow)
0x04	Read data from program memory (2 bytes to be sent to Host)
0x05	Read data from EEPROM memory (2 bytes to be sent to Host)
0x06	Increment address
0x07	Bulk erase setup2 [16F87X only]
0x08	Begin erase/programming cycle
0x09	Bulk erase program memory
0x0A	End programming [16F88X only]
0x0B	Bulk erase EEPROM memory
0x17	End programming [16F87XA only]
0x18	Begin programming only cycle
0x1F	Chip erase [16F87XA only]
0x81	Request programmer to send a &H55 ping byte
0x82	Reset the PIC being programmed

All of the commands except the last two are Microchip programming commands. It is a matter of convenience that the Command byte which the Host computer transmits to the PIC can be sent by the PIC "as is" to the device being programmed.

The last two commands are not Microchip programming commands. Instead they are instructions for the programmer itself. The first of those commands, 0x81, causes the programmer to send one byte, 0x55, back to the Host. This is a ping. The last command, 0x82, causes the programmer to cycle the /MCLR voltage applied to pin 1 of the PIC being programmed. The voltage is brought down to zero for 10ms and then allowed to rise back up to +13.5V. The Data line (pin 28) and Clock line (pin 27) of the PIC being programmed are held low during this reset, so the PIC being programmed will start up in "Program and Verify" mode, in which the PIC can be programmed.

Many of the commands in the above list are not followed by data. Take command 0x18 as an example. The programmer does not know that this is Microchip's "Begin programming only cycle". The programmer simply "writes" this command to the PIC being programmed. As soon as it has "written" the command, the programmer transmits a 0x55 ping byte to the Host computer, and then returns to its infinite loop to await receipt of the next command. The Host computer, on the other hand, knows what the "Begin programming only cycle" command is supposed to do. It also knows that a delay is needed to give the command time to work inside the PIC being programmed. As soon as the Host computer receives the ping byte, it starts timing the required delay.

The only thing the programmer's PIC needs to know is how to send the least-significant six bits of the Command register to the PIC being programmed. Subroutine WriteCommandToPIC carries out that task. The subroutine cycles the Clock line (pin 27) of the PIC being programmed high and low six times, with the appropriate voltage levels exposed on the Data line (pin 28). All of the Microchip programming commands use the same bit timing. The only thing that differs between the commands is the delay after (or sometimes before) a particular command, and the Host computer is responsible for those.

Three of the commands sent by the Host computer are followed by a 14-bit word. The word is transmitted by the Host as two separate bytes. The low-order eight bits of the word are contained in the first byte transmitted, which the programmer stores in user register LowBytePIC. The high-order six bits of the word are contained in the low-order six bits of the second byte transmitted, which the programmer stores in user register HighBytePIC.

When the programmer's PIC is interpreted Commands it receives, it recognizes the three commands which are followed by incoming data. It waits until it receives the following two bytes. Then, it writes Command to the PIC being programmed, followed by the 14-bit word. Subroutine Write14BitWordToPIC carries out that task. All of Microchip commands which write 14-bit words to a PIC use the same timing, the same bit order, and dummy leading and trailing bits.

In the case of these three write commands, the programmer sends a 0x55 ping byte to the Host computer as soon as it has written the 14-bit word to the PIC being programmed, after which the programmer returns to its infinite loop. The Host computer will know whether or how long it should wait before sending the next command.

Two of the Microchip commands read 14-bit words from the PIC being programmed. When the programmer sends these two commands to the PIC being programmed, it knows that it will read 16 bits, which subroutine Read14BitWordFromPIC does. That subroutine extracts the informative 14 bits and stores them in user registers LowBytePIC (low-order eight bits) and HighBytePIC (high-order six bits; right justified in the register). Once again, the Microchip commands which read 14-bit words use the same timing, the same bit order, and dummy leading and trailing bits.

In the case of these two read commands, the programmer does not send a ping byte. The programmer transmits LowBytePIC and HighBytePIC to the Host computer. Receipt of the data bytes is sufficient evidence for the Host to know that the command has been executed.

(Aside: because the timing of command 0x82 (reset the PIC being programmed) is controlled by the programmer, it sends a 0x55 ping byte to the Host computer when the procedure is complete.)

The programmer uses the R-232 protocol to communicate with the Host computer. Subroutine ReceiveOneByteFromHost returns the single byte received in user register ByteRecvd. Subroutine SendOneByteToHost sends the byte in user register ByteToSend to the Host. Both subroutines use what is called "bit bashing" to time their way through each information packet. Each bit is 521 instruction cycles long. With a 20MHz clock, each bit has a duration of 104.2 microseconds.

The programmer calls subroutine ReceiveOneByteFromHost whenever it is expecting to receive a byte. The subroutine begins with a loop, which continually reads the TX line (RB0, or pin 21) until a rising edge makes the start of a start bit. Then the timing of instruction cycles begins.

Subroutine SendOneByteToHost does not lead off with any delay. It starts a start bit as soon as it is called. The program running on the Host computer knows when a byte will be received and will be waiting for it.

Some notes about the Microchip programming

I will not describe the Microchip programming process but will set down a few observations.

1. Make absolutely certain that you use the programming specifications for the PIC you want to program. I used the following documents:

   "PIC16F87X EEPROM Memory Programming Specification" for 16F87X devices
   "PIC16F87XA FLASH Memory Programming Specification" for 16F87XA devices
   "PIC16F88X Memory Programming Specification" for 16F88X devices

   The specifications are NOT the same. Each series has its own different bulk erase procedure. Some commands, like the EndProgramming command, use different codes.

2. I program on an address-by-address basis. Data is loaded and burned into a single address before incrementing to the next address. In some of these series, data can be loaded in blocks of four address or eight addresses and a single programming command used to burn all of those locations. I do not use multi-byte loading.

3. The LoadConfiguration command is followed by a 14-bit word. The principal purpose of the LoadConfiguration command is to change the address pointer so that it points to the start of configuration memory at address &H2000. The 14-bit word which follows is loaded (temporarily) as the data to be burned into that specific location (which happens to be the first UserID word). If the LoadConfiguration command is immediately followed by a LoadDataForProgramMemory, which itself is followed by a 14-bit word, then this new 14-bit word will be loaded (temporarily) as the data to be burned into that specific location. Whichever word was loaded last is the one that will be burned in. This is a long way of saying that the 14-bit word which follows the LoadConfiguration command can be treated as a dummy word. (Except that this 14-bit word <u>must be</u> &H3FFFF when the LoadConfiguration is used to bulk-erase **** series devices.)

4. I program using High Voltage Programming mode. The Microchip documentation is a little bit ambiguous on this point, but all steps in the HVP procedure take place with the /MCLR line (pin 1) of the PIC being programmed held at +13.5V and the Vcc line (pin 20) held at +5V. Tying pin 23 of the socket (the RB3 line) low will ensure that the PIC being programmed does not enter Low Voltage Programming mode. In addition, the LVP bit in the Configuration Word(s) needs to be disabled before High Voltage Programming is enabled. Microchip has recently changed the default setting of this bit to LVP.

Overview of the VisualBasic program running on the Host computer

Appendix "E" below is the listing of the Host computer's program. It is a VisualBssic forms application with five supporting modules.

Module HexFileProcessing.vb contains the procedures which decode an INHX8M hex file when it is opened. The characters in the hex file are copied character-by-character into a vector of integers HexFileContents(150000). The information in that file is decoded into the following vectors of integers:

    OutputBufferProgMemory(32767)
    OutputBufferEEPROMMemory(1023)
    OutputBufferUserIDWords(3)

OutputBufferDeviceIDWord
OutputBuffersConfigWords(1)
OutputBufferCalibWord

The application uses a Boolean variable OutputBuffersHaveBeenFilledFromHexFile to note that the output buffers have been filled. This flag is used by the procedure which verifies the contents of a PIC against the contents of a hex file.

Associated with each output buffer is the starting address at which the contents of these buffers will be burned into the PIC being programmed. These starting addresses are stored in the following integer variables. These variables happen to be constant since all of the PIC devices I use have these same addresses.

ProgMemStartAddress = &H0000
EEPROMStartAddress = &H2100
UserIDStartAddress = &H2000
DeviceIDStartAddress = &H2006
ConfigWordsStartAddress = &H2007
CalibWordStartAddress = &H2009

The number of relevant words in each output buffer are stored in the following integer variables. Most of these numbers differ among the series of PIC and the individual PIC in the series.

NumProgMemWords
NumEEPROMBytes
NumUserIDWords
NumDeviceIDWords
NumConfigWords
NumCalibWords

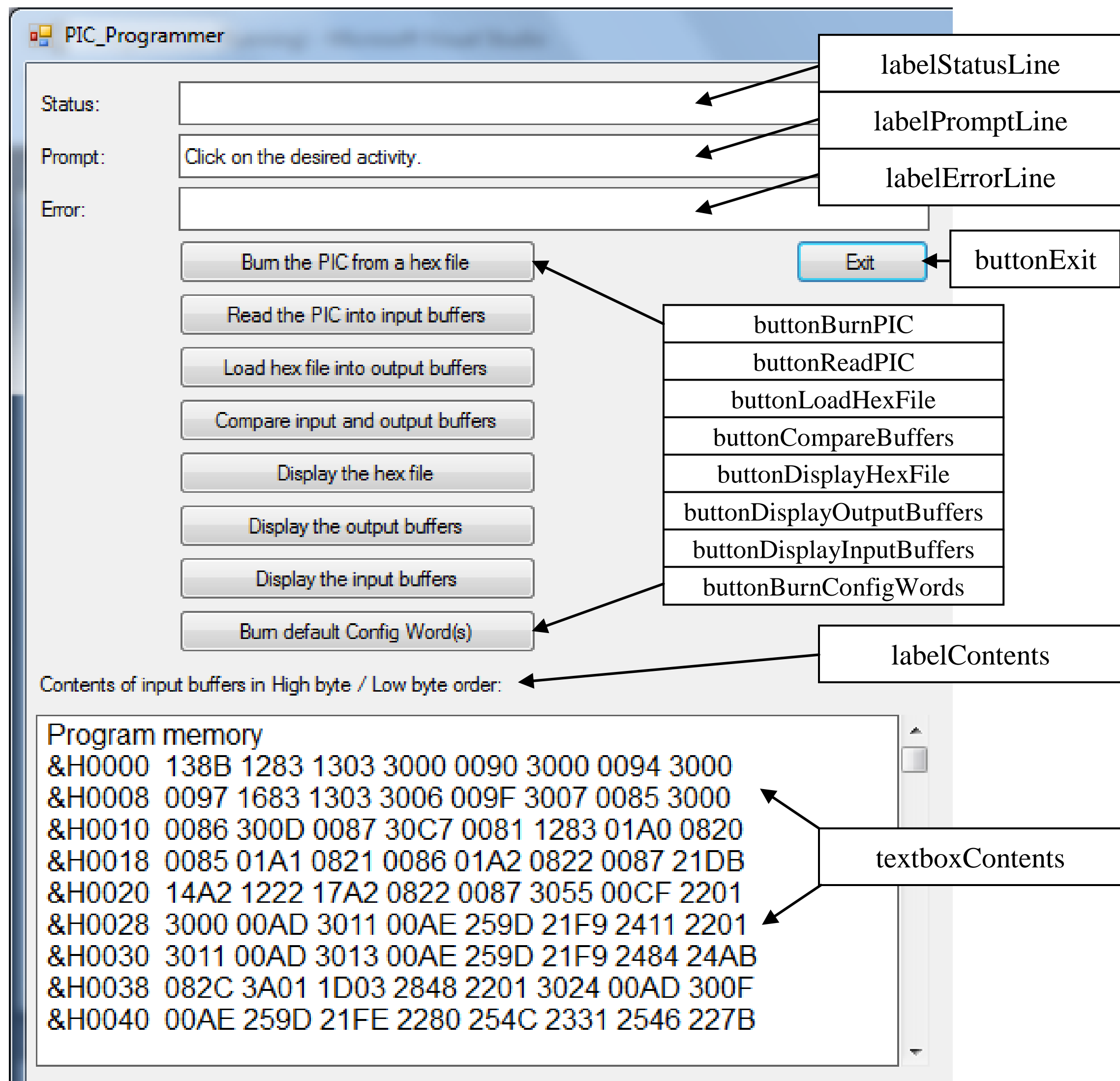
The parameters of the various PICs which can be programmed are loaded by the subroutine SetDeviceParameters(), which is located in module Variables.vb along with declaration of the global variables used by the application. When the user selects to burn a particular PIC, its parameters are loaded into the starting-address and number-of-words variables.

These is a corresponding set of input buffers:

InputBufferProgMemory(32767)
InputBufferEEPROMMemory(1023)
InputBufferUserIDWords(3)
InputBufferDeviceIDWord
InputBuffersConfigWords(1)
InputBufferCalibWord

When a PIC is read, its contents are stored in these buffers. (Any PIC can be read. It is not necessary that the PIC had been programmed already, or programmed by this device.) Another Boolean flag, InputBuffersHaveBeenFilledFromPIC, is used to note that the input buffers now contain valid data.

The following pictures is a screen-shot of the user display at one point during operation. For ease in comparing the display with the code, I have identified the names given to the controls.



Module Display.vb contains subroutines which create the display in textboxContents. One of three things can be displayed: (i) the contents of the hex file as ASCII text, (ii) the contents of the output buffers or (iii) the contents of the input buffers.

Module Procedures.vb contains the subroutines which manage RS-232 communication with the programmer. That module also contains the procedures which burn and read the PIC being programmed. Because of the differences among the different series of PIC, thee are separate subroutines for 16F87X, 16F87XA and 16F88X devices.

Arguably, the only procedure used in the VisualBasic application which is a little bit unusual is the timing function FixedTimeDelay(Int32), which pauses execution for a duration equal to the number of microseconds given in the argument. The more common function Threading.Thread.Sleep(Int32) is used in a couple of places; it pauses execution for a duration equal to the number of milliseconds given in the argument. The Sleep(Int32) function is much too slow for use in sending programming commands to the programmer.

The FixedTimeDelay(Int32) function uses what Microsoft refers to as “Ticks”, which are equal in duration to the reciprocal of the Host computer’s “Stopwatch” frequency. The code for the function is given at the end of module Procedures.vb and the parameters which govern the stopwatch are given at the end of module Variables.vb.

August 2017

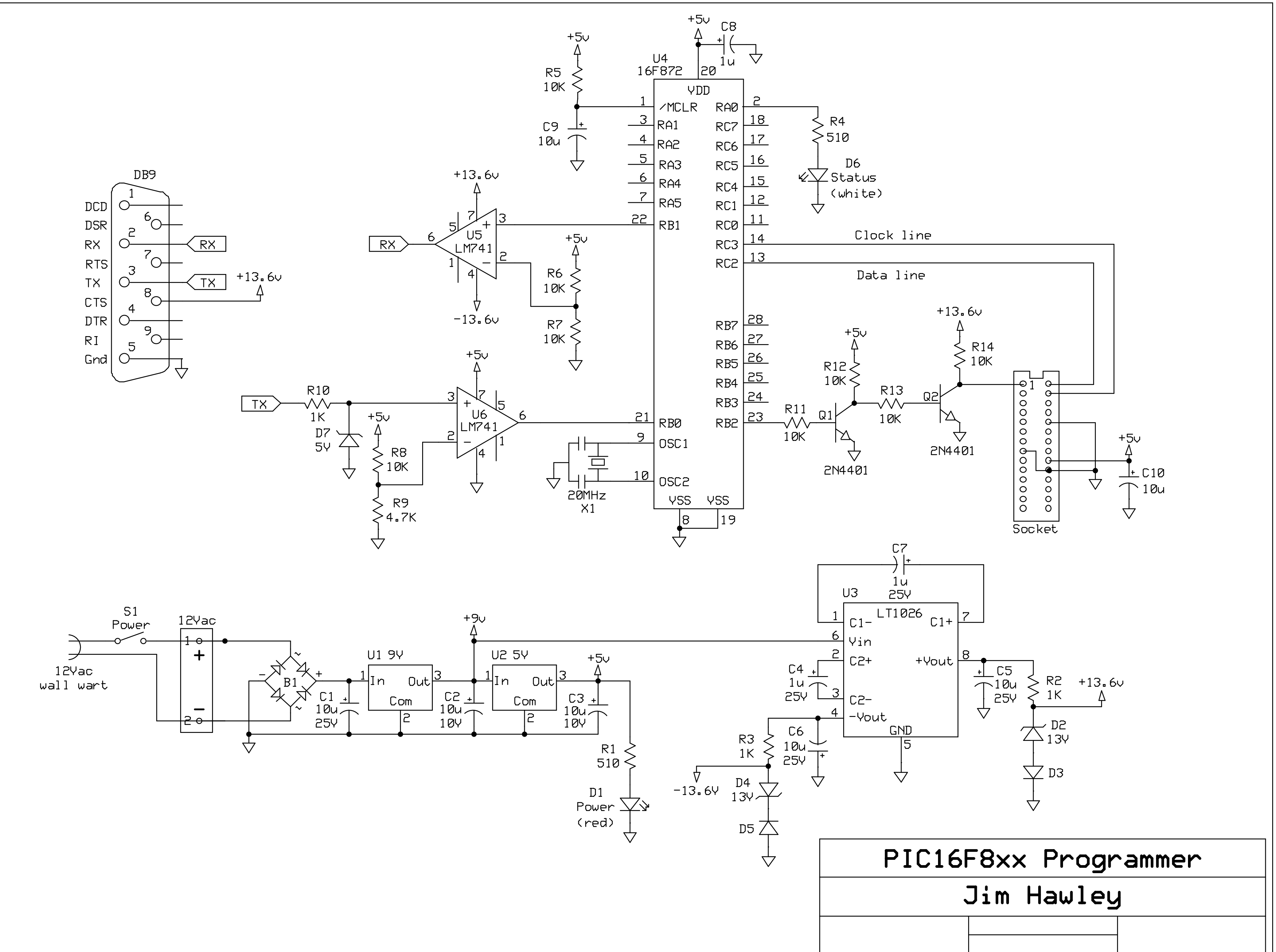
Jim Hawley

As always, I welcome an e-mail describing errors and omissions.

List of Appendices

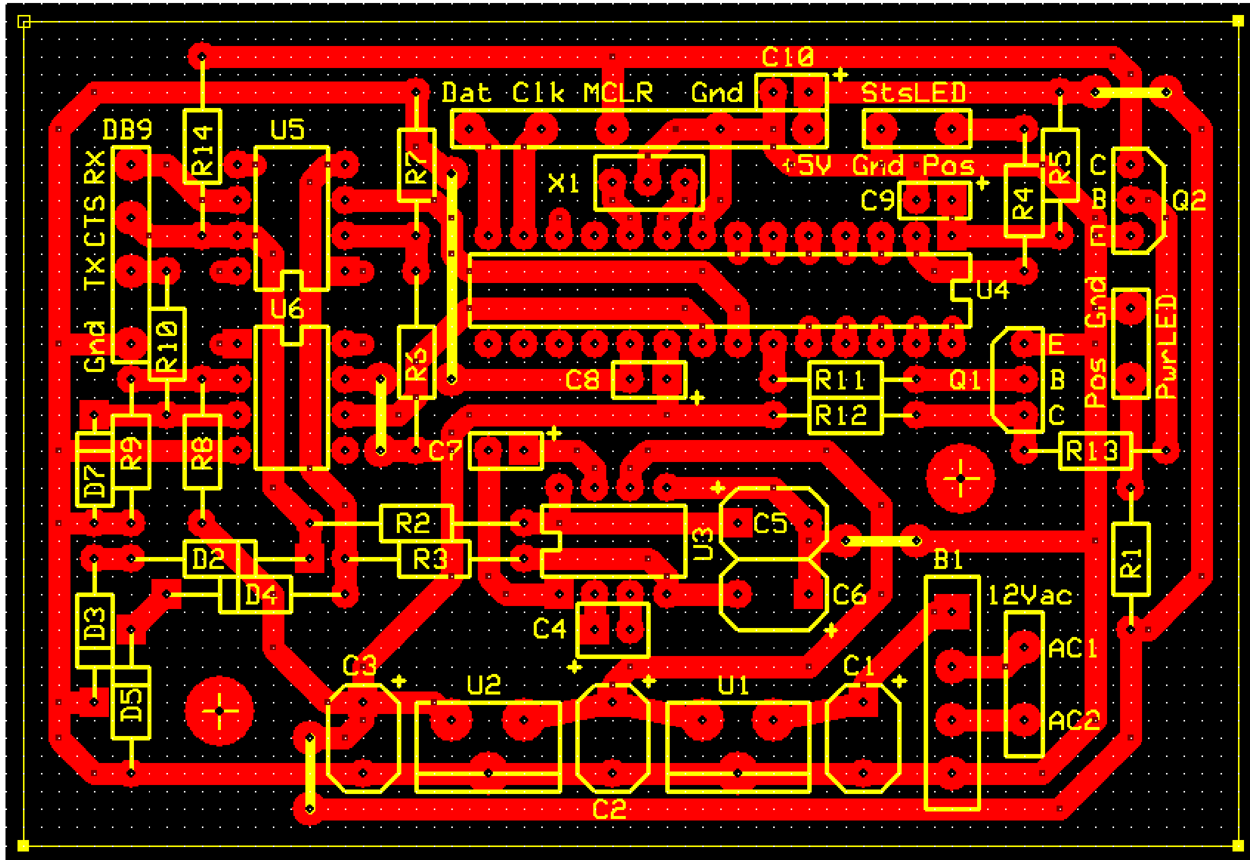
“A” – Schematic diagram
“B” – Printed circuit board layout (drawn using Express123)
“C” – Assembly listing for programmer – if PIC16F882 is used in the programmer
“D” – Assembly listing for programmer – if PIC16F872 is used in the programmer
“E1” – VisualBasic program for the Host computer – Main form
“E2” – VisualBasic program for Host computer – Module HexFileProcessing.vb
“E3” – VisualBasic program for Host computer – Module Variables.vb
“E4” – VisualBasic program for Host computer – Module Procedures.vb
“E5” – VisualBasic program for Host computer – Module Display.vb
“E6” – VisualBasic program for Host computer – Module Debugging.vb

Appendix “A”

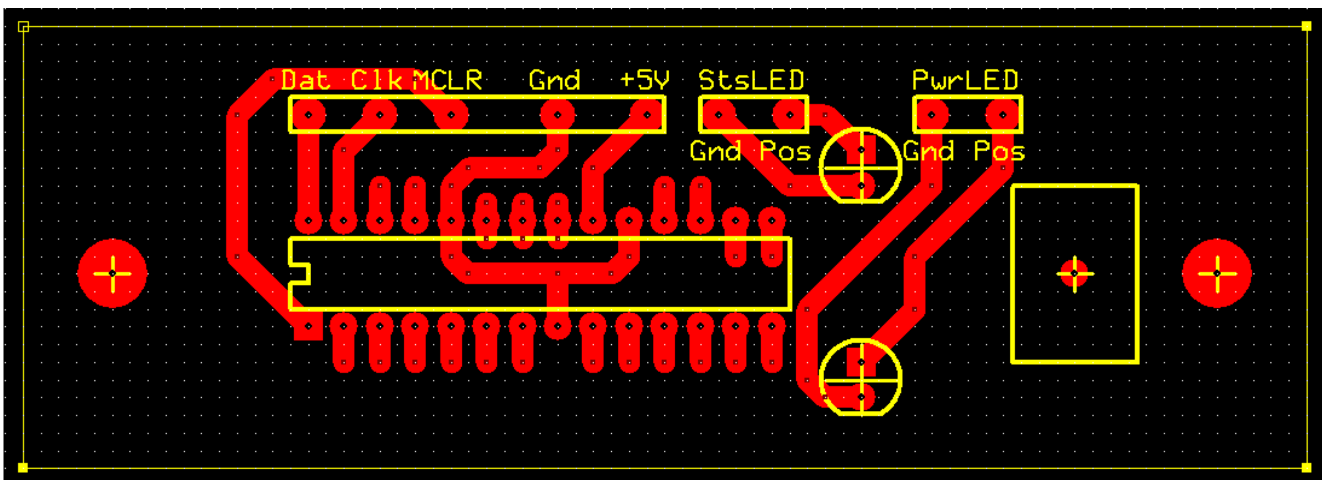


Appendix "B"

Main PCB – 3.4" × 2.3"



Backing board for bottom of lid – 3.6" × 1.25"



Appendix "C"

Assembly listing for programmer - PIC16F882

```
; Program for PIC Programmer, using 16F882 device
;
; This program does not use interrupts.  The subroutines are grouped into
; the following blocks:
; A.  Routines to communicate with the Host PC
; B.  Routines to read and write to the PIC being programmed
; C.  Controls for the Clock, Data and Reset lines of the PIC being programmed
; D.  Timing and delays
; E.  Routines used for debugging purposes only
;
; The program detects three error conditions when receiving data from the Host.
; It alerts the user by flashing the Status LED in the following ways:
; 1.  Start bit is high - 500ms on followed by 1000ms off
; 2.  Stop bit is low - 1000ms on followed by 500ms off
; 3.  Unknown command - 200ms on followed by 200ms off
;
; For maximum versatility of the application, I have decided to have the Host computer
; do as much as possible of the timing and flow control, and to have the programmer do
; as little as possible.  The programmer does not need to have any knowledge about
; the PIC it is programming.  I have made this decision because it is easier to make
; all changes (to accommodate new PICs, for example) in the Host's VisualBasic program
; than to make some changes there and some in the programmer.  Therefore, single-byte
; commands sent by the Host to the programmer come in two types:
; 1.  Microchip flash programming commands, which are written to the PIC being
;     programmed as soon as they are received.  All Microchip flash programming
;     command have a zero most-significant bit.  The following Microchip commands are
;     recognized.  The data bytes which follow these commands, if any, are described
;     in round parentheses.  If the command applies only to one series of 16FXXX
;     devices, the series name is described in square brackets.  Further details are
;     given in the blocks of code below which process these commands.
;      0x00            Load configuration (2 bytes will follow)
;      0x01            Bulk erase setup1 [16F87X only]
;      0x02            Load data for program memory (2 bytes will follow)
;      0x03            Load data for EEPROM memory (2 bytes will follow)
;      0x04            Read data from program memory (2 bytes to be sent to Host)
;      0x05            Read data from EEPROM memory (2 bytes to be sent to Host)
;      0x06            Increment address
;      0x07            Bulk erase setup2 [16F87X only]
;      0x08            Begin erase/programming cycle
;      0x09            Bulk erase program memory
;      0x0A            End programming [16F88X only]
;      0x0B            Bulk erase EEPROM memory
;      0x17            End programming [16F87XA only]
;      0x18            Begin programming only cycle
;      0x1F            Chip erase [16F87XA only]
; 2.  Other commands, like an MCLR reset of the PIC being programmed, which are not
;     programming commands.  These commands are identified by a most-significant bit
;     which is high.
;      0x81            Request programmer to send a &H55 ping byte
;      0x82            Reset the PIC being programmed
;
; RS-232 communication with the Host computer
; 1.  Communication with the Host is done on a byte-by-byte basis, where the bit
;     rate is fixed at 9600 bps, corresponding to 104.2 microseconds per bit.
; 2.  This PIC does not use interrupt-driven send and receive buffers.  This imposes
;     some constraints.  For example, this PIC must be executing the subroutine
;     ReceiveByteFromHost() at the instant when the Host actually transmits a byte,
;     or parts of the byte will be lost.  The Host should never send a byte to this
;     PIC unless it is confident this PIC is listening.  Transmission from this PIC
```

```
;     to the Host is less sensitive since the Host uses a read buffer.
; 3.  When sending a byte to the Host, this PIC stops monitoring the transmission
;     after it starts sending the stop bit.  That is because the low voltage during
;     the stop bit is indistinguishable from the low voltage on the RX line which
;     prevails between bytes.  The advantage of not waiting out the duration of the
;     stop bit is that it gives this PIC 100us or so to do something else.  The
;     disadvantage of not waiting out the stop bit is that this PIC must take care not
;     send a second byte to the Host without waiting out the equivalent time.
; 4.  When receiving a byte from the Host, this PIC stops monitoring the transmission
;     after it confirms the stop bit.  Since is confirms the stop bit midway through
;     it scheduled duration, the subroutine ReceiveOneByteFromHost() returns about
;     one-half bit, or 52us, before the end of the stop bit.  The advantage of not
;     waiting out the stop bit is that it gives this PIC 50us or so to do something
;     else.  The disadvantage of not waiting out the stop bit is that this PIC must
;     take care not to send a byte to the Host without waiting out the equivalent
;     time.
; 5.  Paragraphs 3 and 4 mean that there will not be a problem when this PIC sends a
;     byte to the Host and then immediately sits back and prepares to receive a byte
;     from the Host.  The wait loop at the start of ReceiveOneByteFromHost() will
;     run until the Host gets around to responding.  This conclusion applies so long
;     as "immediately" in the previous sentence does not exceed 104us or so,
;     corresponding to the duration of one bit.
; 6.  The problem in paragraphs 3 and 4 arises only when this PIC receives a byte from
;     the Host, does some processing, and then needs to send something back to the
;     Host.  The reply could be data or it could simply be a ping to tell the Host
;     that it has completed an operation.  Either way, this PIC cannot start sending
;     its reply before a one-half bit time, being 52us or so, has elapsed.
; 7.  To help manage communications, this PIC sends a ping byte &H55 to the Host after
;     receiving every command and after completing whatever work is required to deal
;     with the command received.
;
; Configuration Words for 16F882
;         b<13>=1         Disable in-circuit debugger
;         b<12>=0         Disable Low-Voltage Programming
;         b<11>=0         Disable fail-safe clock monitor
;         b<10>=0         Disable internal/external switchover
;         b<9-8>=00       Disable brown-out reset
;         b<7>=1          Turn OFF EEPROM memory protection
;         b<6>=1          Turn OFF program memory protection
;         b<5>=1          Set standard /MCLR operation
;         b<4>=1          Disable power-up timer
;         b<3>=0          Disable watch-dog timer
;         b<2-0>=010      Set HS oscillator gain
  #include "p16F882.inc"
  processor     16F882
  __CONFIG _CONFIG1, 0x20F2 ; b'xx10 0000 1111 0010'
  __CONFIG _CONFIG2, 0x3FFF
;
; Crystal frequency is 20MHz, so the instruction cycle time is 200ns
;
; **************************************************************************************
; Variable definitions - PIC 16F882 system registers
; **************************************************************************************
;
; Registers in Bank0
TMR0            equ   0x01        ; Timer0 count register
PCL             equ   0x02        ; Program counter, low byte
status          equ   0x03        ; Status register
carry           equ         0x00  ; carry from MSB occurred
zero            equ         0x02  ; result of operation is zero
page0           equ         0x05  ; register bank selector low bit
page1           equ         0x06  ; register bank selector high bit
portA           equ   0x05
portB           equ   0x06
```

```
portC           equu   0x07
PCLATH          equ   0x0A          ; Program counter, high byte
INTCON          equ   0x0B          ; Interrupt control register
T1CON           equ   0x10          ; Timer1 control register
SSPCON          equ   0x14          ; Synch serial port control register 1
CCP1CON         equ   0x17          ; Capture/compare/PWM control register 1
RCSTA           equ   0x18          ; Receive status and control register
CCP2CON         equ   0x1D          ; Capture/compare/PWM control register 2
ADCON0          equ   0x1F          ; Analogue-to-digital control register 0
;
; Registers in Bank1
OPTION_REG      equ   0x81          ; Option register
TRISA           equ   0x85          ; portA pin I/O direction
TRISB           equ   0x86          ; portB pin I/O direction
TRISC           equ   0x87          ; portC pin I/O direction
PCON            equ   0x8E          ; Power control register
WPUB            equ   0x95          ; Weak pull-up portB register
IOCB            equ   0x96          ; Interrupt-on-change portB
PSTRCON         equ   0x9D          ; Pulse steering control
;
; Registers in Bank2
CM1CON0         equ   0x107         ; Comparator C1 control register 0
CM2CON0         equ   0x108         ; Comparator C2 control register 0
CM2CON1         equ   0x109         ; Comparator C2 control register 1
;
; Registers in Bank3
ANSEL           equ   0x188         ; Analogue select low register
ANSELH          equ   0x189         ; Analogue select high register
;
f               equ         0x01    ; f and w identify destination register
w               equ         0x00
;
; ***********************************************************************************
; Variable definitions - User RAM, accessible only in bank0
; ***********************************************************************************
;
; I/O ports
portAmirror     equ   0x20
StatusLED       equ         0x00    ; Output - To Status LED
ncRA1           equ         0x01    ; Output - not connected
ncRA2           equ         0x02    ; Output - not connected
ncRA3           equ         0x03    ; Output - not connected
ncRA4           equ         0x04    ; Output - not connected
ncRA5           equ         0x05    ; Output - not connected
;
portBmirror     equ   0x21
TX              equ         0x00    ; Input - To TX pin of DB9
RX              equ         0x01    ; Output - To RX pin of DB9
MCLRLine        equ         0x02    ; Output - To MCLR of PIC being programmed
ncRB3           equ         0x03    ; Output - not connected
ncRB4           equ         0x04    ; Output - not connected
ncRB5           equ         0x05    ; Output - not connected
ncRB6           equ         0x06    ; Output - not connected
ncRB7           equ         0x07    ; Output - not connected
;
portCmirror     equ   0x22
ncRC0           equ         0x00    ; Output - not connected
ncRC1           equ         0x01    ; Output - not connected
DataLine        equ         0x02    ; IO - To pin RB8 of PIC being programmed
ClockLine       equ         0x03    ; Output - To pin RB7 of PIC being programmed
ncRC4           equ         0x04    ; Output - not connected
ncRC5           equ         0x05    ; Output - not connected
ncRC6           equ         0x06    ; Output - not connected
ncRC7           equ         0x07    ; Output - not connected
```

```
;
; Single bytes received from and to be sent to the Host
ByteRecvd       equ   0x23        ; Any byte received from Host
ByteToSend      equ   0x24        ; Any byte ready to send to Host
;
; Registers holding bytes to send or receive from the PIC being programmed
Command         equ   0x25
LowBytePIC      equ   0x26
HighBytePIC     equ   0x27
;
; temporary storage
tempMP          equ   0x40        ; Used in MainProgram
tempRWFP        equ   0x41        ; Used in subroutine Read14BitWordFromPIC()
tempWWTP        equ   0x42        ; Used in subroutine Write14BitWordToPIC()
tempWCTP        equ   0x43        ; Used in subroutine WriteCommandToPIC()
tempROBFH       equ   0x44        ; Used in subroutine ReceiveOneByteFromHost()
tempRDL         equ   0x45        ; Used in subroutine ReadDataLine()
tempDus         equ   0x46        ; Used in subroutines Delay***us()
ByteToDisplay   equ   0x47        ; Used in subroutine DisplayAByte()
tempDAB1        equ   0x48        ; Used in subroutine DisplayAByte()
tempDAB2        equ   0x49        ; Used in subroutine DisplayAByte()
tempCT1         equ   0x4A        ; Used in subroutines CommTest*()
tempCT2         equ   0x4B        ; Used in subroutines CommTest*()
;
; ************************************************************************************
; Hard start
; ************************************************************************************
    org   0x0000
    goto  InitializeSystemRegisters
;
; ************************************************************************************
; Initialization
; ************************************************************************************
    org   0x0010
InitializeSystemRegisters
    ;
    ; Select register bank 0
    bcf   status,page0
    bcf   status,page1
    ; INTCON=0 disables all interrupt activity (affects portB)
    clrf  INTCON
    ; T1CON=0 disables Timer1 (affects portC)
    clrf  T1CON
    ; SSPCON<5>=0 disables synchronous serial port (affects portA and portC)
    clrf  SSPCON
    ; CCP1CON=0 disables Enhanced C/C/P module (affects portB and portC)
    clrf  CCP1CON
    ; RCSTA=0 disables the serial port (affects portC)
    clrf  RCSTA
    ; CCP2CON=0 disables Capture/Compare/PWM module (affects portC)
    clrf  CCP2CON
    ; ADCON0=0 disables Analogue-to-digital converter (affects portA)
    clrf  ADCON0
    ;
    ; Select register bank 1
    bsf   status,page0
    ; Configure OPTION_REG (affects portB)
    ; <7>=1 disable PortB pull-up resistors
    ; <6>=1 RB0 interrupt on rising edge
    ; <5>=0 internal clock drives Timer0
    ; <4>=0 increment Timer0 on low-to-high
    ; <3>=0 assign prescalar to Timer0
    ; <2-0>=111 set Timer0 prescalar 256:1
    movlw 0xC7
```

```
    movwf OPTION_REG
    ; The RA0 pin controls the statusLED and is always configured for output.
    movlw 0x00              ; RA0-RA5=Output
    movwf TRISA
    ; We will power up with the TX line (RB0) configured for input.  We can do
    ; this because the Host computer will have opened the serial port before
    ; this PIC is powered up.  Therefore, there will be already be a stable
    ; voltage on the TX line when execution reaches this point.  The RX line
    ; (RB1) and the MCLR line to the PIC being programmed (RB2) will be
    ; configured for output.  These I/O directions are never changed.
    movlw 0x01              ; RB0(TX)=Input, other pins output
    movwf TRISB
    ; To avoid any possible voltage conflicts, we will power up with the
    ; DataLine (RC2) and ClockLine (RC3) leading to the PIC being programmed
    ; both configured for input.  These two lines will be reconfigured for
    ; output, and their voltages pulled low, immediately before the PIC being
    ; programmed is reset.  Putting the PIC being programmed into programming
    ; mode requires that the DataLine and ClockLine be low at the time the MCLR
    ; voltage is raised from zero to 13V.  This is done in subroutine
    ; ResetThePIC().
    movlw 0x0C              ; RC2-RC3=Input, other pins output
    movwf TRISC
    ; PCON<ULPWUE>=0 disables ultra low-power wake-up current on RA0
    ; PCON<SBOREN>=0 disbales brown-out reset
    ; (affects portA)
    bcf   PCON,5
    bcf   PCON,4
    ; WPUB=0 disables weak pull-up resistors (affects portB)
    clrf  WPUB
    ; IOCB=0 disables Interrupt-on-change (affects portB)
    clrf  IOCB
    ; PSTRCON=0 zeroes the pulse steering pin assignments (affects portC)
    clrf  PSTRCON
    ;
    ; Select register bank 2
    bcf   status,page0
    bsf   status,page1
    ; Disable Comparator module 1 (affects pins on portA)
    clrf  CM1CON0
    ; Disable Comparator module 2 (affects pins on portA)
    clrf  CM2CON0
    ; Disable Comparator module 2 (affects pins on portA and portB)
    clrf  CM2CON1
    ;
    ; Select register bank 3
    bsf   status,page0
    ; Disable analogue-to-digital on A/D channels 0-7 (affects portA)
    clrf  ANSEL
    ; Disable analogue-to-digital on A/D channels 8-13 (affects portB)
    clrf  ANSELH
    ;
    ; Select register bank 0 for main program
    bcf   status,page0
    bcf   status,page1
InitializePortContents
    clrf  portAmirror
    bsf   portAmirror,StatusLED   ; turn on the StatusLED
    movf  portAmirror,w
    movwf portA
    clrf  portBmirror             ; assert the RX and MCLR lines low
    movf  portBmirror,w
    movwf portB
    clrf  portCmirror             ; assert the non-connected lines of portC
    movf  portCmirror,w
```

```
    movwf portC
VisualConfirmationOfStartup
    call  Delay500ms              ; hold StatusLED on for one-half second ...
    bcf   portAmirror,StatusLED   ; ... so the user can verify startup
    movf  portAmirror,w
    movwf portA
;
; ***************************************************************************************
; Synchronize with the Host by exchanging pings.
; ***************************************************************************************
;
AwaitPingFromHost
    call  ReceiveOneByteFromHost
    movf  ByteRecvd,w
    movwf tempMP
    movlw 0x81                    ; Host's request for ping
    xorwf tempMP,w
    btfss status,zero
    goto  AwaitPingFromHost
    movlw 0x55                    ; programmer's ping byte
    movwf ByteToSend
    call  SendOneByteToHost
;
; ***************************************************************************************
; Main loop to await and decode instructions from the Host
; ***************************************************************************************
;
AwaitCommandFromHost
    call  ReceiveOneByteFromHost
    movf  ByteRecvd,w
    movwf Command
    andlw 0x80                    ; 0x80 = b'10000000'
    btfsc status,zero
    goto  Test_LoadConfiguration  ; process Microchip commands
    goto  Test_RequestPing        ; process non-Microchip commands
;
; Decoding of Microchip commands
;
Test_LoadConfiguration
    movlw 0x00                    ; code for "Load configuration"
    xorwf Command,w
    btfss status,zero
    goto  Test_BulkEraseSetup1
    ; The "Load configuration" command re-locates the address pointer in the PIC
    ; being programmed to the start of configuration memory.  A 14-bit word
    ; follows this command.  This word can be the contents of address 0x2000.
    ; Alternatively, this word can be a dummy word, just so long as the correct
    ; word is loaded using a LoadDataIntoProgramMemory before burning.  Note,
    ; however, that the bulk erase procedure for the 16F87X series uses the
    ; LoadConfiguration command with a 14-bit payload of 0x3FFF.  Immediately
    ; after writing the command and 14-bit word to the PIC being programmed, this
    ; procedure sends a 0x55 ping byte to the Host -- it does not wait.
    call  ReceiveOneByteFromHost
    movf  ByteRecvd,w
    movwf LowBytePIC
    call  ReceiveOneByteFromHost
    movf  ByteRecvd,w
    movwf HighBytePIC
    call  WriteCommandToPIC       ; write command to PIC
    call  Write14BitWordToPIC     ; write 14-bit word to the PIC
    call  SendPingToHost
    goto  AwaitCommandFromHost
;
```

```
Test_BulkEraseSetup1
     movlw 0x01                    ; code for "Bulk erase setup1"
     xorwf Command,w
     btfss status,zero
     goto  Test_LoadDataForProgramMemory
     ; The "BulkEraseSetup1" command is part of the master erase procedure for
     ; 16F87X chips.  Immediately after writing the command to the PIC being
     ; programmed, this procedure sends a 0x55 ping byte to the Host -- it does
     ; not wait.
     call  WriteCommandToPIC
     call  SendPingToHost
     goto  AwaitCommandFromHost
;
Test_LoadDataForProgramMemory
     movlw 0x02                    ; code for "Load data for program memory"
     xorwf Command,w
     btfss status,zero
     goto  Test_LoadDataForEEPROMMemory
     ; The "Load data for program memory" command loads a 14-bit word into the PIC
     ; being programmed, at the current address pointer.  The 14-bit word will be
     ; sent by the Host in the next two bytes, which will be stored in the register
     ; pair HighBytePIC, LowBytePIC.  Immediately after writing the word to the
     ; PIC being programmed, this procedure sends a 0x55 ping byte to the Host --
     ; it does not wait.
     call  ReceiveOneByteFromHost
     movf  ByteRecvd,w
     movwf LowBytePIC
     call  ReceiveOneByteFromHost
     movf  ByteRecvd,w
     movwf HighBytePIC
     call  WriteCommandToPIC
     call  Write14BitWordToPIC
     call  SendPingToHost
     goto  AwaitCommandFromHost
;
Test_LoadDataForEEPROMMemory
     movlw 0x03                    ; code for "Load data for EEPROM memory"
     xorwf Command,w
     btfss status,zero
     goto  Test_ReadDataFromProgramMemory
     ; The "Load data for EEPROM memory" command loads an 8-bit byte into the PIC
     ; being programmed, at the current address pointer.  Even though only 8 bits
     ; are relevant, a whole 14-bit word must be loaded into the PIC being
     ; programmed.  The Host will follow up the command by sending two bytes.  The
     ; relevant 8 bits are the first byte.  The second byte contains 6 irrelevant
     ; bits, which must be sent along to the chip being programmed.  Immediately
     ; after writing the command and the 14-bit word to the PIC being programmed,
     ; this procedure sends a 0x55 ping byte to the Host -- it does not wait.
     call  ReceiveOneByteFromHost
     movf  ByteRecvd,w
     movwf LowBytePIC
     call  ReceiveOneByteFromHost
     movf  ByteRecvd,w
     movwf HighBytePIC
     call  WriteCommandToPIC
     call  Write14BitWordToPIC
     call  SendPingToHost
     goto  AwaitCommandFromHost
;
Test_ReadDataFromProgramMemory
     movlw 0x04                    ; code for "Read data from program memory"
     xorwf Command,w
     btfss status,zero
     goto  Test_ReadDataFromEEPROMMemory
```

```
     ; The "Read data from program memory" command reads a 14-bit word from the PIC
     ; being programmed, at the current address pointer.  The 14-bit word will be
     ; stored in the register pair HighBytePIC, LowBytePIC.  The two bytes will be
     ; sent to the Host as soon as they are available.  This procedure does not
     ; send a 0x55 ping byte since the two bytes of data sent to the Host are
     ; sufficient evidence of completion.
     call  WriteCommandToPIC
     call  Read14BitWordFromPIC
     movf  LowBytePIC,w
     movwf ByteToSend
     call  SendOneByteToHost
     movf  HighBytePIC,w
     movwf ByteToSend
     call  SendOneByteToHost
     goto  AwaitCommandFromHost
;
Test_ReadDataFromEEPROMMemory
     movlw 0x05                    ; code for "Read data from EEPROM memory"
     xorwf Command,w
     btfss status,zero
     goto  Test_IncrementAddress
     ; The "Read data from EEPROM memory" command reads an 8-bit byte from the PIC
     ; being programmed, as the current address pointer.  The command actually
     ; extracts a 14-bit word from the PIC, but the high-order six bits are
     ; irrelevant.  This routine sends both bytes back to the Host including the
     ; 6 irrelevant bits in the high-order bytes.  This procedure does not send a
     ; 0x55 ping byte since the data bytes sent to the Host are sufficient evidence
     ; of completion.
     call  WriteCommandToPIC
     call  Read14BitWordFromPIC
     movf  LowBytePIC,w
     movwf ByteToSend
     call  SendOneByteToHost
     movf  HighBytePIC,w
     movwf ByteToSend
     call  SendOneByteToHost
     goto  AwaitCommandFromHost
;
Test_IncrementAddress
     movlw 0x06                    ; code for "Increment Address"
     xorwf Command,w
     btfss status,zero
     goto  Test_BulkEraseSetup2
     ; The "Increment Address" command increments the address pointer in whatever
     ; block of memory it currently resides.  After writing the command to the PIC
     ; being programmed, this procedure sends a 0x55 ping byte to the Host -- it
     ; does not wait.
     call  WriteCommandToPIC
     call  SendPingToHost
     goto  AwaitCommandFromHost
;
Test_BulkEraseSetup2
     movlw 0x07                    ; code for "Bulk erase setup2"
     xorwf Command,w
     btfss status,zero
     goto  Test_BeginEraseProgrammingCycle
     ; The "BulkEraseSetup2" is part of the master erase procedure for 16F87X
     ; devices.  Immediately after writing the command to the PIC being
     ; programmed, this procedure sends a 0x55 ping byte to the Host -- it does
     ; not wait.
     call  WriteCommandToPIC
     call  SendPingToHost
     goto  AwaitCommandFromHost
;
```

```
Test_BeginEraseProgrammingCycle
     movlw 0x08                    ; code for "Begin erase/programming cycle"
     xorwf Command,w
     btfss status,zero
     goto  Test_BulkEraseProgramMemory
     call  WriteCommandToPIC
     call  SendPingToHost
     goto  AwaitCommandFromHost
;
Test_BulkEraseProgramMemory
     movlw 0x09                    ; code for "Bulk erase program memory"
     xorwf Command,w
     btfss status,zero
     goto  Test_EndProgramming1
     ; The "Bulk erase program memory" command erases the program memory in non-
     ; code-protected chips.  After writing the command to the PIC being
     ; programmed, this procedure sends a 0x55 ping byte to the Host -- it does not
     ; wait.
     call  WriteCommandToPIC
     call  SendPingToHost
     goto  AwaitCommandFromHost
;
Test_EndProgramming1
     movlw 0x0A                    ; code for "End programming"
     xorwf Command,w
     btfss status,zero
     goto  Test_BulkEraseEEPROMMemory
     ; The "End programming" command terminates the burning process.  Only some
     ; PICs use this code.  After writing the command to the PIC being programmed,
     ; this procedure sends a 0x55 ping byte to the Host -- it does not wait.
     call  WriteCommandToPIC
     call  SendPingToHost
     goto  AwaitCommandFromHost
;
Test_BulkEraseEEPROMMemory
     movlw 0x0B                    ; code for "Bulk erase EEPROM memory"
     xorwf Command,w
     btfss status,zero
     goto  Test_EndProgramming2
     ; The "Bulk erase EEPROM memory" command erases the EEPROM memory in non-code-
     ; protected chips.  After writing the command to the PIC being programmed,
     ; this procedure sends a 0x55 ping byte to the Host -- it does not wait.
     call  WriteCommandToPIC
     call  SendPingToHost
     goto  AwaitCommandFromHost
;
Test_EndProgramming2
     movlw 0x17                    ; code for "End programming"
     xorwf Command,w
     btfss status,zero
     goto  Test_BeginProgrammingOnlyCycle
     ; The "End programming" command terminates the burning process.  Only some
     ; PICs use this code.  After writing the command to the PIC being programmed,
     ; this procedure sends a 0x55 ping byte to the Host -- it does not wait.
     call  WriteCommandToPIC
     call  SendPingToHost
     goto  AwaitCommandFromHost
;
Test_BeginProgrammingOnlyCycle
     movlw 0x18                    ; code for "Begin programming only cycle"
     xorwf Command,w
     btfss status,zero
     goto  Test_ChipErase
     ; The "Begin programming only cycle" command begins to burn the new contents
```

```
     ; at the address of the current pointer.  After writing the command to the PIC
     ; being programmed, this procedure sends a 0x55 ping byte to the Host -- it
     ; does not wait.
     call  WriteCommandToPIC
     call  SendPingToHost
     goto  AwaitCommandFromHost
;
Test_ChipErase
     movlw 0x1F                    ; code for "Chip erase"
     xorwf Command,w
     btfss status,zero
     goto  CommandError
     ; The "Chip erase" command erase all memory except for the UserID Words.
     ; After writing the to the PIC being programmed, this procedure sends a 0x55
     ; ping byte to the Host -- it does not wait.
     call  WriteCommandToPIC
     call  SendPingToHost
     goto  AwaitCommandFromHost
;
; Decoding of non-Microchip commands
;
Test_RequestPing
     movlw 0x81
     xorwf Command,w
     btfss status,zero
     goto  Test_ResetPIC
     call  SendPingToHost
     goto  AwaitCommandFromHost
;
Test_ResetPIC
     movlw 0x82
     xorwf Command,w
     btfss status,zero
     goto  CommandError
     ; This procedure resets the PIC being programmed.  It pulls the MCLR line low
     ; and holds it low for 10ms.  It asserts the DataLine and ClockLine low so
     ; that the PIC being programmed will be put into programming mode when the
     ; MCLR line is set high again.  Although it is never used for output, the
     ; ClockLine is configured for input when the programmer is powered up.  It is
     ; re-configured for output here just before the PIC being programmed is reset.
     ; After the reset is complete, this procedure sends a 0x55 ping byte to the
     ; Host.
     call  AssertMCLRLineLow
     call  Delay10ms
     call  ConfigDataLineForOutput
     call  ConfigClockLineForOutput
     call  AssertDataLineLow
     call  AssertClockLineLow
     call  Delay1ms                ; delay tset0 (rounded up from 100ns to 1ms)
     call  SetMCLRLineHigh
     call  Delay10ms               ; delay thold0 (rounded up from 5us to 10ms)
     call  SendPingToHost
     goto  AwaitCommandFromHost
;
; Error: Unrecognized command
;
CommandError
     ; CommandError() flashes the Status LED at 200ms high followed by 200ms low.
     call  TurnOnStatusLED
     call  Delay100ms
     call  Delay100ms
     call  TurnOffStatusLED
     call  Delay100ms
     call  Delay100ms
```

```
    goto  CommandError
;
; ************************************************************************************
; Block A subroutines
; Routines to communicate with the Host PC
;   SendPingToHost()
;   ReceiveOneByteFromHost(ByteRecvd)
;   SendOneByteToHost(ByteToSend)
; ************************************************************************************


;
SendPingToHost
; This subroutine sends the ping byte 0x55=b'01010101' to the Host.
    movlw 0x55
    movwf ByteToSend
    call  SendOneByteToHost
    return
;
ReceiveOneByteFromHost
; This subroutine waits for one byte to be received from the Host, which byte is
; returned in register ByteRecvd.  Timing is based on 9600 baud, in which the bit
; duration is 104.167 microseconds.  It is assumed that this subroutine is called
; before the Host has started sending the start bit, when the TX line is still low.
;
; ***CAUTION***
; This subroutine returns at the mid-point of the stop bit, once it has confirmed that
; the TX line is low.  In other words, it returns about 52us before the Host has
; finished sending the stop bit.  The advantage of doing this is that it gives this
; processor time to save the byte received, and then to resume listening for any
; subsequent byte the Host might send.  The host can send a continuous stream at
; 9600 bps and this processor will receive them all (subject to the availability of
; registers in which to save the bytes received).  On the other hand, the disadvantage
; of returning early is that this processor will have to wait before it starts to send
; a byte to the Host.  The length of time it should wait will not be known precisely,
; since it depends on what work this processor did on the last byte it received from
; the Host, and how long that work took.  I have compensated for this disadvantage by
; adding an explicit 100us delay before this processor sends any byte to the Host.
; That will give the Host an extra 48us (at the very least) between the time it ends
; the stop bit on a transmission and time this processor sets the RX line high at the
; the start of a start bit.  This explicit delay is coded at the outset of subroutine
; SendOneByteToHost().
;
;
; Loop to wait for the Host to set the TX line high at the start of a start bit
    movf  portB,w                ; 1 cycle to read portB
    movwf tempROBFH              ; 1 cycle to save in temporary register
    btfss tempROBFH,TX           ; 1 cycle if TX=1; 2 cycles if TX=0
    goto  ReceiveOneByteFromHost ; 2 cycles
;
; The TX line is sampled during the "movf portB" instruction.  If the TX line rises
; an instant before that instruction, it will take 1+1+2=4 cycles to arrive here.  On
; the other hand, if the TX line rises an instant after that instruction, it will take
; 1+1+1+2+1+1+2=9 cycles to arrive here.  On average, we will arrive here 7 cycles
; after the TX line rises, but it could be anywhere between 4 and 9 cycles.
;
; The first thing we want to do is to ensure that the start bit is still high at its
; mid-point, which will occur one-half bit, or 52.084 microseconds, or 260 instruction
; cycles, after the rising edge of the TX line.  Since it takes 7 cycles on average
; to arrive here, we need to wait 253 more cycles before taking the sample.
    call  Delay50us              ; delay exactly 50us = 250 cycles
    nop                          ; 3 nop's add 3 additional cycles
    nop
    nop
    movf  portB,w                ; 1 cycle to read portB (start bit)
    movwf tempROBFH              ; 1 cycle to save in temporary register
```

```
   btfss tempROBFH,TX           ; 1 cycle if TX=1; 2 cycles if TX=0
   goto  StartBitError          ; Error if the start bit has gone low
;
; The next thing we want to do, assuming the start bit is still high at its mid-point,
; is wait until the mid-point of the first bit.  Ideally, the duration from one bit's
; mid-point to the next bit's mid-point is 104.167 microseconds, or 521 instruction
; cycles with a 20MHz clock.  The "movf portB" instruction in the block of code above
; is as close to the mid-point of the start bit as we can get.  It takes 1+1+2=4
; cycles to arrive here.  Therefore, we need to delay 517 more cycles before taking
; the sample of the first bit.  In addition, remember that RS232 inverts all data
; bites, so that, for example, low voltages correspond to logic ones.
   call  Delay100us             ; delay exactly 100us = 500 cycles
   call  Delay1us               ; delay exactly 1us = 5 cycles
   call  Delay1us               ; delay exactly 1us = 5 cycles
   call  Delay1us               ; delay exactly 1us = 5 cycles
   nop                          ; 2 nop's add 2 additional cycles
   nop
   movf  portB,w                ; 1 cycle to read portB (LSB)
   movwf tempROBFH              ; 1 cycle to save in temporary register
   bsf   status,carry           ; 1 cycle to clear the carry flag
   btfsc tempROBFH,TX           ; 1 cycle if TX=1; 2 cycles if TX=0
   bcf   status,carry           ; 1 cycle to set the carry flag
   rrf   ByteRecvd,f            ; 1 cycle to shift the carry into ByteRecvd
;
; The carry flag in the status register is set according to the inverse of the voltage
; on the TX line.  The carry bit is then right-shifted into register ByteRecvd, which
; will hold the byte being sent by the Host.  Since the Host transmits the least-
; significant bit first, right-shifting into ByteRecvd will bring all bits into their
; normal left-to-right order.
;
; Now, we have to wait until the middle of the second bit.  The "movf portB"
; instruction in the block of code above is as close to the mid-point of the first bit
; as we can get.  If the first bit is high (TX=1), it will take 1+1+1+1+1+1=6 cycles
; to arrive here.  If the first bit is low (TX=0), the branching will be slightly
; different and it will take 1+1+1+2+1=6 cycles to arrive here.  Either way, 6 cycles
; are required.  To make up the rest of the required 521 bit-to-bit time, we need to
; delay 515 more instruction cycles.  This is slightly different than the time between
; the middle of the start bit and the middle of the first bit, because the two bits
; are processed differently.
   call  Delay100us             ; delay exactly 515 cycles
   call  Delay1us
   call  Delay1us
   call  Delay1us
   movf  portB,w                ; read bit #2
   movwf tempROBFH
   bsf   status,carry           ; set the carry flag according to /TX
   btfsc tempROBFH,TX
   bcf   status,carry
   rrf   ByteRecvd,f            ; right-shift bit #2 into ByteRecvd
;
; The remaining six data bytes have the same bit-to-bit timing.
   call  Delay100us             ; delay exactly 515 cycles
   call  Delay1us
   call  Delay1us
   call  Delay1us
   movf  portB,w                ; read bit #3
   movwf tempROBFH
   bsf   status,carry           ; set the carry bit according to /TX
   btfsc tempROBFH,TX
   bcf   status,carry
   rrf   ByteRecvd,f            ; right-shift bit #3 into ByteRecvd
   call  Delay100us             ; delay exactly 515 cycles
   call  Delay1us
   call  Delay1us
```

```
    call  Delay1us
    movf  portB,w               ; read bit #4
    movwf tempROBFH
    bsf   status,carry          ; set the carry bit according to /TX
    btfsc tempROBFH,TX
    bcf   status,carry
    rrf   ByteRecvd,f           ; right-shift bit #4 into ByteRecvd
    call  Delay100us            ; delay exactly 515 cycles
    call  Delay1us
    call  Delay1us
    call  Delay1us
    movf  portB,w               ; read bit #5
    movwf tempROBFH
    bsf   status,carry          ; set the carry bit according to /TX
    btfsc tempROBFH,TX
    bcf   status,carry
    rrf   ByteRecvd,f           ; right-shift bit #5 into ByteRecvd
    call  Delay100us            ; delay exactly 515 cycles
    call  Delay1us
    call  Delay1us
    call  Delay1us
    movf  portB,w               ; read bit #6
    movwf tempROBFH
    bsf   status,carry          ; set the carry bit according to /TX
    btfsc tempROBFH,TX
    bcf   status,carry
    rrf   ByteRecvd,f           ; right-shift bit #6 into ByteRecvd
    call  Delay100us            ; delay exactly 515 cycles
    call  Delay1us
    call  Delay1us
    call  Delay1us
    movf  portB,w               ; read bit #7
    movwf tempROBFH
    bsf   status,carry          ; set the carry bit according to /TX
    btfsc tempROBFH,TX
    bcf   status,carry
    rrf   ByteRecvd,f           ; right-shift bit #7 into ByteRecvd
    call  Delay100us            ; delay exactly 515 cycles
    call  Delay1us
    call  Delay1us
    call  Delay1us
    movf  portB,w               ; read bit #8
    movwf tempROBFH
    bsf   status,carry          ; set the carry bit according to /TX
    btfsc tempROBFH,TX
    bcf   status,carry
    rrf   ByteRecvd,f           ; right-shift bit #8 into ByteRecvd
;
; The time from the mid-point of the eighth bit to the mid-point of the stop bit is
; the same as the preceding bit-to-bit time, but what we do to process the stop bit
; is slightly different.  Note that the stop bit must be low; if not, there is a
; problem.
    call  Delay100us            ; delay exactly 515 cycles
    call  Delay1us
    call  Delay1us
    call  Delay1us
    movf  portB,w               ; sample the stop bit
    movwf tempROBFH
    btfsc tempROBFH,TX          ; test the stop bit
    goto  StopBitError          ; Error if the stop bit is high
;
; If the stop bit is low, we can return now, even before the stop bit ends.  We can
; return early because the stop bit is characterized by TX low, which is the same
; voltage as the wait state before the start of the start bit of the next byte.  See
```

```
; my comments above (at the start of this subroutine) about how I compensated by
; adding an explicit 100us delay at the start of subroutine SendOneByteToHost().
     return
StartBitError
     ; Start bit error flashes the Status LED at 200ms high followed by 1000ms low
     call  TurnOnStatusLED
     call  Delay100ms
     call  Delay100ms
     call  TurnOffStatusLED
     call  Delay500ms
     call  Delay500ms
     goto  StartBitError
StopBitError
     ; Stop bit error flashes the Status LED at 1000ms high followed by 200ms low
     call  TurnOnStatusLED
     call  Delay500ms
     call  Delay500ms
     call  TurnOffStatusLED
     call  Delay100ms
     call  Delay100ms
     goto  StopBitError
;
SendOneByteToHost
; This subroutine sends the byte in register ByteToSend to the Host.  Timing is based
; on 9600 baud, in which the bit duration is 104.167 microseconds, or 521 instruction
; cycles with a 20Hz clock.  It is assumed that the Host is ready and able to receive.
; In this subroutine, the delays needed to get the timing correct must be calculated
; backwards, starting with the known time at which the next event must occur, and
; subtracting from it the cycles which are going to be needed to prepare for that next
; event.  The least significant bit of ByteToSend is sent first.  Remember that RS232
; inverts all data bits, so that, for example, low voltages correspond to logic ones.
; This inversion does not apply to the start and stop bits; for greater certainty, the
; start bit is sent at the high voltage and the stop bit is sent at the low voltage.
;
; ***CAUTION***
; Like subroutine ReadOneByteFromHost(), this subroutine returns early, at the mid-
; point of the stop bit.  If the transmission generated by this subroutine is followed
; immediately by another transmission, the 100us delay in the following instruction is
; sufficient to ensure that there is no overlap.  On the other hand, if this
; transmission is followed by the Host sending a byte, returning early gives the
; calling routine time to get ready to receive.
;
; The following 100us delay is required to compensate for the fact that subroutine
; ReadOneByteFromHost() returns about 52us early.
     call  Delay100us
;
; Start the start bit.
     bsf   portBmirror,RX
     movf  portBmirror,w
     movwf portB                   ; voltages on the RX line change now
;
; The first data bit must be presented 521 cycles later.  Preparation to send bit #1
; takes 6 cycles (see the next block of code), so the required explicit delay is 515
; cycles.
     call  Delay100us              ; delay exactly 100us = 500 cycles
     call  Delay1us                ; delay exactly 1us = 5 cycles
     call  Delay1us                ; delay exactly 1us = 5 cycles
     call  Delay1us                ; delay exactly 1us = 5 cycles
;
; The following block of code is the preparation needed to send bit #1.  The LSB of
; ByteToSend is right-shifted into the carry bit.  Right-shifting ensures that the LSB
; is transmitted first.  Then, the RX bit in register portBmirror is set high or
; low according to the carry bit.  Remember that the RS232 voltages are the complement
; of the logic.  Finally, portBmirror is sent to portB, and the RX line goes high or
```

```
; low.  The following code takes 1+1+2+1+1=6 cycles, whether bit #1 is high or low.
     rrf   ByteToSend,f           ; shift LSB into the carry bit; 1 cycle
     bsf   portBmirror,RX         ; 1 cycle
     btfsc status,carry           ; 1 cycle is RX=1; 2 cycles if RX=0
     bcf   portBmirror,RX         ; 1 cycle, but only if RX=1
     movf  portBmirror,w          ; 1 cycle
     movwf portB                  ; send LSB to the RX line; 1 cycle
;
; The following seven data bits take the same processing time as bit #1.
     call  Delay100us             ; delay exactly 515 cycles
     call  Delay1us
     call  Delay1us
     call  Delay1us
     rrf   ByteToSend,f           ; shift bit #2 into the carry bit
     bsf   portBmirror,RX         ; set portBmirror<RX> equal to /(carry bit)
     btfsc status,carry
     bcf   portBmirror,RX
     movf  portBmirror,w
     movwf portB                  ; send bit #2 to the RX line
     call  Delay100us             ; delay exactly 515 cycles
     call  Delay1us
     call  Delay1us
     call  Delay1us
     rrf   ByteToSend,f           ; shift bit #3 into the carry bit
     bsf   portBmirror,RX         ; set portBmirror<RX> equal to /(carry bit)
     btfsc status,carry
     bcf   portBmirror,RX
     movf  portBmirror,w
     movwf portB                  ; send bit #3 to the RX line
     call  Delay100us             ; delay exactly 515 cycles
     call  Delay1us
     call  Delay1us
     call  Delay1us
     rrf   ByteToSend,f           ; shift bit #4 into the carry bit
     bsf   portBmirror,RX         ; set portBmirror<RX> equal to /(carry bit)
     btfsc status,carry
     bcf   portBmirror,RX
     movf  portBmirror,w
     movwf portB                  ; send bit #4 to the RX line
     call  Delay100us             ; delay exactly 515 cycles
     call  Delay1us
     call  Delay1us
     call  Delay1us
     rrf   ByteToSend,f           ; shift bit #5 into the carry bit
     bsf   portBmirror,RX         ; set portBmirror<RX> equal to /(carry bit)
     btfsc status,carry
     bcf   portBmirror,RX
     movf  portBmirror,w
     movwf portB                  ; send bit #5 to the RX line
     call  Delay100us             ; delay exactly 515 cycles
     call  Delay1us
     call  Delay1us
     call  Delay1us
     rrf   ByteToSend,f           ; shift bit #6 into the carry bit
     bsf   portBmirror,RX         ; set portBmirror<RX> equal to /(carry bit)
     btfsc status,carry
     bcf   portBmirror,RX
     movf  portBmirror,w
     movwf portB                  ; send bit #6 to the RX line
     call  Delay100us             ; delay exactly 515 cycles
     call  Delay1us
     call  Delay1us
     call  Delay1us
     rrf   ByteToSend,f           ; shift bit #7 into the carry bit
```

```
     bsf   portBmirror,RX          ; set portBmirror<RX> equal to /(carry bit)
     btfsc status,carry
     bcf   portBmirror,RX
     movf  portBmirror,w
     movwf portB                   ; send bit #7 to the RX line
     call  Delay100us              ; delay exactly 515 cycles
     call  Delay1us
     call  Delay1us
     call  Delay1us
     rrf   ByteToSend,f            ; shift bit #8 into the carry bit
     bsf   portBmirror,RX          ; set portBmirror<RX> equal to /(carry bit)
     btfsc status,carry
     bcf   portBmirror,RX
     movf  portBmirror,w
     movwf portB                   ; send bit #8 to the RX line
;
; The stop bit must be presented 521 cycles later.  Preparation for the stop bit takes
; only 3 cycles (see the next block of code), so the required delay is 518 cycles.
     call  Delay100us              ; delay exactly 100us = 500 cycles
     call  Delay1us                ; delay exactly 1us = 5 cycles
     call  Delay1us                ; delay exactly 1us = 5 cycles
     call  Delay1us                ; delay exactly 1us = 5 cycles
     nop                           ; 3 nop's take 3 cycles
     nop
     nop
;
; The following block of code is the preparation needed to send the stop bit, which is
; always low.  It takes 3 cycles.
     bcf   portBmirror,RX          ; 1 cycle
     movf  portBmirror,w           ; 1 cycle
     movwf portB                   ; 1 cycle
;
; The stop bit must end 104.2us = 521 cycles later.  It is not necessary to assert the
; RX line low since the stop bit voltage cannot be distinguished from the low voltage
; at which is the RX line is held between bytes.  For the reasons I described in my
; comments at the start of this subroutine, I will return for this subroutine at the
; mid-point of the stop bit.
     call  Delay50us               ; delay 5us, or one-half bit
     call  Delay1us
     call  Delay1us
     return
;
; ************************************************************************************
; Block B subroutines
; Routines to read and write from the PIC being programmed
;   WriteCommandToPIC()
;   Write14BitWordToPIC()
;   Read14BitWordFromPIC()
; ************************************************************************************
;
WriteCommandToPIC
; This subroutine writes the low-order six bits of Command to the PIC, with the LSB
; sent first.  This routine waits thld0 (rounded up from 5us to 10us) before the
; first rising edge of the ClockLine.  It waits tset1 (rounded up from 100ns to 1us)
; after the DataLine is set/reset before pulling the ClockLine low.  It waits thld1
; (rounded up from 100ns to 1us) after pulling the ClockLine low before changing the
; DataLine.  After the sixth and last Clock pulse, it waits an additional tdly1
; (rounded up from 1us to 2us) before returning.
     call  Delay10us               ; delay thld0
     movlw 0x06                    ; 0x06 = d'6
     movwf tempWCTP                ; counter for six bits
WCTP1
     call  SetClockLineHigh        ; set the ClockLine high
     rrf   Command,f               ; right-shift LSB of command into carry bit
```

```
     btfsc status,carry
     goto  WCTP2
     call  AssertDataLineLow       ; bit=0 --> assert the DataLine low
     goto  WCTP3
WCTP2
     call  SetDataLineHigh         ; bit=1 --> set the DataLine high
WCTP3
     call  Delay1us                ; delay tset1
     call  AssertClockLineLow
     call  Delay1us                ; delay thld1
     decfsz tempWCTP,f             ; decrement the counter
     goto  WCTP1                   ; start to process the next bit
     call  Delay1us                ; delay tdly1
     call  Delay1us
     return
;
Write14BitWordToPIC
; This subroutine writes a 14-bit word to the PIC.  At the outset, the word is
; stored right-justified in the register pair HighBytePIC, LowBytePIC.  The register
; pair is shifted left one place so that the 14 bits are centered.  The values of the
; MSB and LSB do not matter.  All 16 bits are transmitted to the PIC, starting with
; the LSB.  This subroutine waits tdly2 (rounded up from 100ns to 1us) before the
; first rising edge of the ClockLine.  It waits tset1 (rounded up from 100ns to 1us)
; after the DataLine is set/reset before pulling the ClockLine low.  It waits
; thld1 (rounded up from 100ns to 1us) after pulling the ClockLine low before
; changing the DataLine.  There is no extra wait tie after the 16th and last Clock
; pulse.  The DataLine is returned with a random value.
     rlf   LowBytePIC,f            ; shift the register pair left one place
     rlf   HighBytePIC,f
     call  Delay1us                ; delay tdly2
     movlw 0x10                    ; 0x10 = d'16
     movwf tempWWTP                ; counter for 16 bits
WWTP1
     call  SetClockLineHigh        ; set the ClockLine high
     rrf   HighBytePIC,f           ; right-shift LSB of word into carry bit
     rrf   LowBytePIC,f
     btfsc status,carry
     goto  WWTP2
     call  AssertDataLineLow       ; bit=0 --> assert the DataLine low
     goto  WWTP3
WWTP2
     call  SetDataLineHigh         ; bit=1 --> set the DataLine high
WWTP3
     call  Delay1us                ; delay tset1
     call  AssertClockLineLow
     call  Delay1us                ; delay thld1
     decfsz tempWWTP,f             ; decrement the counter
     goto  WWTP1                   ; start to process the next bit
     return
;
Read14BitWordFromPIC
; This subroutine reads a 14-bit word from the PIC.  At the end, the word is stored
; right-justified in the register pair HighBytePIC, LowBytePIC.  The two high-order
; bits of HighBytePIC are b'00'.  This subroutine waits tdly2 (rounded up from 100ns
; to 1us) before the first rising edge of the ClockLine.  It waits tdly3 (rounded up
; from 80ns to 1us) to allow the PIC to set/reset the DataLine, before reading the
; bit.  It waits an additional 1us (an arbitrary delay not required in the datasheet)
; before pulling the ClockLine low.  The routine then waits a further 1us (also not
; required in the datasheet) before setting the ClockLine high for the net bit.  As
; the bits are received, they are right-shifted into the left, top-order, end of the
; register pair.  Once all 14 bits have been pushed into the register pair, it is then
; right-shifted twice more and cleared at the upper end.  Note that the first and last
; (16 bits) are not part of the 14-bit word.  They are clocked separately from the 14
; meaningful bits because the DataLine must be re-configured at the correct times
```

```
; during the first and 16th bits.
; Three more things:
; 1.  The DataLine is returned with a random value.
; 2.  This subroutine is responsible for setting the configuration of portB to allow
;     for reading the DataLine.  This subroutine comes in and goes out with the
;     RB1 pin configures for output.
; 3.  When a byte is read from EEPROM data memory, the second through ninth bits in
;     the 16 bits received comprise the relevant byte, in reverse order.  It is
;     therefore the case that the EEPROM byte (in its proper order) is register
;     LowBytePIC.  HighBytePIC contains garbage and can be ignored.
    call  Delay1us                ; delay tdly2
    ; clock through the first bit with the DataLine still configured for output
    call  SetClockLineHigh        ; set the ClockLine high
    call  Delay1us
    call  AssertClockLineLow
    call  Delay1us                ; arbitrary extra delay
    movlw 0x0E                    ; 0x0E = d'14'
    movwf tempRWFP                ; counter for following 14 bits
    ; re-configure the DataLine for input just before the start of the second bit
    call  ConfigDataLineForInput
RWFP
    call  SetClockLineHigh        ; set the ClockLine high
    call  Delay1us
    call  ReadDataLine            ; returns bit read in the carry flag
    rrf   HighBytePIC,f           ; right-shift carry bit into byte pair
    rrf   LowBytePIC,f
    call  Delay1us                ; arbitrary extra delay
    call  AssertClockLineLow
    call  Delay1us                ; arbitrary extra delay
    decfsz tempRWFP,f             ; decrement the counter
    goto  RWFP                    ; start to process the next bit
    ; clock through the last bit, re-configuring the DataLine at the right moment
    call  SetClockLineHigh        ; set the ClockLine high
    call  Delay1us
    call  AssertClockLineLow
    call  ConfigDataLineForOutput
    ; right-justify the 14-bit word
    rrf   HighBytePIC,f
    rrf   LowBytePIC,f
    rrf   HighBytePIC,f
    rrf   LowBytePIC,f
    ; clear the top two bits of the high byte
    movf  HighBytePIC,w
    andlw 0x3F
    movwf HighBytePIC
    return
;
; ****************************************************************************************
; Block C subroutines
; Controls for the Clock, Data and Reset lines of the PIC being programmed
;   ConfigDataLineForOuput()
;   ConfigDataLineForInput()
;   ConfigClockLineForOutput()
;   AssertDataLineLow()
;   SetDataLineLow()
;   AssertClockLineLow()
;   SetClockLineHigh()
;   AssertMCLRLineLow()
;   SetMCLRLineHigh()
;   TurnOnStatusLED()
;   TurnOffStatusLED()
;   ReadDataLine()
; ****************************************************************************************
```

```
;
ConfigDataLineForOutput
     ; Do not clear TRISC bits directly.  DataLine is portC<2>.
     bsf   status,page0            ; select register bank 1
     movf  TRISC,w
     andlw 0xFB                    ; 0xFB = b'11111011'
     movwf TRISC
     bcf   status,page0            ; return to register bank 0
     return
;
ConfigDataLineForInput
     ; Do not set TRISC bits directly.  DataLine is portC<2>.
     bsf   status,page0            ; select register bank 1
     movf  TRISC,w
     iorlw 0x04                    ; 0x04 = b'00000100'
     movwf TRISC
     bcf   status,page0            ; return to register bank 0
     return
;
ConfigClockLineForOutput
; The program starts with both the DataLine and the ClockLine configured for input.
; Just before the PIC being programmed is powered up (reset), both lines are
; re-configured for output, and asserted low.  The ClockLine remains configured for
; output thereafter, but the DataLine is used for both reads and writes to the PIC
; being programmed.  Do not clear TRISC bits directly.  ClockLine is portC<3>.
     bsf   status,page0            ; select register bank 1
     movf  TRISC,w
     andlw 0xF7                    ; 0xF7 = b'11110111'
     movwf TRISC
     bcf   status,page0            ; return to register bank 0
     return
;
AssertDataLineLow
     ; Call only when the DataLine is configured for output
     bcf   portCmirror,DataLine
     movf  portCmirror,w
     movwf portC
     return
;
SetDataLineHigh
     ; Call only when the DataLine is configured for output
     bsf   portCmirror,DataLine
     movf  portCmirror,w
     movwf portC
     return
;
AssertClockLineLow
     ; Call only when the ClockLine is configured for output
     bcf   portCmirror,ClockLine
     movf  portCmirror,w
     movwf portC
     return
;
SetClockLineHigh
     ; Call only when the ClockLine is configured for output
     bsf   portCmirror,ClockLine
     movf  portCmirror,w
     movwf portC
     return
;
AssertMCLRLineLow
     bcf   portBmirror,MCLRLine
     movf  portBmirror,w
     movwf portB
```

```
	return
;
SetMCLRLineHigh
	bsf   portBmirror,MCLRLine
	movf  portBmirror,w
	movwf portB
	return
;
TurnOnStatusLED
	bsf   portAmirror,StatusLED
	movf  portAmirror,w
	movwf portA
	return
;
TurnOffStatusLED
	bcf   portAmirror,StatusLED
	movf  portAmirror,w
	movwf portA
	return
;
ReadDataLine
; This subroutine reads the DataLine and returns the value in the carry flag of
; the status register.  Call only when the DataLine is configured for input.
	movf  portC,w
	movwf tempRDL
	bcf   status,carry
	btfsc tempRDL,DataLine
	bsf   status,carry
	return
;
; **************************************************************************************

; Block D subroutines
; Timing and delays
;   Delay1us() - uninterruptable delay of exactly one microsecond
;   Delay10us() - uninterruptable delay of exactly 10 microseconds
;   Delay50us() - uninterruptable delay of exactly 50 microseconds
;   Delay100us() - uninterruptable delay of exactly 100 microseconds
;   Delay1ms() - uninterruptable delay of about one millisecond
;   Delay10ms() - uninterruptable delay of about 10 milliseconds
;   Delay100ms() - uninterruptable delay of about 100 milliseconds
;   Delay500ms() - uninterruptable delay of about 500 milliseconds
; **************************************************************************************
;
Delay1us
; This subroutine is an uninterruptable delay of exactly one microsecond, including
; the invoking "call".  At 20MHz, each instruction cycle takes 200ns, or 0.2us.  To
; delay 1us, we need 5 instruction cycles.  The "call" takes 2 instruction cycles.
; The "return" takes 2 instruction cycles.  The "nop" takes 1 instruction cycle.
	nop
	return
;
Delay10us
; This subroutine is an uninterruptable delay of exactly 10 microseconds, including
; the invoking "call".  This is equal to 50 instruction cycles at 20MHz.
	call  Delay1us              ; 5 cycles
	movlw 0x0A                  ; 1 cycle; 0x0A = d'10'
	movwf tempDus               ; 1 cycle
D10us
	nop                         ; 10 cycles
	decfsz tempDus,f            ; 9 interim tests + 2 on final = 11 cycles
	goto  D10us                 ; 9 x 2 cycles = 18 cycles
	return                      ; 2 cycles
;
```

```
Delay50us
; This subroutine is an uninterruptable delay of exactly 50 microseconds, including
; the invoking "call".  This is equal to 250 instruction cycles at 20MHz.
     nop                          ; 1 cycle
     movlw 0x3D                   ; 1 cycle; 0x3D = d'61'
     movwf tempDus                ; 1 cycle
D50us
     nop                          ; 61 cycles
     decfsz tempDus,f             ; 60 interim tests + 2 on final = 62 cycles
     goto  D50us                  ; 60 x 2 cycles = 120 cycles
     return                       ; 2 cycles
;
Delay100us
; This subroutine is an uninterruptable delay of exactly 100 microseconds, including
; the invoking "call".  This is equal to 500 instruction cycles are 20MHz.
     nop                          ; 5 cycles for the nop's
     nop
     nop
     nop
     nop
     movlw 0x62                   ; 1 cycle; 0x62 = d'98'
     movwf tempDus                ; 1 cycle
D100us
     nop                          ; 98 cycles
     nop                          ; 98 cycles
     decfsz tempDus,f             ; 97 interim tests + 2 on final = 99 cycles
     goto  D100us                 ; 97 x 2 cycles = 194 cycles
     return                       ; 2 cycles
;
Delay1ms
; This subroutine is an uninterruptable delay of about one millisecond.  It calls
; subroutine Delay100us() ten times.
     call  Delay100us
     call  Delay100us
     call  Delay100us
     call  Delay100us
     call  Delay100us
     call  Delay100us
     call  Delay100us
     call  Delay100us
     call  Delay100us
     call  Delay100us
     return
;
Delay10ms
; This subroutine is an uninterruptable delay of about ten milliseconds.  It calls
; subroutine Delay100us() ten times.
     call  Delay1ms
     call  Delay1ms
     call  Delay1ms
     call  Delay1ms
     call  Delay1ms
     call  Delay1ms
     call  Delay1ms
     call  Delay1ms
     call  Delay1ms
     call  Delay1ms
     return
;
Delay100ms
; This subroutine is an uninterruptable delay of about 100 milliseconds.  It calls
; subroutine Delay10ms() ten times.
     call  Delay10ms
     call  Delay10ms
```

```
    call  Delay10ms
    call  Delay10ms
    call  Delay10ms
    call  Delay10ms
    call  Delay10ms
    call  Delay10ms
    call  Delay10ms
    call  Delay10ms
    return
;
Delay500ms
; This subroutine is an uninterruptable delay of about 500 milliseconds.  It calls
; subroutine Delay100ms() five times.
    call  Delay100ms
    call  Delay100ms
    call  Delay100ms
    call  Delay100ms
    call  Delay100ms
    return
;
; ***************************************************************************************

; Block E subroutines for debugging purposes only
; ***************************************************************************************
;
; ***************************************************************************************

; Display routine for testing and debugging purposes only
; The following routine is not used in the program above.  It is included here simply
; for convenience during debugging.  This routine takes the eight bits in the
; accumulator (register w) and displays them one-by-one on the Status LED.  Timing is
; important for interpreting the display.  Each of the eight bits takes one second to
; display, with the bit value (LED on for b'1' and LED off for b'0') display for 500ms
; followed by 500ms with the LED off.  This pattern continues until the programmer is
; turned off.  But, to distinguish between displays of the byte, there are three short
; blinks of 100ms on followed by 100ms off.  Note that the eight bits are displayed
; in the order with the least-significant bit first.
; ***************************************************************************************

;
DisplayAByte                       ; displays w<0-7> on the Status LED
    movwf ByteToDisplay            ; temporary save
DAB1
    call  TurnOffStatusLED         ; three short blinks
    call  Delay100ms
    call  TurnOnStatusLED
    call  Delay100ms
    call  TurnOffStatusLED
    call  Delay100ms
    call  TurnOnStatusLED
    call  Delay100ms
    call  TurnOffStatusLED
    call  Delay100ms
    call  TurnOnStatusLED
    call  Delay100ms
    call  TurnOffStatusLED
    call  Delay100ms
    movf  ByteToDisplay,w
    movwf tempDAB1                 ; tempDAB1 holds the 8 bits to be displayed
    movlw 0x08
    movwf tempDAB2                 ; tempDAB2 is counter for 8 bits
    movlw 0x0F
DAB2
    bcf   portAmirror,StatusLED    ; as the default, turn the LED bit off
```

```
    rrf   tempDAB1,f              ; right0shift the next LSB into carry flag
    btfsc status,carry
    bsf   portAmirror,StatusLED   ; if carry flag is high, set the LED bit
    movf  portAmirror,w
    movwf portA                   ; turn the LED on or off as per the logic bit
    call  Delay500ms              ; hold the display for 500ms
    call  TurnOffStatusLED        ; now turn the LED off
    call  Delay500ms              ; hold off for 500ms
    decfsz tempDAB2,f             ; repeat 8 times
    goto  DAB2                    ; start to process the next bit
    goto  DAB1                    ; end of byte; start the display over again
;
; ************************************************************************************
; Communications test #1 for debugging purposes only
; The following routine is not used in the program above.  It is included here simply
; for convenience during debugging.  This routine is a continuous loop which sends an
; incrementing byte to the Host.  The bytes are sent in groups of eight, with one
; millisecond between bytes and 500ms between groups.
; ************************************************************************************

;
CommTest1
    call  Delay100ms
    movlw 0x08
    movwf tempCT1                 ; tempCT1 is the 8-byte counter
    clrf  tempCT2                 ; tempCT2 is the incrementing byte
CT1
    incf  tempCT2,f               ; increment the byte
    movf  tempCT2,w
    movwf ByteToDisplay
    call  DisplayAByte            ; display the byte to be sent
    movf  tempCT2,w
    movwf ByteToSend
    call  SendOneByteToHost       ; send the byte to the Host
    call  Delay1ms                ; delay 1ms between bytes
    decfsz tempCT1,f
    goto  CT1                     ; end of 8-byte loop
    call  Delay500ms              ; delay 500ms between groups
    movlw 0x08
    movwf tempCT1                 ; initialize counter for next group
    goto  CT1
;
; ************************************************************************************
; Communications test #2 for debugging purposes only
; The following routine is not used in the program above.  It is included here simply
; for convenience during debugging.  This routine is a continuous loop which receives
; an incrementing byte from the Host.  The bytes are sent in groups of eight, with one
; millisecond between bytes and 500ms between groups.
; ************************************************************************************

;
CommTest2
    call  ReceiveOneByteFromHost  ; wait for the first byte to be received
    movf  ByteRecvd,w
    movwf tempCT1                 ; tempCT1 holds the byte expected next
CT2
    incf  tempCT1,f               ; increment the expected byte
    call  ReceiveOneByteFromHost  ; wait for the byte to be received
    movf  ByteRecvd,w
    movwf ByteToDisplay
    call  DisplayAByte            ; display the byte received
    movf  ByteRecvd,w
    xorwf tempCT1,w               ; compare the expected and received bytes
    btfsc status,zero             ; zero bit is one if they are the same
```

```
    goto  CT2                     ; goto if they are the same
    call  TurnOnStatusLED         ; if there is an error, turn on the LED
    call  Delay500ms              ; leave the LED on for one-half second ...
    goto  CommTest2               ; ... then start all over again
;
; ****************************************************************************************
; Communications test #3 for debugging purposes only
; The following routine is not used in the program above.  It is included here simply
; for convenience during debugging.  This routine is a continuous loop which receives
; an incrementing byte from the Host.  This routine complements the byte received,
; waits one millisecond and then sends it back to the Host.  The Host will send the
; next incremented byte after one millisecond.
; ****************************************************************************************
;
CommTest3
    call  ReceiveOneByteFromHost  ; wait for the next byte to be received
    movf  ByteRecvd,w
    movwf ByteToDisplay
    call  DisplayAByte            ; display the byte received
    movf  ByteRecvd,w
    xorlw 0xFF                    ; complement the byte received
    movwf ByteToSend
    call  Delay1ms                ; delay one millisecond
    call  SendOneByteToHost       ; send the complemented byte to the Host
    goto  CommTest3
;
; ****************************************************************************************
; EEPROM data for testing purposes only
; The following data is not used in the program above.  It is included here simply for
; convenience should the hex file for this program be used to test-program a PIC.
; ****************************************************************************************
;
    org   0x2100
    de    "These are some test EEPROM bytes.", 1, "to", 99
;
; Code in hex is:
; T=0x54  h=0x68  e=0x65  s=0x73  e=0x65  blank=0x20  a=0x61  r=0x72  e=0x65
; blank=0x20  s=0x73  o=0x6F  m=0x6D  e=0x65  blank=0x20 t=0x74  e=0x65  s=0x73
; t=0x74  blank=0x20  E=0x45  E=0x45  P=0x50  R=0x52  O=0x4F  M=0x4D  blank=0x20
; b=0x62  y=0x79  t=0x74  e=0x65  s=0x73  .=2E  1=0x01  t=0x74  o=0x6F  99=0x99
; Note that the last number '99' is treated as a one-byte hex value.
;
    END   ; end assembly
```

Appendix "D"

Assembly listing for programmer - PIC16F872

```
; Program for PIC Programmer using 16F872 device
;
; This program does not use interrupts.  The subroutines are grouped into
; the following blocks:
; A.  Routines to communicate with the Host PC
; B.  Routines to read and write to the PIC being programmed
; C.  Controls for the Clock, Data and Reset lines of the PIC being programmed
; D.  Timing and delays
; E.  Routines used for debugging purposes only
;
; The program detects three error conditions when receiving data from the Host.
; It alerts the user by flashing the Status LED in the following ways:
; 1.  Start bit is high - 500ms on followed by 1000ms off
; 2.  Stop bit is low - 1000ms on followed by 500ms off
; 3.  Unknown command - 200ms on followed by 200ms off
;
; For maximum versatility of the application, I have decided to have the Host computer
; do as much as possible of the timing and flow control, and to have the programmer do
; as little as possible.  The programmer does not need to have any knowledge about
; the PIC it is programming.  I have made this decision because it is easier to make
; all changes (to accommodate new PICs, for example) in the Host's VisualBasic program
; than to make some changes there and some in the programmer.  Therefore, single-byte
; commands sent by the Host to the programmer come in two types:
; 1.  Microchip flash programming commands, which are written to the PIC being
;     programmed as soon as they are received.  All Microchip flash programming
;     command have a zero most-significant bit.  The following Microchip commands are
;     recognized.  The data bytes which follow these commands, if any, are described
;     in round parentheses.  If the command applies only to one series of 16FXXX
;     devices, the series name is described in square brackets.  Further details are
;     given in the blocks of code below which process these commands.
;     0x00             Load configuration (2 bytes will follow)
;      0x01            Bulk erase setup1 [16F87X only]
;      0x02            Load data for program memory (2 bytes will follow)
;      0x03            Load data for EEPROM memory (2 bytes will follow)
;      0x04            Read data from program memory (2 bytes to be sent to Host)
;      0x05            Read data from EEPROM memory (2 bytes to be sent to Host)
;      0x06            Increment address
;      0x07            Bulk erase setup2 [16F87X only]
;      0x08            Begin erase/programming cycle
;      0x09            Bulk erase program memory
;      0x0A            End programming [16F88X only]
;      0x0B            Bulk erase EEPROM memory
;      0x17            End programming [16F87XA only]
;      0x18            Begin programming only cycle
;      0x1F            Chip erase [16F87XA only]
; 2.  Other commands, like an MCLR reset of the PIC being programmed, which are not
;     programming commands.  These commands are identified by a most-significant bit
;     which is high.
;      0x81            Request programmer to send a &H55 ping byte
;      0x82            Reset the PIC being programmed
;
; RS-232 communication with the Host computer
; 1.  Communication with the Host is done on a byte-by-byte basis, where the bit
;     rate is fixed at 9600 bps, corresponding to 104.2 microseconds per bit.
; 2.  This PIC does not use interrupt-driven send and receive buffers.  This imposes
;     some constraints.  For example, this PIC must be executing the subroutine
;     ReceiveByteFromHost() at the instant when the Host actually transmits a byte,
;     or parts of the byte will be lost.  The Host should never send a byte to this
;     PIC unless it is confident this PIC is listening.  Transmission from this PIC
```

```
;     to the Host is less sensitive since the Host uses a read buffer.
; 3.  When sending a byte to the Host, this PIC stops monitoring the transmission
;     after it starts sending the stop bit.  That is because the low voltage during
;     the stop bit is indistinguishable from the low voltage on the RX line which
;     prevails between bytes.  The advantage of not waiting out the duration of the
;     stop bit is that it gives this PIC 100us or so to do something else.  The
;     disadvantage of not waiting out the stop bit is that this PIC must take care not
;     send a second byte to the Host without waiting out the equivalent time.
; 4.  When receiving a byte from the Host, this PIC stops monitoring the transmission
;     after it confirms the stop bit.  Since is confirms the stop bit midway through
;     it scheduled duration, the subroutine ReceiveOneByteFromHost() returns about
;     one-half bit, or 52us, before the end of the stop bit.  The advantage of not
;     waiting out the stop bit is that it gives this PIC 50us or so to do something
;     else.  The disadvantage of not waiting out the stop bit is that this PIC must
;     take care not to send a byte to the Host without waiting out the equivalent
;     time.
; 5.  Paragraphs 3 and 4 mean that there will not be a problem when this PIC sends a
;     byte to the Host and then immediately sits back and prepares to receive a byte
;     from the Host.  The wait loop at the start of ReceiveOneByteFromHost() will
;     run until the Host gets around to responding.  This conclusion applies so long
;     as "immediately" in the previous sentence does not exceed 104us or so,
;     corresponding to the duration of one bit.
; 6.  The problem in paragraphs 3 and 4 arises only when this PIC receives a byte from
;     the Host, does some processing, and then needs to send something back to the
;     Host.  The reply could be data or it could simply be a ping to tell the Host
;     that it has completed an operation.  Either way, this PIC cannot start sending
;     its reply before a one-half bit time, being 52us or so, has elapsed.
; 7.  To help manage communications, this PIC sends a ping byte &H55 to the Host after
;     receiving every command and after completing whatever work is required to deal
;     with the command received.
;
; Configuration Word for 16F872
;          b<13-12>=1  Turn OFF program memory protection
;          b<11>=1         Disable in-circuit debugger
;          b<10>=1         Unimplemented, reads as 1
;          b<9>=1          WRT enabled
;          b<8>=1          Turn OFF EEPROM memory protection
;          b<7>=0          Disable Low-Voltage Programming
;          b<6>=0          Disable brown-out reset
;          b<5-4>=1    Data protection off
;          b<3>=1          Disable power-up timer
;          b<2>=0          Disable watch-dog timer
;          b<1-0>=10   Set HS oscillator gain
  #include "p16F872.inc"
  processor     16F872
  __CONFIG 0x3F3A          ; b'xx11 1111 0011 1010'
;
; Crystal frequency is 20MHz, so the instruction cycle time is 200ns
;
; ***************************************************************************************
; Variable definitions - PIC 16F872 system registers
; ***************************************************************************************
;
; Registers in Bank0
timer0          equ   0x01        ; Timer0 count register
status          equ   0x03        ; status register
carry           equ         0x00  ; carry from MSB occurred
zero            equ         0x02  ; result of operation is zero
page0           equ         0x05  ; register bank selector low bit
page1           equ         0x06  ; register bank selector high bit
portA           equ   0x05
portB           equ   0x06
portC           equ   0x07
INTCon          equ   0x0B        ; interrupt control register
```

```
GIE             equu        0x07  ; global interrupt enable
```

```
tempRDL         equ   0x45        ; Used in subroutine ReadDataLine()
tempDus         equ   0x46        ; Used in subroutines Delay***us()
ByteToDisplay   equ   0x47        ; Used in subroutine DisplayAByte()
tempDAB1        equ   0x48        ; Used in subroutine DisplayAByte()
tempDAB2        equ   0x49        ; Used in subroutine DisplayAByte()
tempCT1         equ   0x4A        ; Used in subroutines CommTest*()
tempCT2         equ   0x4B        ; Used in subroutines CommTest*()
;
; ************************************************************************************
; Hard start
; ************************************************************************************
    org   0x0000
    goto  InitializeSystemRegisters
;
; ************************************************************************************
; InitializeSystemRegisters
; ************************************************************************************
    org   0x0010
InitializeSystemRegisters
    clrf  INTCon                  ; disable all interrupt activity
    bcf   status,page0            ; select register bank 0
    bcf   status,page1            ; (The page1 bit is never changed)
    movlw 0x00                    ; set T1CON=0 to disable Timer1 and ...
    movwf T1CON                   ; ... release portC<1-0> for digital I/O
    movlw 0x00                    ; set SSPCON<5>=0 to disable serial port and ...
    movwf SSPCON                  ; ... release portA<5> for digital I/O
    movlw 0x00                    ; set CCP1CON=0 to disable Capture/Compare/PWM ...
    movwf CCP1CON                 ; ... and release portC<2> for digital I/O
    bsf   status,page0            ; select register bank 1
    bcf   optionreg,7             ; enable PortB pull-up resistors
    bsf   optionreg,6             ; trigger RB0 interrupts on rising edge
    bcf   optionreg,5             ; increment timer0 using internal clock
    bcf   optionreg,4             ; increment timer0 on RA4 rising edge
    bcf   optionreg,3             ; apply prescalar to timer0
    bsf   optionreg,2             ; code b'111' sets ...
    bsf   optionreg,1             ; ... timer0 prescalar ...
    bsf   optionreg,0             ; ... to 1:256
    movlw 0x06                    ; set ADCON1<3-1>=b'011' to ...
    movwf ADCON1                  ; ... configure portA for digital I/O
ConfigurePorts
    ; The RA0 pin controls the statusLED and is always configured for output.
    movlw 0x00                    ; RA0-RA5=Output
    movwf TRISA
    ; We will power up with the TX line (RB0) configured for input.  We can do
    ; this because the Host computer will have opened the serial port before
    ; this PIC is powered up.  Therefore, there will be already be a stable
    ; voltage on the TX line when execution reaches this point.  The RX line
    ; (RB1) and the MCLR line to the PIC being programmed (RB2) will be
    ; configured for output.  These I/O directions are never changed.
    movlw 0x01                    ; RB0(TX)=Input, other pins output
    movwf TRISB
    ; To avoid any possible voltage conflicts, we will power up with the
    ; DataLine (RC2) and ClockLine (RC3) leading to the PIC being programmed
    ; both configured for input.  These two lines will be reconfigured for
    ; output, and their voltages pulled low, immediately before the PIC being
    ; programmed is reset.  Putting the PIC being programmed into programming
    ; mode requires that the DataLine and ClockLine be low at the time the MCLR
    ; voltage is raised from zero to 13V.  This is done in subroutine
    ; ResetThePIC().
    movlw 0x0C                    ; RC2-RC3=Input, other pins output
    movwf TRISC
    bcf   status,page0            ; return to register bank 0
InitializePortContents
    clrf  portAmirror
```

```
    bsf   portAmirror,StatusLED   ; turn on the StatusLED
    movf  portAmirror,w
    movwf portA
    clrf  portBmirror             ; assert the RX and MCLR lines low
    movf  portBmirror,w
    movwf portB
    clrf  portCmirror             ; assert the non-connected lines of portC
    movf  portCmirror,w
    movwf portC
VisualConfirmationOfStartup
    call  Delay500ms              ; hold StatusLED on for one-half second ...
    bcf   portAmirror,StatusLED   ; ... so the user can verify startup
    movf  portAmirror,w
    movwf portA
;
; ************************************************************************************
; Synchronize with the Host by exchanging pings.
; ************************************************************************************
;
```

After this point, the assembly listing for the PIC16F872 chip is identical to that for the PIC16F882 chip.

Appendix "E1"

VisualBasic program for the Host computer – Main form

```
Option Strict On
Option Explicit On

Imports System
Imports System.Windows.Forms
Imports System.ComponentModel
Imports System.Threading
Imports System.IO.Ports

Public Class Main
    Inherits System.Windows.Forms.Form

    Public Sub New()
        ' Set parameters of the screen and the display
        Name = "Main"
        Text = "PIC_Programmer"
        FormBorderStyle = System.Windows.Forms.FormBorderStyle.Fixed3D
        Size = New Drawing.Size(My.Computer.Screen.Bounds.Width, My.Computer.Screen.Bounds.Height)
        CenterToScreen()
        MinimizeBox = True
        MaximizeBox = True
        Me.Refresh()
    End Sub

    Private Sub Main_Load() Handles Me.Load
        ' This subroutine runs automatically when the form is loaded
        ' Detect all serial ports on the computer
        ListOfSerialPorts = IO.Ports.SerialPort.GetPortNames()
        ' List the serial ports in the comboboxCommPorts control
        For I As Int32 = 0 To UBound(ListOfSerialPorts) Step 1
            comboboxCommPort.Items.Add(ListOfSerialPorts(I).ToString)
        Next I
        ' Take the first item in the list as the default value
        comboboxCommPort.SelectedIndex = 0
        comboboxCommPort.Text = comboboxCommPort.Items.Item(0).ToString
        ' Load the parameters for all Microchip devices
        SetDeviceParameters()
        ' List all Microchip devices in the comboboxDevices control
        For I As Int32 = 1 To NumOfDevices Step 1
            comboboxDevice.Items.Add(DeviceNames(I))
        Next I
        ' Take the first item in the list as the default value
        comboboxDevice.SelectedIndex = 0
        comboboxDevice.Text = comboboxDevice.Items.Item(0).ToString
        ' Set the buffer status flags
        OutputBuffersHaveBeenFilledFromHexFile = False
        InputBuffersHaveBeenFilledFromPIC = False
        ' Prompt the user to turn the programmer off
        labelStatusLine.Text = ""
        labelPromptLine.Text = "Ensure the programmer is turned OFF, and then click on Continue."
        labelErrorLine.Text = ""
        buttonContinue_1.Visible = True
        Me.Refresh()
```

```
End Sub

' ************************************************************************************
' *** Controls ***********************************************************************

Private labelStatus As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(75, 25), .Location = New Drawing.Point(5, 10),
    .Text = "Status:", .TextAlign = ContentAlignment.MiddleLeft,
    .Visible = True, .Parent = Me}

Public labelStatusLine As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(420, 25), .Location = New Drawing.Point(85, 10),
    .Text = "", .TextAlign = ContentAlignment.MiddleLeft, .BackColor = Color.White,
    .BorderStyle = BorderStyle.FixedSingle, .Visible = True, .Parent = Me}

Private labelPrompt As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(75, 25), .Location = New Drawing.Point(5, 40),
    .Text = "Prompt:", .TextAlign = ContentAlignment.MiddleLeft,
    .Visible = True, .Parent = Me}

Public labelPromptLine As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(420, 25), .Location = New Drawing.Point(85, 40),
    .Text = "", .TextAlign = ContentAlignment.MiddleLeft, .BackColor = Color.White,
    .BorderStyle = BorderStyle.FixedSingle, .Visible = True, .Parent = Me}

Private labelError As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(75, 25), .Location = New Drawing.Point(5, 70),
    .Text = "Error:", .TextAlign = ContentAlignment.MiddleLeft,
    .Visible = True, .Parent = Me}

Public labelErrorLine As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(420, 25), .Location = New Drawing.Point(85, 70),
    .Text = "", .TextAlign = ContentAlignment.MiddleLeft, .BackColor = Color.White,
    .BorderStyle = BorderStyle.FixedSingle, .Visible = True, .Parent = Me}

Private WithEvents buttonExit As New System.Windows.Forms.Button With
    {.Size = New Drawing.Size(75, 25), .Location = New Drawing.Point(430, 100),
    .Text = "Exit", .TextAlign = ContentAlignment.MiddleCenter,
    .Visible = True, .Parent = Me}

Public WithEvents buttonContinue_1 As New System.Windows.Forms.Button With
    {.Size = New Drawing.Size(75, 25), .Location = New Drawing.Point(85, 100),
    .Text = "Continue", .TextAlign = ContentAlignment.MiddleCenter,
    .Visible = False, .Parent = Me}

Private labelCommPort As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(75, 25), .Location = New Drawing.Point(5, 100),
    .Text = "Comm Port:", .TextAlign = ContentAlignment.MiddleLeft,
    .BorderStyle = BorderStyle.None, .Visible = False, .Parent = Me}

Public comboboxCommPort As New System.Windows.Forms.ComboBox With
    {.Size = New Drawing.Size(75, 25), .Location = New Drawing.Point(85, 100 + 2),
    .Text = "", .Visible = False, .Parent = Me}

Private WithEvents buttonContinue_2 As New System.Windows.Forms.Button With
    {.Size = New Drawing.Size(75, 25), .Location = New Drawing.Point(180, 100),
    .Text = "Continue", .TextAlign = ContentAlignment.MiddleCenter,
    .Visible = False, .Parent = Me}
```

```
Public WithEvents buttonContinue_3 As New System.Windows.Forms.Button With
    {.Size = New Drawing.Size(75, 25), .Location = New Drawing.Point(85, 100),
    .Text = "Continue", .TextAlign = ContentAlignment.MiddleCenter,
    .Visible = False, .Parent = Me}

Private labelDevice As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(75, 25), .Location = New Drawing.Point(5, 100),
    .Text = "Device:", .TextAlign = ContentAlignment.MiddleLeft,
    .BorderStyle = BorderStyle.None, .Visible = False, .Parent = Me}

Public comboboxDevice As New System.Windows.Forms.ComboBox With
    {.Size = New Drawing.Size(75, 25), .Location = New Drawing.Point(85, 100 + 2),
    .Text = "", .Visible = False, .Parent = Me}

Public WithEvents buttonContinue_4 As New System.Windows.Forms.Button With
    {.Size = New Drawing.Size(75, 25), .Location = New Drawing.Point(185, 100),
    .Text = "Continue", .TextAlign = ContentAlignment.MiddleCenter,
    .Visible = False, .Parent = Me}

Public WithEvents buttonBurnPIC As New System.Windows.Forms.Button With
    {.Size = New Drawing.Size(200, 25), .Location = New Drawing.Point(85, 100),
    .Text = "Burn the PIC from a hex file", .TextAlign = ContentAlignment.MiddleCenter,
    .Visible = False, .Parent = Me}

Public WithEvents buttonReadPIC As New System.Windows.Forms.Button With
    {.Size = New Drawing.Size(200, 25), .Location = New Drawing.Point(85, 130),
    .Text = "Read the PIC into input buffers", .TextAlign = ContentAlignment.MiddleCenter,
    .Visible = False, .Parent = Me}

Public WithEvents buttonLoadHexFile As New System.Windows.Forms.Button With
    {.Size = New Drawing.Size(200, 25), .Location = New Drawing.Point(85, 160),
    .Text = "Load hex file into output buffers", .TextAlign = ContentAlignment.MiddleCenter,
    .Visible = False, .Parent = Me}

Public WithEvents buttonCompareBuffers As New System.Windows.Forms.Button With
    {.Size = New Drawing.Size(200, 25), .Location = New Drawing.Point(85, 190),
    .Text = "Compare input and output buffers", .TextAlign = ContentAlignment.MiddleCenter,
    .Visible = False, .Parent = Me}

Public WithEvents buttonDisplayHexFile As New System.Windows.Forms.Button With
    {.Size = New Drawing.Size(200, 25), .Location = New Drawing.Point(85, 220),
    .Text = "Display the hex file", .TextAlign = ContentAlignment.MiddleCenter,
    .Visible = False, .Parent = Me}

Public WithEvents buttonDisplayOutputBuffers As New System.Windows.Forms.Button With
    {.Size = New Drawing.Size(200, 25), .Location = New Drawing.Point(85, 250),
    .Text = "Display the output buffers", .TextAlign = ContentAlignment.MiddleCenter,
    .Visible = False, .Parent = Me}

Public WithEvents buttonDisplayInputBuffers As New System.Windows.Forms.Button With
    {.Size = New Drawing.Size(200, 25), .Location = New Drawing.Point(85, 280),
    .Text = "Display the input buffers", .TextAlign = ContentAlignment.MiddleCenter,
    .Visible = False, .Parent = Me}

Public WithEvents buttonBurnConfigWords As New System.Windows.Forms.Button With
    {.Size = New Drawing.Size(200, 25), .Location = New Drawing.Point(85, 310),
    .Text = "Burn default Config Word(s)", .TextAlign = ContentAlignment.MiddleCenter,
```

```
    .Visible = False, .Parent = Me}

Public labelContents As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(500, 25), .Location = New Drawing.Point(5, 340),
    .Text = "", .TextAlign = ContentAlignment.MiddleLeft,
    .Visible = False, .Parent = Me}

Public textboxContents As New System.Windows.Forms.TextBox With
    {.Size = New Drawing.Size(500, 200), .Location = New Drawing.Point(5, 370),
    .Text = "", .TextAlign = HorizontalAlignment.Left, .Multiline = True,
    .ScrollBars = ScrollBars.Vertical, .Font = New Drawing.Font("Arial", 12),
    .BorderStyle = BorderStyle.FixedSingle, .Visible = False, .Parent = Me}

' ***********************************************************************************
' *** Control handlers **************************************************************

Public Sub buttonExit_Click() Handles buttonExit.MouseClick
    ' Close the serial port
    Try
        SerialPortToPIC.Close()
    Catch
    End Try
    Application.Exit()
End Sub

Private Sub buttonContinue_1_Click() Handles buttonContinue_1.MouseClick
    ' Blank out all controls
    labelStatusLine.Text = ""
    labelPromptLine.Text = ""
    labelErrorLine.Text = ""
    buttonContinue_1.Visible = False
    ' Prompt the user to select the programmer's communication port
    labelPromptLine.Text = "Select the PIC programmer's port, and then click on Continue."
    labelCommPort.Visible = True
    comboboxCommPort.Visible = True
    buttonContinue_2.Visible = True
    Me.Refresh()
End Sub

Private Sub buttonContinue_2_Click() Handles buttonContinue_2.MouseClick
    Dim TestByte As Int32
    ' Blank out all controls
    labelStatusLine.Text = ""
    labelPromptLine.Text = ""
    labelErrorLine.Text = ""
    labelCommPort.Visible = False
    comboboxCommPort.Visible = False
    buttonContinue_2.Visible = False
    ' Instantiate a serial port
    SerialPortToPIC = New SerialPort
    ' Store the name of the selected serial port
    SerialPortToPIC.PortName = comboboxCommPort.Text
    ' Open the selected serial port
    SerialPortToPIC.Open()
    ' Set the communication properties
    SerialPortToPIC.BaudRate = 9600
    SerialPortToPIC.DataBits = 8
    SerialPortToPIC.StopBits = StopBits.One
```

```
        SerialPortToPIC.Parity = Parity.None
        SerialPortToPIC.Handshake = Handshake.None
        SerialPortToPIC.ReceivedBytesThreshold = 1
        SerialPortToPIC.ReadTimeout = SerialPort.InfiniteTimeout
        ' Prompt the user to turn the programmer on
        labelPromptLine.Text = "Now, turn the programmer ON, and then click on Continue."
        buttonContinue_3.Visible = True
        buttonContinue_3.Enabled = False
        Me.Refresh()
        ' Wait for the programmer to be turned on.  Among other things, turning on the
        ' programmer will activate the 13.6V supply, which is connected directly to the
        ' ClearToSend line of the DB9.  The following code samples the voltage on the
        ' CTS line, and proceeds only after the 13.6V is detected.
        labelStatusLine.Text = "Waiting for the programmer to be turned on ..."
        labelStatusLine.Refresh()
        Do
            If (SerialPortToPIC.CtsHolding = True) Then
                Exit Do
            End If
        Loop
        ' Wait one second (time not critical) for the programmer to settle.  The principal delay
        ' when the programmer powers up is to turn the status LED on and hold it on for 500ms.
        labelStatusLine.Text = "Waiting for the programmer to settle ..."
        labelStatusLine.Refresh()
        labelPromptLine.Text = "Click on Continue."
        labelPromptLine.Refresh()
        Threading.Thread.Sleep(1000)
        ' Clear the ReceiveBuffer and the SendBuffer to remove any spurious bytes which may
        ' have been exchanged due to voltage transients as the programmer was powering up.
        SerialPortToPIC.DiscardOutBuffer()
        SerialPortToPIC.DiscardInBuffer()
        ' Exchange pings with the programmer.  This exchange uses the basic subroutines
        ' SendOneByteToProgrammer() and ReceiveOneByteFromProgrammer(), not the more complex
        ' multi-byte subroutines.
        labelStatusLine.Text = "Pinging the programmer ..."
        labelStatusLine.Refresh()
        SendOneByteToProgrammer(&H81)
        ReceiveOneByteFromProgrammer(TestByte)
        labelStatusLine.Text = "The byte received from the programmer was &&H" &
            ConvertIntegerTo8BitHexString(TestByte) & "."
        labelStatusLine.Refresh()
        If (TestByte <> &H55) Then
            labelErrorLine.Text = "The ping test failed."
            labelErrorLine.Refresh()
            MsgBox("This application will now close.")
            buttonExit_Click()
            Exit Sub
        End If
        labelStatusLine.Text = "The ping test was successful."
        buttonContinue_3.Enabled = True
        Me.Refresh()
    End Sub

    Private Sub buttonContinue_3_Click() Handles buttonContinue_3.MouseClick
        ' Blank out all controls
        labelStatusLine.Text = ""
        labelPromptLine.Text = ""
        labelErrorLine.Text = ""
```

```
        buttonContinue_3.Visible = False
        ' Prompt the user to select the device.
        labelPromptLine.Text = "Select the device to be programmed, and then click on Continue."
        labelErrorLine.Text = ""
        labelDevice.Visible = True
        comboboxDevice.Visible = True
        buttonContinue_4.Visible = True
        Me.Refresh()
    End Sub

    Private Sub buttonContinue_4_Click() Handles buttonContinue_4.MouseClick
        ' Blank out all controls
        labelStatusLine.Text = ""
        labelPromptLine.Text = ""
        labelErrorLine.Text = ""
        labelDevice.Visible = False
        comboboxDevice.Visible = False
        buttonContinue_4.Visible = False
        Me.Refresh()
        ' Load the parameters for the selected device
        IndexOfDevice = comboboxDevice.SelectedIndex + 1
        DeviceName = DeviceNames(IndexOfDevice)
        NumProgMemWords = NumsProgMemWords(IndexOfDevice)
        NumEEPROMBytes = NumsEEPROMBytes(IndexOfDevice)
        NumConfigWords = NumsConfigWords(IndexOfDevice)
        NumCalibWords = NumsCalibWords(IndexOfDevice)
        ' Prompt the user to select the appropriate action
        labelPromptLine.Text = "Click on the desired activity."
        buttonBurnPIC.Visible = True
        buttonReadPIC.Visible = True
        buttonLoadHexFile.Visible = True
        buttonCompareBuffers.Visible = True
        buttonDisplayHexFile.Visible = True
        buttonDisplayOutputBuffers.Visible = True
        buttonDisplayInputBuffers.Visible = True
        buttonBurnConfigWords.Visible = True
        Me.Refresh()
    End Sub

    Private Sub buttonBurnPIC_Click() Handles buttonBurnPIC.MouseClick
        Dim ReturnString As String = ""
        ' Blank out all controls
        labelStatusLine.Text = ""
        labelPromptLine.Text = ""
        labelErrorLine.Text = ""
        buttonBurnPIC.Visible = False
        buttonReadPIC.Visible = False
        buttonLoadHexFile.Visible = False
        buttonCompareBuffers.Visible = False
        buttonDisplayHexFile.Visible = False
        buttonDisplayOutputBuffers.Visible = False
        buttonDisplayInputBuffers.Visible = False
        buttonBurnConfigWords.Visible = False
        Me.Refresh()
        ' Prompt the user to select the hex file
        SelectOpenAndReadHexFile()
        ' Confirm that the file is in INHX8M format
        labelStatusLine.Text = "Checking the format of the hex file ..."
```

```
    Me.Refresh()
    ReturnString = ConfirmINHX8MHexFileFormat()
    If (ReturnString <> "SUCCESS") Then
        ' We are here if the hex file is in the wrong format
        labelStatusLine.Text = "The hex file has an incorrect format."
        labelPromptLine.Text = "Click on Exit."
        labelErrorLine.Text = ReturnString
        Me.Refresh()
        Exit Sub
    End If
    ' Create the output buffers
    labelStatusLine.Text = "Loading the hex file into the output buffers ..."
    Me.Refresh()
    ReturnString = ConstructOutputBuffersFromHexFile()
    If (ReturnString <> "SUCCESS") Then
        ' We are here if the hex file has a syntax error
        labelStatusLine.Text = "The hex file has an incorrect format."
        labelPromptLine.Text = "Click on Exit."
        labelErrorLine.Text = ReturnString
        Me.Refresh()
        Exit Sub
    End If
    ' Burn the PIC being programmed
    If ((DeviceName = "16F872") Or (DeviceName = "16F874") Or (DeviceName = "16F876")) Then
        BurnPIC16F87XFromOutputBuffers()
    Else
        If ((DeviceName = "16F882") Or (DeviceName = "16F884") Or (DeviceName = "16F886")) Then
            BurnPIC16F88XFromOutputBuffers()
        Else
            If (DeviceName = "16F872A") Then
                BurnPIC16F87XAFromOutputBuffers()
            Else
                MsgBox("Internal error: Device is not recognized.")
                buttonExit_Click()
                Exit Sub
            End If
        End If
    End If
    ' Show the user what was burned
    DisplayOutputBuffers()
    ' Prompt the user to select the appropriate action
    labelStatusLine.Text = "Programming is complete."
    labelPromptLine.Text = "Click on the desired activity."
    buttonBurnPIC.Visible = True
    buttonReadPIC.Visible = True
    buttonLoadHexFile.Visible = True
    buttonCompareBuffers.Visible = True
    buttonDisplayHexFile.Visible = True
    buttonDisplayOutputBuffers.Visible = True
    buttonDisplayInputBuffers.Visible = True
    buttonBurnConfigWords.Visible = True
    Me.Refresh()
End Sub

Private Sub buttonReadPIC_Click() Handles buttonReadPIC.MouseClick
    ' Blank out all controls
    labelStatusLine.Text = ""
    labelPromptLine.Text = ""
```

```
        labelErrorLine.Text = ""
        buttonBurnPIC.Visible = False
        buttonReadPIC.Visible = False
        buttonLoadHexFile.Visible = False
        buttonCompareBuffers.Visible = False
        buttonDisplayHexFile.Visible = False
        buttonDisplayOutputBuffers.Visible = False
        buttonDisplayInputBuffers.Visible = False
        buttonBurnConfigWords.Visible = False
        Me.Refresh()
        ' Read the PIC into the input buffers
        ReadPICIntoInputBuffers()
        ' Show the user the input buffers
        DisplayInputBuffers()
        ' Prompt the user to select the appropriate action
        labelPromptLine.Text = "Click on the desired activity."
        buttonBurnPIC.Visible = True
        buttonReadPIC.Visible = True
        buttonLoadHexFile.Visible = True
        buttonCompareBuffers.Visible = True
        buttonDisplayHexFile.Visible = True
        buttonDisplayOutputBuffers.Visible = True
        buttonDisplayInputBuffers.Visible = True
        buttonBurnConfigWords.Visible = True
        Me.Refresh()
    End Sub

    Private Sub LoadHexFile_Click() Handles buttonLoadHexFile.MouseClick
        Dim ReturnString As String
        ' Prompt the user to select the hex file
        SelectOpenAndReadHexFile()
        ' Confirm that the file is in INHX8M format
        labelStatusLine.Text = "Checking the format of the hex file ..."
        Me.Refresh()
        ReturnString = ConfirmINHX8MHexFileFormat()
        If (ReturnString <> "SUCCESS") Then
            ' We are here if the hex file is in the wrong format
            labelStatusLine.Text = "The hex file has an incorrect format."
            labelPromptLine.Text = "Click on Exit."
            labelErrorLine.Text = ReturnString
            Me.Refresh()
            Exit Sub
        End If
        ' Create the output buffers
        labelStatusLine.Text = "Loading the hex file into the output buffers ..."
        Me.Refresh()
        ReturnString = ConstructOutputBuffersFromHexFile()
        If (ReturnString <> "SUCCESS") Then
            ' We are here if the hex file has a syntax error
            labelStatusLine.Text = "The hex file has an incorrect format."
            labelPromptLine.Text = "Click on Exit."
            labelErrorLine.Text = ReturnString
            Me.Refresh()
            Exit Sub
        End If
        labelStatusLine.Text = "Load is complete."
        labelStatusLine.Refresh()
        ' Display the contents of the hex file in textboxContents
```

```
        DisplayHexFileContents()
        labelPromptLine.Text = "Click on the desired activity."
        labelPromptLine.Refresh()
    End Sub

    Private Sub buttonCompareBuffers_Click() Handles buttonCompareBuffers.MouseClick
        Dim ReturnString As String
        ' Blank out all controls
        labelStatusLine.Text = ""
        labelPromptLine.Text = ""
        labelErrorLine.Text = ""
        buttonBurnPIC.Visible = False
        buttonReadPIC.Visible = False
        buttonLoadHexFile.Visible = False
        buttonCompareBuffers.Visible = False
        buttonDisplayHexFile.Visible = False
        buttonDisplayOutputBuffers.Visible = False
        buttonDisplayInputBuffers.Visible = False
        buttonBurnConfigWords.Visible = False
        Me.Refresh()
        ' Ensure that the output buffers are valid
        If (OutputBuffersHaveBeenFilledFromHexFile = False) Then
            labelErrorLine.Text = "This PIC was not programmed during this session."
            labelPromptLine.Text = "Click on the desired activity."
            buttonBurnPIC.Visible = True
            buttonReadPIC.Visible = True
            buttonLoadHexFile.Visible = True
            buttonCompareBuffers.Visible = True
            buttonDisplayHexFile.Visible = True
            buttonDisplayOutputBuffers.Visible = True
            buttonDisplayInputBuffers.Visible = True
            buttonBurnConfigWords.Visible = True
            Me.Refresh()
            Exit Sub
        End If
        ' If necessary, read the PIC's contents into the input buffers
        If (InputBuffersHaveBeenFilledFromPIC = False) Then
            ReadPICIntoInputBuffers()
        End If
        ' Verify the contents
        ReturnString = CompareBuffers()
        If (ReturnString <> "SUCCESS") Then
            labelContents.Text = ""
            labelContents.Refresh()
            textboxContents.Text = "Verification error" & vbCrLf & ReturnString
            textboxContents.Refresh()
            buttonExit.Visible = True
        Else
            labelStatusLine.Text = "Verification was successful."
            labelPromptLine.Text = "Click on the desired activity."
            buttonBurnPIC.Visible = True
            buttonReadPIC.Visible = True
            buttonLoadHexFile.Visible = True
            buttonCompareBuffers.Visible = True
            buttonDisplayHexFile.Visible = True
            buttonDisplayOutputBuffers.Visible = True
            buttonDisplayInputBuffers.Visible = True
            buttonBurnConfigWords.Visible = True
```

```
            Me.Refresh()
        End If
    End Sub

    Private Sub buttonDisplayHexFile_Click() Handles buttonDisplayHexFile.MouseClick
        DisplayHexFileContents()
    End Sub

    Private Sub buttonDiplayOutputBuffers_Click() Handles buttonDisplayOutputBuffers.MouseClick
        DisplayOutputBuffers()
    End Sub

    Private Sub buttonDiplayInputBuffers_Click() Handles buttonDisplayInputBuffers.MouseClick
        DisplayInputBuffers()
    End Sub

    Private Sub buttonBurnConfigWords_Click() Handles buttonBurnConfigWords.MouseClick
        Dim ReturnString As String = ""
        ' Blank out all controls
        labelStatusLine.Text = ""
        labelPromptLine.Text = ""
        labelErrorLine.Text = ""
        buttonBurnPIC.Visible = False
        buttonReadPIC.Visible = False
        buttonLoadHexFile.Visible = False
        buttonCompareBuffers.Visible = False
        buttonDisplayHexFile.Visible = False
        buttonDisplayOutputBuffers.Visible = False
        buttonDisplayInputBuffers.Visible = False
        buttonBurnConfigWords.Visible = False
        Me.Refresh()
        ' Burn the default Configuration Word(s)
        If ((DeviceName = "16F872") Or (DeviceName = "16F874") Or (DeviceName = "16F876")) Then
            BurnPIC16F87XDefaultConfigWord()
        Else
            If ((DeviceName = "16F882") Or (DeviceName = "16F884") Or (DeviceName = "16F886")) Then
                BurnPIC16F88XDefaultConfigWords()
            Else
                If (DeviceName = "16F872A") Then
                    BurnPIC16F87XADefaultConfigWord()
                Else
                    MsgBox("Internal error: Device is not recognized.")
                    buttonExit_Click()
                    Exit Sub
                End If
            End If
            End If
        ' Prompt the user to select the appropriate action
        labelStatusLine.Text = "Burning is complete."
        labelPromptLine.Text = "Click on the desired activity."
        buttonBurnPIC.Visible = True
        buttonReadPIC.Visible = True
        buttonLoadHexFile.Visible = True
        buttonCompareBuffers.Visible = True
        buttonDisplayHexFile.Visible = True
        buttonDisplayOutputBuffers.Visible = True
        buttonDisplayInputBuffers.Visible = True
        buttonBurnConfigWords.Visible = True
```

```
        Me.Refresh()
    End Sub

End Class
```

Appendix "E2"

VisualBasic program for the Host computer – Module Variables.vb

```
Option Strict On
Option Explicit On
Imports System.IO.Ports

Public Module Variables

    ' The serial port which connects to the programmer
    Public WithEvents SerialPortToPIC As SerialPort

    ' ListOfSerialPorts holds the names of all serial ports on the computer
    Public ListOfSerialPorts() As String

    ' Microchip commands which the programmer should send to the PIC being programmed.
    ' Commands which apply to only one series of devices are identified with the name.
    Public Command_LoadConfiguration As Int32 = &H00             ' b'00000000'
    Public Command_BulkEraseSetup1_16F87X As Int32 = &H01        ' b'00000001'
    Public Command_LoadDataForProgramMemory As Int32 = &H02      ' b'00000010'
    Public Command_LoadDataForEEPROMMemory As Int32 = &H03       ' b'00000011'
    Public Command_ReadDataFromProgramMemory As Int32 = &H04     ' b'00000100'
    Public Command_ReadDataFromEEPROMMemory As Int32 = &H05      ' b'00000101'
    Public Command_IncrementAddress As Int32 = &H06              ' b'00000110'
    Public Command_BulkEraseSetup2_16F87X As Int32 = &H07        ' b'00000111'
    Public Command_BeginEraseProgrammingCycle As Int32 = &H08    ' b'00001000'
    Public Command_BulkEraseProgramMemory As Int32 = &H09        ' b'00001001'
    Public Command_EndProgramming_16F88X As Int32 = &H0A         ' b'00001010'
    Public Command_BulkEraseEEPROMMemory As Int32 = &H0B         ' b'00001011'
    Public Command_EndProgramming_16F87XA As Int32 = &H17        ' b'00010001'
    Public Command_BeginProgrammingOnlyCycle As Int32 = &H18     ' b'00010010'
    Public Command_ChipErase_16F87XA As Int32 = &H1F             ' b'00011111'

    ' Non-Microchip commands aimed at the programmer, not the PIC being programmed
    Public Command_RequestPing As Int32 = &H81                   ' b'10000001'
    Public Command_ResetPIC As Int32 = &H82                      ' b'10000010'

    ' Array to hold details about all of Microchip's devices of interest to the user
    Private Size1 As Int32 = 100                    ' Used only to declare vectors
    Public NumOfDevices As Int32                    ' Number of different devices tabulated
    Public DeviceNames(Size1) As String             ' Name of device
    Public ProgMemStartAddress As Int32 = &H0000
    Public NumsProgMemWords(Size1) As Int32         ' Number of 14-bit program memory words
    Public EEPROMStartAddress As Int32 = &H2100
    Public NumsEEPROMBytes(Size1) As Int32          ' Number of 8-bit EEPROM bytes
    Public UserIDStartAddress As Int32 = &H2000
    Public NumUserIDWords As Int32 = 4              ' Number of 14-bit UserID Words
    Public DeviceIDStartAddress As Int32 = &H2006
    Public NumDeviceIDWords As Int32 = 1            ' Number of 14-bit DeviceID Words
    Public ConfigWordsStartAddress As Int32 = &H2007
    Public NumsConfigWords(Size1) As Int32          ' Number of 14-bit Configuration Words
    Public CalibWordStartAddress As Int32 = &H2009
    Public NumsCalibWords(Size1) As Int32           ' Number of 14-bit Calibration Words

    ' Subroutine to set parameters for all of Microchip's devices of interest to the user
    Public Sub SetDeviceParameters()
```

```
    NumOfDevices = 0
    ' 16F872
    NumOfDevices = NumOfDevices + 1
    DeviceNames(NumOfDevices) = "16F872"
    NumsProgMemWords(NumOfDevices) = 2048
    NumsEEPROMBytes(NumOfDevices) = 64
    NumsConfigWords(NumOfDevices) = 1
    NumsCalibWords(NumOfDevices) = 0
    ' 16F874
    NumOfDevices = NumOfDevices + 1
    DeviceNames(NumOfDevices) = "16F874"
    NumsProgMemWords(NumOfDevices) = 4096
    NumsEEPROMBytes(NumOfDevices) = 128
    NumsConfigWords(NumOfDevices) = 1
    NumsCalibWords(NumOfDevices) = 0
    ' 16F876
    NumOfDevices = NumOfDevices + 1
    DeviceNames(NumOfDevices) = "16F876"
    NumsProgMemWords(NumOfDevices) = 8192
    NumsEEPROMBytes(NumOfDevices) = 256
    NumsConfigWords(NumOfDevices) = 1
    NumsCalibWords(NumOfDevices) = 0
    ' 16F872X
    NumOfDevices = NumOfDevices + 1
    DeviceNames(NumOfDevices) = "16F872A"
    NumsProgMemWords(NumOfDevices) = 2048
    NumsEEPROMBytes(NumOfDevices) = 64
    NumsConfigWords(NumOfDevices) = 1
    NumsCalibWords(NumOfDevices) = 0
    ' 16F882
    NumOfDevices = NumOfDevices + 1
    DeviceNames(NumOfDevices) = "16F882"
    NumsProgMemWords(NumOfDevices) = 2048
    NumsEEPROMBytes(NumOfDevices) = 128
    NumsConfigWords(NumOfDevices) = 2
    NumsCalibWords(NumOfDevices) = 1
    ' 16F884
    NumOfDevices = NumOfDevices + 1
    DeviceNames(NumOfDevices) = "16F884"
    NumsProgMemWords(NumOfDevices) = 4096
    NumsEEPROMBytes(NumOfDevices) = 256
    NumsConfigWords(NumOfDevices) = 2
    NumsCalibWords(NumOfDevices) = 1
    ' 16F886
    NumOfDevices = NumOfDevices + 1
    DeviceNames(NumOfDevices) = "16F886"
    NumsProgMemWords(NumOfDevices) = 8192
    NumsEEPROMBytes(NumOfDevices) = 256
    NumsConfigWords(NumOfDevices) = 2
    NumsCalibWords(NumOfDevices) = 1
End Sub

' Details about the particular device being programmed
Public IndexOfDevice As Int32
Public DeviceName As String
Public NumProgMemWords As Int32
Public NumEEPROMBytes As Int32
Public NumConfigWords As Int32
```

```
    Public NumCalibWords As Int32

    ' Default UserID Words
    Public DefaultUserID_1 As Int32 = &H004A    ' J
    Public DefaultUserID_2 As Int32 = &H0069    ' i
    Public DefaultUserID_3 As Int32 = &H006D    ' m
    Public DefaultUserID_4 As Int32 = &H0048    ' H

    ' Default Configuration Word(s)
    Public Default16F87XConfigWord As Int32 = &H3F3A
    Public Default16F88XConfigWord1 As Int32 = &H20F2
    Public Default16F88XConfigWord2 As Int32 = &H3FFF

    ' Output buffers for writing to PIC
    ' Each index contains one 14-bit word or one 8-bit byte, stored as a binary number.  There
    ' are separate buffers for the four types of memory.  The buffers are declared so they are
    ' big enough to hold 32K program memory words, 1K EEPROM data bytes, four UserID words one
    ' DeviceID word, two Configuration Words and one calibration word.  Each index in each
    ' buffer represents one address location in the PIC.  When the hex file is read, the
    ' instructions it contains are used to make an address-by-address representation of the
    ' PIC memory in these buffers.  In essence, the PIC is programmed by copying these buffers
    ' into the PIC's four memory types.
    Public OutputBufferProgMemory(32767) As Int32
    Public OutputBufferEEPROMMemory(1023) As Int32
    Public OutputBufferUserIDWords(3) As Int32
    Public OutputBufferDeviceIDWord As Int32
    Public OutputBufferConfigWords(1) As Int32
    Public OutputBufferCalibWord As Int32
    Public OutputBuffersHaveBeenFilledFromHexFile As Boolean

    ' Input buffers for reading from PIC
    ' Each index contains one 14-bit word or one 8-bit byte, stored as a binary number.  When
    ' PIC is read, the four types of its memory are copied into these buffers.
    Public InputBufferProgMemory(32767) As Int32
    Public InputBufferEEPROMMemory(1023) As Int32
    Public InputBufferUserIDWords(3) As Int32
    Public InputBufferDeviceIDWord As Int32
    Public InputBufferConfigWords(1) As Int32
    Public InputBufferCalibWord As Int32
    Public InputBuffersHaveBeenFilledFromPIC As Boolean

    ' Variables used in short-term timing
    Public StopwatchFrequency As Double = Stopwatch.Frequency
    Public TicksPerMicroSecond As Double = StopwatchFrequency / 1000000
    Public DelayInTicks As Double

End Module
```

Appendix "E3"

VisualBasic program for the Host computer – Module HexFileProcessing.vb

```
Option Strict On
Option Explicit On

Public Module HexFileProcessing

    ' INHX8M hex file contents
    ' Each index in HexFileContents() contains one ASCII characters.  14-bit words to be
    ' programmed into the PIC are stored as four consecutive ASCII characters, but in Low byte/
    ' High byte order.  A 150K hex file should be big enough to hold about 32K 14-bit words.
    Public HexFileContents(150000) As Int32
    Public HexFileName As String = ""
    Public HexFileLength As Int32 = 0

    ' ******************************************************************************************
    ' *** List of procedures *******************************************************************
    '
    ' A. Character manipulation routines
    '       Convert4BitHexInASCIIToInteger(Int32) As Int32
    '       ConvertIntegerTo16BitHexString(Int32) As String
    '       ConvertDecimaTo8BitHexString(Int32) As String
    '       ReverseBytesInHexString(String) As String
    ' B. Hex file management routines
    '       SelectOpenAndReadHexFile()
    '       ConfirmINHX8MHexFileFormat() As String
    '       ConstructOutputBuffersFromHexFile() As String

    '******************************************************************************************
    '*** A. Character manipulation routines
    '******************************************************************************************

    Public Function Convert4BitHexInASCIIToInteger(ByVal HexInASCII As Int32) As Int32
        ' This routine takes an integer which is the 4-bit ASCII representation of a hex
        ' character and returns the integer value.  This routine does not do any validation.
        If (Chr(HexInASCII) <= "9") Then
            Return HexInASCII - 48
        Else
            Return HexInASCII - 55
        End If
    End Function

    Public Function ConvertIntegerTo16BitHexString(ByVal Number As Int32) As String
        ' This routine takes a 16-bit integer and returns a 4-character string
        ' with the corresponding representation in hex.
        Dim Hex1, Hex2, Hex3, Hex4 As Int32
        Dim ReturnString As String
        Hex1 = CInt(Math.Floor(Number / 4096))
        Number = Number - (Hex1 * 4096)
        Hex2 = CInt(Math.Floor(Number / 256))
        Number = Number - (Hex2 * 256)
        Hex3 = CInt(Math.Floor(Number / 16))
        Hex4 = Number - (Hex3 * 16)
        If (Hex1 <= 9) Then
            ReturnString = Chr(Hex1 + 48)
```

```
    Else
        ReturnString = Chr(Hex1 + 55)
    End If
    If (Hex2 <= 9) Then
        ReturnString = ReturnString & Chr(Hex2 + 48)
    Else
        ReturnString = ReturnString & Chr(Hex2 + 55)
    End If
    If (Hex3 <= 9) Then
        ReturnString = ReturnString & Chr(Hex3 + 48)
    Else
        ReturnString = ReturnString & Chr(Hex3 + 55)
    End If
    If (Hex4 <= 9) Then
        ReturnString = ReturnString & Chr(Hex4 + 48)
    Else
        ReturnString = ReturnString & Chr(Hex4 + 55)
    End If
    Return ReturnString
End Function

Public Function ConvertIntegerTo8BitHexString(ByVal Number As Int32) As String
    ' This routine takes an 8-bit integer and returns a 2-character string
    ' with the corresponding representation in hex.
    Dim Hex1, Hex2 As Int32
    Dim ReturnString As String
    Hex1 = CInt(Math.Floor(Number / 16))
    Hex2 = Number - (Hex1 * 16)
    If (Hex1 <= 9) Then
        ReturnString = Chr(Hex1 + 48)
    Else
        ReturnString = Chr(Hex1 + 55)
    End If
    If (Hex2 <= 9) Then
        ReturnString = ReturnString & Chr(Hex2 + 48)
    Else
        ReturnString = ReturnString & Chr(Hex2 + 55)
    End If
    Return ReturnString
End Function

Public Function ReverseBytesInHexString(ByVal InString As String) As String
    ' This routine takes a 4-character hex string and reverses the order of
    ' the two bytes.  It is used during the display of the input and output
    ' buffers.
    Dim OutString As String
    If (Len(InString) <> 4) Then
        MsgBox("Internal error: A hex string does not contain four characters.")
        Main.buttonExit_Click()
        End
    End If
    OutString = Mid(InString, 3, 1) & Mid(InString, 4, 1) &
        Mid(InString, 1, 1) & Mid(InString, 2, 1)
    Return OutString
End Function

'*****************************************************************************
'*** B. Hex file management routines
```

```
'*******************************************************************************************

Public Sub SelectOpenAndReadHexFile()
    ' This subroutine guides the user through the process of selecting the hex file
    ' from the PC's directories.  The directory and file name are stored in variables
    ' MainDirectory and HexFileName, respectively.  This subroutine then opens the file
    ' and reads it into vector HexFileContents().
    Dim InputFileStream As IO.Stream = Nothing
    Dim OpenFileDialogue As New OpenFileDialog()
    OpenFileDialogue.Title = "Select hex file in the computer's file system"
    OpenFileDialogue.InitialDirectory = "C:\"
    OpenFileDialogue.Filter = "txt files (*.txt)|*.txt|hex files (*.hex)|*.hex"
    OpenFileDialogue.FilterIndex = 2
    OpenFileDialogue.RestoreDirectory = True
    If (OpenFileDialogue.ShowDialog() = System.Windows.Forms.DialogResult.OK) Then
        Try
            InputFileStream = OpenFileDialogue.OpenFile()
            If (InputFileStream IsNot Nothing) Then
                HexFileName = OpenFileDialogue.FileName
                HexFileLength = CInt(InputFileStream.Length)
                For I As Int32 = 0 To (HexFileLength - 1) Step 1
                    HexFileContents(I) = InputFileStream.ReadByte
                Next I
            End If
        Catch Ex As Exception
            MessageBox.Show("Fatal error: Cannot read file from disk." & vbCrLf & Ex.Message)
        Finally
            ' Close the file
            If (InputFileStream IsNot Nothing) Then
                InputFileStream.Close()
            End If
        End Try
    End If
End Sub

Public Function ConfirmINHX8MHexFileFormat() As String
    ' This routine returns "SUCCESS" if the hex file, which has been stored in vector
    ' HexFileContents(), is in INHX8M format.  Otherwise, it returns a string which
    ' describes the error.  Each line, or record, in an INHX8M file contains the following
    ' seven fields:
    ' FieldNum FieldName NumHexChars Description
    '     1    Start code     1      An ASCII colon ":"
    '     2    Byte count     2      Number of bytes (two hex chars each) in this record
    '     3    Address        4      Starting address of this record in PIC memory
    '     4    Record type    2      Type of this record (00, 01 or 04)
    '     5    Data           x      Each data word is two bytes, in High/Low format
    '     6    Checksum       2      Low byte of binary addition of fields 2 through 5
    '     7    End code       2      Chr(13) / Chr(10) = carriage return / line feed
    Dim Istart, Iend As Int32        ' Index counters in the hex file
    Dim StartCode As Int32
    Dim ByteCount As Int32
    Dim CharCount As Int32
    Dim RecordType As Int32
    Dim GivenCheckSum As Int32
    Dim EndCode As Int32
    Dim TempCheckSum As Int32
    Istart = 0
    Do
```

```
        ' Istart is the index of the start of a record in the hex file
        If (Istart >= HexFileLength) Then
            Exit Do
        End If
        ' Check the start code
        StartCode = HexFileContents(Istart)
        If (Chr(StartCode) <> ":") Then
            Return "Fatal error: The start code in a record is not ':'."
        End If
        ' Store the byte count
        ByteCount =
            (16 * Convert4BitHexInASCIIToInteger(HexFileContents(Istart + 1))) +
            Convert4BitHexInASCIIToInteger(HexFileContents(Istart + 2))
        CharCount = 2 * ByteCount
        ' Check the record type
        RecordType =
            (16 * Convert4BitHexInASCIIToInteger(HexFileContents(Istart + 7))) +
            Convert4BitHexInASCIIToInteger(HexFileContents(Istart + 8))
        If ((RecordType <> 0) And (RecordType <> 1) And (RecordType <> 4)) Then
            Return "Fatal error: Unrecognized record type: " & Trim(RecordType.ToString)
        End If
        ' Check the end code
        Iend = Istart + CharCount + 11
        EndCode = (16 * HexFileContents(Iend)) + HexFileContents(Iend + 1)
        If (EndCode <> 218) Then
            Return "Fatal error: The end code in a record is not VbCrLf."
        End If
        ' Store the given check sum
        GivenCheckSum =
            (16 * Convert4BitHexInASCIIToInteger(HexFileContents(Iend - 2))) +
            Convert4BitHexInASCIIToInteger(HexFileContents(Iend - 1))
        ' Calculate the check sum
        TempCheckSum = 0
        For I As Int32 = (Istart + 1) To (Iend - 4) Step 2
            TempCheckSum = TempCheckSum +
                (16 * Convert4BitHexInASCIIToInteger(HexFileContents(I))) +
                Convert4BitHexInASCIIToInteger(HexFileContents(I + 1))
        Next I
        ' Take the two's-complement
        TempCheckSum = (TempCheckSum Xor &HFFFFFFFF) + 1
        ' Keep only the low-order byte
        TempCheckSum = TempCheckSum And &H000000FF
        ' Test equality
        If (TempCheckSum <> GivenCheckSum) Then
            Return "Fatal error: A record has an incorrect check sum."
        End If
        'Set the index counter to the start of the next record
        Istart = Iend + 2
    Loop
    Return "SUCCESS"
End Function

Public Function ConstructOutputBuffersFromHexFile() As String
    ' This routine runs through the hex file, which has been stored in vector
    ' HexFileContents(), and builds buffers holding the various contents to be burned
    ' into the device.  While doing that, it ensures that the contents are consistent
    ' with the device being programmed.  This routine returns "SUCCESS" if the process
    ' succeeds.  Otherwise, it returns a string which describes the error.
```

```
'
' Step #1: Fill buffers with default values.  There are no default values for the
' DeviceID word or the Calibration Word, since these words are never burned into a PIC.
For I As Int32 = 1 To NumProgMemWords Step 1
    OutputBufferProgMemory(I - 1) = &H3FFF
Next I
For I As Int32 = 1 To NumEEPROMBytes Step 1
    OutputBufferEEPROMMemory(I - 1) = &HFF
Next I
OutputBufferUserIDWords(0) = DefaultUserID_1
OutputBufferUserIDWords(1) = DefaultUserID_2
OutputBufferUserIDWords(2) = DefaultUserID_3
OutputBufferUserIDWords(3) = DefaultUserID_4
If (Strings.Left(DeviceName, 5) = "16F87") Then
    OutputBufferConfigWords(0) = Default16F87XConfigWord
Else
    If (Strings.Left(DeviceName, 5) = "16F88") Then
        OutputBufferConfigWords(0) = Default16F88XConfigWord1
        OutputBufferConfigWords(1) = Default16F88XConfigWord2
    Else
        Return "Internal error: Device is not recognized."
        Exit Function
    End If
End If
' Step #2: Run through the hex file
Dim Istart, Iend As Int32       ' Index counters in the hex file
Dim ByteCount As Int32
Dim CharCount As Int32
Dim RecordType As Int32
Dim HexFileAddress, PICAddress As Int32
Dim MemoryClass As String
Dim ValueOfWord As Int32
Istart = 0
Do
    ' Istart is the index of the start of a record in the hex file
    If (Istart >= HexFileLength) Then
        Exit Do
    End If
    ' Store the byte count
    ByteCount =
        (16 * Convert4BitHexInASCIIToInteger(HexFileContents(Istart + 1))) +
        Convert4BitHexInASCIIToInteger(HexFileContents(Istart + 2))
    CharCount = 2 * ByteCount
    Iend = Istart + CharCount + 11
    ' Store the record type.
    RecordType =
        (16 * Convert4BitHexInASCIIToInteger(HexFileContents(Istart + 7))) +
        Convert4BitHexInASCIIToInteger(HexFileContents(Istart + 8))
    If (RecordType <> 0) Then
        ' Ignore all records except RecordType=0
        Istart = Iend + 2
    Else
        ' Calculate the starting address for this record
        HexFileAddress =
            (4096 * Convert4BitHexInASCIIToInteger(HexFileContents(Istart + 3))) +
            (256 * Convert4BitHexInASCIIToInteger(HexFileContents(Istart + 4))) +
            (16 * Convert4BitHexInASCIIToInteger(HexFileContents(Istart + 5))) +
            (1 * Convert4BitHexInASCIIToInteger(HexFileContents(Istart + 6)))
```

```
PICAddress = CInt(HexFileAddress / 2)
' Determine the class of memory this record describes
MemoryClass = "Unknown"
If ((PICAddress >= ProgMemStartAddress) And
    (PICAddress <= ProgMemStartAddress + NumProgMemWords - 1)) Then
    MemoryClass = "ProgMem"
End If
If ((PICAddress >= EEPROMStartAddress) And
    (PICAddress <= EEPROMStartAddress + NumEEPROMBytes - 1)) Then
    MemoryClass = "EEPROM"
End If
If ((PICAddress >= UserIDStartAddress) And
    (PICAddress <= CalibWordStartAddress + NumCalibWords - 1)) Then
    MemoryClass = "ConfigMemory"
End If
If (MemoryClass = "Unknown") Then
    Return "Address &&H" & Trim(HexFileAddress.ToString) &
        " in the hex file does not fit into this device."
    Exit Function
End If
' Load the data into the appropriate buffer
If (MemoryClass = "ProgMem") Then
    For I As Int32 = (Istart + 9) To (Iend - 6) Step 4
        ' Note that the 14-bit words of program memory are stored in the hex
        ' file as four nibbles (two bytes) in Low byte / High byte order, so
        ' that the nibbles need to be re-ordered in order to calculate the
        ' integer value of the 14-bit word.
        ValueOfWord =
            (16 * Convert4BitHexInASCIIToInteger(HexFileContents(I))) +
            (1 * Convert4BitHexInASCIIToInteger(HexFileContents(I + 1))) +
            (4096 * Convert4BitHexInASCIIToInteger(HexFileContents(I + 2))) +
            (256 * Convert4BitHexInASCIIToInteger(HexFileContents(I + 3)))
        OutputBufferProgMemory(PICAddress - ProgMemStartAddress) = ValueOfWord
        PICAddress = PICAddress + 1
    Next I
End If
If (MemoryClass = "EEPROM") Then
    For I As Int32 = (Istart + 9) To (Iend - 6) Step 4
        ' Note that the EEPROM bytes are stored in the hex file as four nibbles
        ' (two bytes) with the second pair of nibbles identically zero.  The
        ' first pair of nibbles constitutes the EEPROM data byte.
        ValueOfWord =
            (16 * Convert4BitHexInASCIIToInteger(HexFileContents(I))) +
            (1 * Convert4BitHexInASCIIToInteger(HexFileContents(I + 1)))
        OutputBufferEEPROMMemory(PICAddress - EEPROMStartAddress) = ValueOfWord
        PICAddress = PICAddress + 1
    Next I
End If
If (MemoryClass = "ConfigMem") Then
    For I As Int32 = (Istart + 9) To (Iend - 6) Step 4
        ' Note that the 14-bit words of configuration memory are stored in the
        ' hex file as four nibbles (two bytes) in Low byte / High byte order,
        ' so that the nibbles need to be re-ordered in order to calculate the
        ' integer value of the 14-bit word.
        ValueOfWord =
            (16 * Convert4BitHexInASCIIToInteger(HexFileContents(I))) +
            (1 * Convert4BitHexInASCIIToInteger(HexFileContents(I + 1))) +
            (4096 * Convert4BitHexInASCIIToInteger(HexFileContents(I + 2))) +
```

```
                            (256 * Convert4BitHexInASCIIToInteger(HexFileContents(I + 3)))
                        If ((PICAddress >= UserIDStartAddress) And
                            (PICAddress <= UserIDStartAddress + NumUserIDWords - 1)) Then
                            OutputBufferUserIDWords(PICAddress - UserIDStartAddress) =
                                ValueOfWord
                        End If
                        If (PICAddress = DeviceIDStartAddress) Then
                            OutputBufferDeviceIDWord = ValueOfWord
                        End If
                        If ((PICAddress >= ConfigWordsStartAddress) And
                            (PICAddress <= ConfigWordsStartAddress + NumConfigWords - 1)) Then
                            OutputBufferConfigWords(PICAddress - ConfigWordsStartAddress) =
                                ValueOfWord
                        End If
                        If (PICAddress = CalibWordStartAddress) Then
                            OutputBufferCalibWord = ValueOfWord
                        End If
                        PICAddress = PICAddress + 1
                    Next I
                End If
                'Set the index counter to the start of the next record
                Istart = Iend + 2
            End If
        Loop
        OutputBuffersHaveBeenFilledFromHexFile = True
        Return "SUCCESS"
    End Function

End Module
```

Appendix "E4"

VisualBasic program for the Host computer – Module Procedures.vb

```
Option Strict On
Option Explicit On

' *******************************************************************************************
' *** List of procedures *******************************************************************
'
' A. RS-232 communication with the programmer
'       SendOneByteToProgrammer(Int32)
'       ReceiveOneByteFromProgrammer(Int32)
'       AwaitPingFromProgrammer() As Boolean
' B. Sending instructions to the programmer.  Note that these are not Microchip commands.
'    The programmer winds up each of these commands by sending an &H55 ping.
'       RequestPing()
'       ResetProgrammer()
' C. Top-level software routines
'       BurnPIC16F87XFromOutputBuffers()
'       BurnPIC16F87XAFromOutputBuffers()
'       BurnPIC16F88XFromOutputBuffers()
'       ReadPICIntoInputBuffers()
'       CompareBuffers()
'       BulkErasePIC16F87X()
'       BulkErasePIC16F87XA()
'       BulkErasePIC16F88X()
'       BurnPIC16F87XConfigWord()
'       BurnPIC16F87XAConfigWord()
'       BurnPIC16F88XConfigWords()
' D. Timing routines
'       FixedTimeDelay(Microseconds)

Public Module Procedures

    '*****************************************************************************************
    '*** A. RS-232 communication with the programmer
    '*****************************************************************************************

    Public Sub SendOneByteToProgrammer(ByVal lOutByte As Int32)
        ' This routine sends lOutByte to the programmer.  It does not do any processing of this
        ' byte.  The programmer must be smart enough to figure out what it is receiving.
        Dim OutByteArray(0) As Byte
        OutByteArray(0) = CByte(lOutByte And &HFF)
        ' Write the byte to the SendBuffer
        SerialPortToPIC.Write(OutByteArray, 0, 1)
    End Sub

    Public Sub ReceiveOneByteFromProgrammer(ByRef lInByte As Int32)
        ' This routine waits until it receives one byte from the programmer.  It does do any
        ' processing of this byte.  The calling routine must be smart enough to know that the
        ' programmer will be sending a single byte at this time.
        Do
            ' Wait until there is a byte in the serial port's ReadBuffer
            If (SerialPortToPIC.BytesToRead >= 1) Then
                lInByte = SerialPortToPIC.ReadByte
                Exit Sub
```

```
        End If
    Loop
End Sub

Public Sub AwaitPingFromProgrammer()
    ' This subroutine is a special case of ReceiveOneByteFromProgrammer().  It has been
    ' included because of the frequency at which this program waits for a ping from the
    ' programmer to signal that it has completed an operation.  If the ping byte &H55 is
    ' received, this subroutine returns and the calling routine continues.  If the byte
    ' received is not the ping byte, the application terminates with an error message.  If
    ' no byte is received, the applications hangs.
    Dim lByteReceived As Int32
    ReceiveOneByteFromProgrammer(lByteReceived)
    If (lByteReceived <> &H55) Then
        Main.labelErrorLine.Text =
            "Ping byte received is &&H" & ConvertIntegerTo8BitHexString(lByteReceived)
        Main.labelErrorLine.Refresh()
        MsgBox("This application will now close.")
        Main.buttonExit_Click()
        Exit Sub
    End If
End Sub

'*****************************************************************************************
'*** B. Sending instructions to the programmer
'*****************************************************************************************

Public Sub RequestPing()
    ' This routine requests a ping from the programmer
    Main.labelStatusLine.Text = "Now pinging the programmer ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_RequestPing)
    Main.labelStatusLine.Text = "Ping has been sent."
    Main.labelStatusLine.Refresh()
End Sub

Public Sub ResetProgrammer()
    ' This routine resets the PIC being programmed
    Main.labelStatusLine.Text = "Now resetting the PIC being programmed ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_ResetPIC)
    AwaitPingFromProgrammer()
    Main.labelStatusLine.Text = "Reset is complete."
    Main.labelStatusLine.Refresh()
End Sub

'*****************************************************************************************
'*** C. Top-level software routines
'*****************************************************************************************

Public Sub BurnPIC16F87XFromOutputBuffers()
    ' This routine should only be called after the output buffers have been filled.  This
    ' routine does not care how the output buffers were filled, only that they exist.  This
    ' routine burns all addresses in the memory, including blocks of default data.
    ' Bulk erase the PIC being programmed
    BulkErasePIC16F87X()
    ' Burn the default Configuration Word
    BurnPIC16F87XDefaultConfigWord()
```

```
' Reset the PIC to reset the address pointer to &H0000
SendOneByteToProgrammer(Command_ResetPIC)
AwaitPingFromProgrammer()
' Burn the program memory
For I As Int32 = 0 To (NumProgMemWords - 1) Step 1
    Main.labelStatusLine.Text = "Now burning program memory address &&H" &
        ConvertIntegerTo16BitHexString(I) & " ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_LoadDataForProgramMemory)
    FixedTimeDelay(10)      ' Wait 10us before sending first data byte
    SendOneByteToProgrammer(OutputBufferProgMemory(I) And &HFF)
    FixedTimeDelay(10)      ' Wait 10us before sending second data byte
    SendOneByteToProgrammer(CInt((OutputBufferProgMemory(I) And &H3F00) / 256))
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
    AwaitPingFromProgrammer()
    FixedTimeDelay(5000)    ' Round tprog1 up from 1ms (min) to 5ms
    SendOneByteToProgrammer(Command_IncrementAddress)
    AwaitPingFromProgrammer()
Next I
' Burn the EEPROM memory, if applicable
If (NumEEPROMBytes > 0) Then
    For I As Int32 = 0 To (NumEEPROMBytes - 1) Step 1
        Main.labelStatusLine.Text = "Now burning EEPROM data address &&H" &
            ConvertIntegerTo16BitHexString(EEPROMStartAddress + I) & " ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_LoadDataForEEPROMMemory)
        FixedTimeDelay(10)      ' Wait 10us before sending data byte
        SendOneByteToProgrammer(OutputBufferEEPROMMemory(I) And &HFF)
        FixedTimeDelay(10)      ' Wait 10us before sending dummy data byte
        SendOneByteToProgrammer(&H00)
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
        AwaitPingFromProgrammer()
        FixedTimeDelay(5000)    ' Round tprog1 up from 1ms (min) to 5ms
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
    Next I
End If
' Relocate the address pointer to the start of configuration memory
Main.labelStatusLine.Text = "Now re-locating to configuration memory ..."
Main.labelStatusLine.Refresh()
SendOneByteToProgrammer(Command_LoadConfiguration)
FixedTimeDelay(10)      ' Wait 10us before sending first dummy data byte
SendOneByteToProgrammer(&H00)
FixedTimeDelay(10)      ' Wait 10us before sending second dummy data byte
SendOneByteToProgrammer(&H00)
AwaitPingFromProgrammer()
' Burn UserID words
For I As Int32 = 0 To (NumUserIDWords - 1) Step 1
    Main.labelStatusLine.Text = "Now burning UserID address &&H" &
        ConvertIntegerTo16BitHexString(UserIDStartAddress + I) & " ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_LoadDataForProgramMemory)
    FixedTimeDelay(10)      ' Wait 10us before sending first data byte
    SendOneByteToProgrammer(OutputBufferUserIDWords(I) And &HFF)
    FixedTimeDelay(10)      ' Wait 10us before sending second data byte
    SendOneByteToProgrammer(CInt((OutputBufferUserIDWords(I) And &H3F00) / 256))
```

```
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
        AwaitPingFromProgrammer()
        FixedTimeDelay(5000)    ' Round tprog1 up from 1ms (min) to 5ms
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
    Next I
    ' Increment up to the Configuration Word.  Note that the DeviceID word cannot be
    ' over-written.
    Main.labelStatusLine.Text = "Now skipping in configuration memory ..."
    Main.labelStatusLine.Refresh()
    Dim NumSkips As Int32
    NumSkips = ConfigWordsStartAddress - (UserIDStartAddress + NumUserIDWords)
    For I As Int32 = 1 To NumSkips Step 1
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
    Next I
    ' Burn the Configuration Word
    Main.labelStatusLine.Text = "Now burning Configuration Word address &&H" &
         ConvertIntegerTo16BitHexString(ConfigWordsStartAddress) & " ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_LoadDataForProgramMemory)
    FixedTimeDelay(10)      ' Wait 10us before sending first data byte
    SendOneByteToProgrammer(OutputBufferConfigWords(0) And &HFF)
    FixedTimeDelay(10)      ' Wait 10us before sending second data byte
    SendOneByteToProgrammer(CInt((OutputBufferConfigWords(0) And &H3F00) / 256))
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
    AwaitPingFromProgrammer()
    FixedTimeDelay(5000)    ' Round tprog1 up from 1ms (min) to 5ms
    SendOneByteToProgrammer(Command_IncrementAddress)
    AwaitPingFromProgrammer()
    ' Reset the PIC being programmed.  There is no Calibration Word.
    Main.labelStatusLine.Text = "Now resetting the PIC being programmed ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_ResetPIC)
    AwaitPingFromProgrammer()
    ' All finished
    Main.labelStatusLine.Text = "Burn is complete."
    Main.labelStatusLine.Refresh()
End Sub

Public Sub BurnPIC16F87XAFromOutputBuffers()
    ' This routine should only be called after the output buffers have been filled.  This
    ' routine does not care how the output buffers were filled, only that they exist.  This
    ' routine burns all addresses in the memory, including blocks of default data.
    ' Bulk erase the PIC being programmed
    BulkErasePIC16F87XA()
    ' Burn the default Configuration Word
    BurnPIC16F87XDefaultConfigWord()
    ' Reset the PIC to reset the address pointer to &H0000
    SendOneByteToProgrammer(Command_ResetPIC)
    AwaitPingFromProgrammer()
    ' Burn the program memory
    For I As Int32 = 0 To (NumProgMemWords - 1) Step 1
        Main.labelStatusLine.Text = "Now burning program memory address &&H" &
            ConvertIntegerTo16BitHexString(I) & " ..."
        Main.labelStatusLine.Refresh()
```

```
    SendOneByteToProgrammer(Command_LoadDataForProgramMemory)
    FixedTimeDelay(10)      ' Wait 10us before sending first data byte
    SendOneByteToProgrammer(OutputBufferProgMemory(I) And &HFF)
    FixedTimeDelay(10)      ' Wait 10us before sending second data byte
    SendOneByteToProgrammer(CInt((OutputBufferProgMemory(I) And &H3F00) / 256))
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
    AwaitPingFromProgrammer()
    FixedTimeDelay(5000)    ' Round tprog1 up from 1ms (min) to 5ms
    SendOneByteToProgrammer(Command_EndProgramming_16F87XA)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_IncrementAddress)
    AwaitPingFromProgrammer()
Next I
' Burn the EEPROM memory, if applicable
If (NumEEPROMBytes > 0) Then
    For I As Int32 = 0 To (NumEEPROMBytes - 1) Step 1
        Main.labelStatusLine.Text = "Now burning EEPROM data address &&H" &
            ConvertIntegerTo16BitHexString(EEPROMStartAddress + I) & " ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_LoadDataForEEPROMMemory)
        FixedTimeDelay(10)      ' Wait 10us before sending data byte
        SendOneByteToProgrammer(OutputBufferEEPROMMemory(I) And &HFF)
        FixedTimeDelay(10)      ' Wait 10us before sending dummy data byte
        SendOneByteToProgrammer(&H00)
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
        AwaitPingFromProgrammer()
        FixedTimeDelay(5000)    ' Round tprog1 up from 1ms (min) to 5ms
        SendOneByteToProgrammer(Command_EndProgramming_16F87XA)
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
    Next I
End If
' Relocate the address pointer to the start of configuration memory
Main.labelStatusLine.Text = "Now re-locating to configuration memory ..."
Main.labelStatusLine.Refresh()
SendOneByteToProgrammer(Command_LoadConfiguration)
FixedTimeDelay(10)      ' Wait 10us before sending first dummy data byte
SendOneByteToProgrammer(&H00)
FixedTimeDelay(10)      ' Wait 10us before sending second dummy data byte
SendOneByteToProgrammer(&H00)
AwaitPingFromProgrammer()
' Burn UserID words
For I As Int32 = 0 To (NumUserIDWords - 1) Step 1
    Main.labelStatusLine.Text = "Now burning UserID address &&H" &
        ConvertIntegerTo16BitHexString(UserIDStartAddress + I) & " ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_LoadDataForProgramMemory)
    FixedTimeDelay(10)      ' Wait 10us before sending first data byte
    SendOneByteToProgrammer(OutputBufferUserIDWords(I) And &HFF)
    FixedTimeDelay(10)      ' Wait 10us before sending second data byte
    SendOneByteToProgrammer(CInt((OutputBufferUserIDWords(I) And &H3F00) / 256))
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
    AwaitPingFromProgrammer()
    FixedTimeDelay(5000)    ' Round tprog1 up from 1ms (min) to 5ms
```

```
            SendOneByteToProgrammer(Command_EndProgramming_16F87XA)
            AwaitPingFromProgrammer()
            SendOneByteToProgrammer(Command_IncrementAddress)
            AwaitPingFromProgrammer()
        Next I
        ' Increment up to the Configuration Word.  Note that the DeviceID word cannot be
        ' over-written.
        Main.labelStatusLine.Text = "Now skipping in configuration memory ..."
        Main.labelStatusLine.Refresh()
        Dim NumSkips As Int32
        NumSkips = ConfigWordsStartAddress - (UserIDStartAddress + NumUserIDWords)
        For I As Int32 = 1 To NumSkips Step 1
            SendOneByteToProgrammer(Command_IncrementAddress)
            AwaitPingFromProgrammer()
        Next I
        ' Burn the Configuration Word
        Main.labelStatusLine.Text = "Now burning Configuration Word address &&H" &
             ConvertIntegerTo16BitHexString(ConfigWordsStartAddress) & " ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_LoadDataForProgramMemory)
        FixedTimeDelay(10)       ' Wait 10us before sending first data byte
        SendOneByteToProgrammer(OutputBufferConfigWords(0) And &HFF)
        FixedTimeDelay(10)       ' Wait 10us before sending second data byte
        SendOneByteToProgrammer(CInt((OutputBufferConfigWords(0) And &H3F00) / 256))
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
        AwaitPingFromProgrammer()
        FixedTimeDelay(5000)     ' Round tprog1 up from 1ms (min) to 5ms
        SendOneByteToProgrammer(Command_EndProgramming_16F87XA)
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
        ' Reset the PIC being programmed.  There is no Calibration Word.
        Main.labelStatusLine.Text = "Now resetting the PIC being programmed ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_ResetPIC)
        AwaitPingFromProgrammer()
        ' All finished
        Main.labelStatusLine.Text = "Burn is complete."
        Main.labelStatusLine.Refresh()
    End Sub

    Public Sub BurnPIC16F88XFromOutputBuffers()
        ' This routine should only be called after the output buffers have been filled.  This
        ' routine does not care how the output buffers were filled, only that they exist.  This
        ' routine burns all addresses in the memory, including blocks of default data.
        ' Bulk erase the PIC being programmed
        BulkErasePIC16F88X()
        ' Burn the default Configuration Word
        BurnPIC16F88XDefaultConfigWords()
        ' Reset the PIC to reset the address pointer to &H0000
        SendOneByteToProgrammer(Command_ResetPIC)
        AwaitPingFromProgrammer()
        ' Burn the program memory
        For I As Int32 = 0 To (NumProgMemWords - 1) Step 1
            Main.labelStatusLine.Text = "Now burning program memory address &&H" &
                ConvertIntegerTo16BitHexString(I) & " ..."
            Main.labelStatusLine.Refresh()
```

```
    SendOneByteToProgrammer(Command_LoadDataForProgramMemory)
    FixedTimeDelay(10)      ' Wait 10us before sending first data byte
    SendOneByteToProgrammer(OutputBufferProgMemory(I) And &HFF)
    FixedTimeDelay(10)      ' Wait 10us before sending second data byte
    SendOneByteToProgrammer(CInt((OutputBufferProgMemory(I) And &H3F00) / 256))
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
    AwaitPingFromProgrammer()
    FixedTimeDelay(5000)    ' Round tprog2 up from 2.5ms (max) to 5ms
    SendOneByteToProgrammer(Command_EndProgramming_16F88X)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_IncrementAddress)
    AwaitPingFromProgrammer()
Next I
' Burn the EEPROM memory, if applicable
If (NumEEPROMBytes > 0) Then
    For I As Int32 = 0 To (NumEEPROMBytes - 1) Step 1
        Main.labelStatusLine.Text = "Now burning EEPROM data address &&H" &
            ConvertIntegerTo16BitHexString(EEPROMStartAddress + I) & " ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_LoadDataForEEPROMMemory)
        FixedTimeDelay(10)      ' Wait 10us before sending data byte
        SendOneByteToProgrammer(OutputBufferEEPROMMemory(I) And &HFF)
        FixedTimeDelay(10)      ' Wait 10us before sending dummy data byte
        SendOneByteToProgrammer(&H00)
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
        AwaitPingFromProgrammer()
        FixedTimeDelay(5000)    ' Round tprog2 up from 2.5ms (max) to 5ms
        SendOneByteToProgrammer(Command_EndProgramming_16F88X)
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
    Next I
End If
' Relocate the address pointer to the start of configuration memory
Main.labelStatusLine.Text = "Now re-locating to configuration memory ..."
Main.labelStatusLine.Refresh()
SendOneByteToProgrammer(Command_LoadConfiguration)
FixedTimeDelay(10)      ' Wait 10us before sending first dummy data byte
SendOneByteToProgrammer(&H00)
FixedTimeDelay(10)      ' Wait 10us before sending second dummy data byte
SendOneByteToProgrammer(&H00)
AwaitPingFromProgrammer()
' Burn UserID words
For I As Int32 = 0 To (NumUserIDWords - 1) Step 1
    Main.labelStatusLine.Text = "Now burning UserID address &&H" &
        ConvertIntegerTo16BitHexString(UserIDStartAddress + I) & " ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_LoadDataForProgramMemory)
    FixedTimeDelay(10)      ' Wait 10us before sending first data byte
    SendOneByteToProgrammer(OutputBufferUserIDWords(I) And &HFF)
    FixedTimeDelay(10)      ' Wait 10us before sending second data byte
    SendOneByteToProgrammer(CInt((OutputBufferUserIDWords(I) And &H3F00) / 256))
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
    AwaitPingFromProgrammer()
    FixedTimeDelay(5000)    ' Round tprog2 up from 2.5ms (max) to 5ms
```

```
        SendOneByteToProgrammer(Command_EndProgramming_16F88X)
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
    Next I
    ' Increment up to the Configuration Words.  Note that the DeviceID word cannot be
    ' over-written.
    Main.labelStatusLine.Text = "Now skipping in configuration memory ..."
    Main.labelStatusLine.Refresh()
    Dim NumSkips As Int32
    NumSkips = ConfigWordsStartAddress - (UserIDStartAddress + NumUserIDWords)
    For I As Int32 = 1 To NumSkips Step 1
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
    Next I
    ' Burn the Configuration Words
    For I As Int32 = 0 To (NumConfigWords - 1) Step 1
        Main.labelStatusLine.Text = "Now burning Configuration Word address &&H" &
            ConvertIntegerTo16BitHexString(ConfigWordsStartAddress + I) & " ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_LoadDataForProgramMemory)
        FixedTimeDelay(10)      ' Wait 10us before sending first data byte
        SendOneByteToProgrammer(OutputBufferConfigWords(I) And &HFF)
        FixedTimeDelay(10)      ' Wait 10us before sending second data byte
        SendOneByteToProgrammer(CInt((OutputBufferConfigWords(I) And &H3F00) / 256))
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
        AwaitPingFromProgrammer()
        FixedTimeDelay(5000)    ' Round tprog2 up from 2.5ms (max) to 5ms
        SendOneByteToProgrammer(Command_EndProgramming_16F88X)
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
    Next I
    ' Reset the PIC being programmed.  Do not overwrite the Calibration Word.
    Main.labelStatusLine.Text = "Now resetting the PIC being programmed ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_ResetPIC)
    AwaitPingFromProgrammer()
    ' All finished
    Main.labelStatusLine.Text = "Burn is complete."
    Main.labelStatusLine.Refresh()
End Sub

Public Sub ReadPICIntoInputBuffers()
    ' This routine reads the contents of the PIC into the input buffers.  This routine
    ' reads all addresses, including blocks which are filled with default data.
    Dim lLowByte, lHighByte As Int32
    Main.labelStatusLine.Text = "Now resetting the PIC being programmed ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_ResetPIC)
    AwaitPingFromProgrammer()
    ' Read the program memory
    For I As Int32 = 0 To (NumProgMemWords - 1) Step 1
        Main.labelStatusLine.Text = "Now reading program memory address &&H" &
            ConvertIntegerTo16BitHexString(I) & " ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_ReadDataFromProgramMemory)
```

```
    ReceiveOneByteFromProgrammer(lLowByte)
    ReceiveOneByteFromProgrammer(lHighByte)
    InputBufferProgMemory(I) = (lHighByte * 256) + lLowByte
    SendOneByteToProgrammer(Command_IncrementAddress)
    AwaitPingFromProgrammer()
Next I
' Read the EEPROM memory
If (NumEEPROMBytes > 0) Then
    For I As Int32 = 0 To (NumEEPROMBytes - 1) Step 1
        Main.labelStatusLine.Text = "Now reading EEPROM data address &&H" &
            ConvertIntegerTo16BitHexString(EEPROMStartAddress + I) & " ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_ReadDataFromEEPROMMemory)
        ReceiveOneByteFromProgrammer(lLowByte)
        ReceiveOneByteFromProgrammer(lHighByte)
        InputBufferEEPROMMemory(I) = lLowByte
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
    Next I
End If
' Relocate the address pointer to the start of configuration memory
Main.labelStatusLine.Text = "Now re-locating to configuration memory ..."
Main.labelStatusLine.Refresh()
SendOneByteToProgrammer(Command_LoadConfiguration)
FixedTimeDelay(10)      ' Wait 10us before sending first dummy data byte
SendOneByteToProgrammer(&H00)
FixedTimeDelay(10)      ' Wait 10us before sending second dummy data byte
SendOneByteToProgrammer(&H00)
AwaitPingFromProgrammer()
' Read UserID words
For I As Int32 = 0 To (NumUserIDWords - 1) Step 1
    Main.labelStatusLine.Text = "Now reading UserID address &&H" &
        ConvertIntegerTo16BitHexString(UserIDStartAddress + I) & " ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_ReadDataFromProgramMemory)
    ReceiveOneByteFromProgrammer(lLowByte)
    ReceiveOneByteFromProgrammer(lHighByte)
    InputBufferUserIDWords(I) = (lHighByte * 256) + lLowByte
    SendOneByteToProgrammer(Command_IncrementAddress)
    AwaitPingFromProgrammer()
Next I
' Increment up to the DeviceID word
Main.labelStatusLine.Text = "Now skipping in configuration memory ..."
Main.labelStatusLine.Refresh()
Dim NumSkips As Int32
NumSkips = DeviceIDStartAddress - (UserIDStartAddress + NumUserIDWords)
For I As Int32 = 1 To NumSkips Step 1
    SendOneByteToProgrammer(Command_IncrementAddress)
    AwaitPingFromProgrammer()
Next I
' Read DeviceID word (only one)
Main.labelStatusLine.Text = "Now reading DeviceID address &&H" &
    ConvertIntegerTo16BitHexString(DeviceIDStartAddress) & " ..."
Main.labelStatusLine.Refresh()
SendOneByteToProgrammer(Command_ReadDataFromProgramMemory)
ReceiveOneByteFromProgrammer(lLowByte)
ReceiveOneByteFromProgrammer(lHighByte)
InputBufferDeviceIDWord = (lHighByte * 256) + lLowByte
```

```
    SendOneByteToProgrammer(Command_IncrementAddress)
    AwaitPingFromProgrammer()
    ' Read Configuration Word(s)
    For I As Int32 = 0 To (NumConfigWords - 1) Step 1
        Main.labelStatusLine.Text = "Now reading Configuration Word address &&H" &
            ConvertIntegerTo16BitHexString(ConfigWordsStartAddress + I) & " ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_ReadDataFromProgramMemory)
        ReceiveOneByteFromProgrammer(lLowByte)
        ReceiveOneByteFromProgrammer(lHighByte)
        InputBufferConfigWords(I) = (lHighByte * 256) + lLowByte
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
    Next I
    ' Read Calibration Word
    If (NumCalibWords = 1) Then
        Main.labelStatusLine.Text = "Now reading Calibration Word address &&H" &
            ConvertIntegerTo16BitHexString(CalibWordStartAddress) & " ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_ReadDataFromProgramMemory)
        ReceiveOneByteFromProgrammer(lLowByte)
        ReceiveOneByteFromProgrammer(lHighByte)
        InputBufferCalibWord = (lHighByte * 256) + lLowByte
    End If
    Main.labelStatusLine.Text = "Read is complete."
    Main.labelStatusLine.Refresh()
    InputBuffersHaveBeenFilledFromPIC = True
End Sub

Public Function CompareBuffers() As String
    ' This routine compares the output buffers with the input buffers.  It assumes that
    ' both sets of buffers have already been filled with valid data.  If the buffers are
    ' identical at all addresses below the maximums for the device being programmed, this
    ' routine returns "SUCCESS".  Otherwise, it returns a description of the first
    ' difference it detected between them.
    Dim ReturnString As String
    ' Verify program memory
    For I As Int32 = 0 To (NumProgMemWords - 1) Step 1
        If (OutputBufferProgMemory(I) <> InputBufferProgMemory(I)) Then
            ReturnString =
                "Mismatch in program memory at address &H" &
                ConvertIntegerTo16BitHexString(ProgMemStartAddress + I) & vbCrLf &
                "Output buffer = &H" &
                ConvertIntegerTo16BitHexString(OutputBufferProgMemory(I)) & vbCrLf &
                "Input buffer = &H" &
                ConvertIntegerTo16BitHexString(InputBufferProgMemory(I))
            Return ReturnString
            Exit Function
        End If
    Next I
    ' Verify EEPROM data
    For I As Int32 = 0 To (NumEEPROMBytes - 1) Step 1
        If (OutputBufferEEPROMMemory(I) <> InputBufferEEPROMMemory(I)) Then
            ReturnString =
                "Mismatch in EEPROM data at address &H" &
                ConvertIntegerTo16BitHexString(EEPROMStartAddress + I) & vbCrLf &
                "Output buffer = &H" &
                ConvertIntegerTo8BitHexString(OutputBufferEEPROMMemory(I)) & vbCrLf &
```

```
                "Input buffer = &H" &
                ConvertIntegerTo8BitHexString(InputBufferEEPROMMemory(I))
            Return ReturnString
            Exit Function
        End If
    Next I
    ' Verify UserID words
    For I As Int32 = 0 To (NumUserIDWords - 1) Step 1
        If (OutputBufferUserIDWords(I) <> InputBufferUserIDWords(I)) Then
            ReturnString =
                "Mismatch in UserID words at address &H" &
                ConvertIntegerTo16BitHexString(UserIDStartAddress + I) & vbCrLf &
                "Output buffer = &H" &
                ConvertIntegerTo16BitHexString(OutputBufferUserIDWords(I)) & vbCrLf &
                "Input buffer = &H" &
                ConvertIntegerTo16BitHexString(InputBufferUserIDWords(I))
            Return ReturnString
            Exit Function
        End If
    Next I
    ' *** Do NOT verify the DeviceID word, since the Burn process does not overwrite it. ***
    ' Verify Configuration Word(s)
    For I As Int32 = 0 To (NumConfigWords - 1) Step 1
        If (OutputBufferConfigWords(I) <> InputBufferConfigWords(I)) Then
            ReturnString =
                "Mismatch in Configuration Words at address &H" &
                ConvertIntegerTo16BitHexString(ConfigWordsStartAddress + I) & vbCrLf &
                "Output buffer = &H" &
                ConvertIntegerTo16BitHexString(OutputBufferConfigWords(I)) & vbCrLf &
                "Input buffer = &H" &
                ConvertIntegerTo16BitHexString(InputBufferConfigWords(I))
            Return ReturnString
            Exit Function
        End If
    Next I
    ' *** Do NOT verify the Calibration, since the Burn process does not overwrite it. ***
    Return "SUCCESS"
End Function

Public Sub BulkErasePIC16F87X()
    ' Reset the PIC being programmed, and set the address pointer to &H0000
    Main.labelStatusLine.Text = "Now bulk-erasing the PIC being programmed ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_ResetPIC)
    AwaitPingFromProgrammer()
    ' Bulk erase the program memory
    SendOneByteToProgrammer(Command_LoadDataForProgramMemory)
    FixedTimeDelay(10)           ' Wait 10us before sending first data byte
    SendOneByteToProgrammer(&HFF)
    FixedTimeDelay(10)           ' Wait 10us before sending second data byte
    SendOneByteToProgrammer(&H3F)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BulkEraseSetup1_16F87X)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BulkEraseSetup2_16F87X)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BeginEraseProgrammingCycle)
    AwaitPingFromProgrammer()
```

```
    FixedTimeDelay(20000)       ' Round up from 8ms (max) to 20ms
    SendOneByteToProgrammer(Command_BulkEraseSetup1_16F87X)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BulkEraseSetup2_16F87X)
    AwaitPingFromProgrammer()
    FixedTimeDelay(20000)       ' Round up from 8ms (max) to 20ms
    ' Bulk erase EEPROM memory
    SendOneByteToProgrammer(Command_LoadDataForEEPROMMemory)
    FixedTimeDelay(10)          ' Wait 10us before sending first data byte
    SendOneByteToProgrammer(&HFF)
    FixedTimeDelay(10)          ' Wait 10us before sending second data byte
    SendOneByteToProgrammer(&H3F)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BulkEraseSetup1_16F87X)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BulkEraseSetup2_16F87X)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BeginEraseProgrammingCycle)
    AwaitPingFromProgrammer()
    FixedTimeDelay(20000)       ' Round up from 8ms (max) to 20ms
    SendOneByteToProgrammer(Command_BulkEraseSetup1_16F87X)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BulkEraseSetup2_16F87X)
    AwaitPingFromProgrammer()
    FixedTimeDelay(20000)       ' Round up from 8ms (max) to 20ms
    ' Bulk erase configuration memory
    SendOneByteToProgrammer(Command_LoadConfiguration)
    FixedTimeDelay(10)          ' Wait 10us before sending first data byte
    SendOneByteToProgrammer(&HFF)
    FixedTimeDelay(10)          ' Wait 10us before sending second data byte
    SendOneByteToProgrammer(&H3F)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BulkEraseSetup1_16F87X)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BulkEraseSetup2_16F87X)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BeginEraseProgrammingCycle)
    AwaitPingFromProgrammer()
    FixedTimeDelay(20000)       ' Round up from 8ms (max) to 20ms
    SendOneByteToProgrammer(Command_BulkEraseSetup1_16F87X)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BulkEraseSetup2_16F87X)
    AwaitPingFromProgrammer()
    FixedTimeDelay(20000)       ' Round up from 8ms (max) to 20ms
End Sub

Public Sub BulkErasePIC16F87XA()
    ' Reset the PIC being programmed, and set the address pointer to &H0000
    Main.labelStatusLine.Text = "Now bulk-erasing the PIC being programmed ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_ResetPIC)
    AwaitPingFromProgrammer()
    ' Bulk erase all memories except the UserID words
    SendOneByteToProgrammer(Command_ChipErase_16F87XA)
    AwaitPingFromProgrammer()
    FixedTimeDelay(20000)       ' Round tprog2 up from 8ms (max) to 20ms
End Sub
```

```
Public Sub BulkErasePIC16F88X()
    ' Reset the PIC being programmed, and set the address pointer to &H0000
    Main.labelStatusLine.Text = "Now bulk-erasing the PIC being programmed ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_ResetPIC)
    AwaitPingFromProgrammer()
    ' A. Bulk erase the program memory.  Incidentally, this command also erases the
    '    Configuration Words.  Some authorities suggest that it also erases EEPROM
    '    memory, but I explicitly erase EEPROM memory in Step B following.
    SendOneByteToProgrammer(Command_BulkEraseProgramMemory)
    AwaitPingFromProgrammer()
    FixedTimeDelay(20000)       ' Round TERA up from 6ms (max) to 20ms
    ' B. Bulk erase the EEPROM memory
    SendOneByteToProgrammer(Command_BulkEraseEEPROMMemory)
    AwaitPingFromProgrammer()
    FixedTimeDelay(20000)       ' Round TERA up from 6ms (max) to 20ms
    ' C. Bulk erase the configuration memory.  This command erases the UserID Words
    '    and the Configuration Words, but it does not erase the DeviceID Word or the
    '    Calibration Word.
    SendOneByteToProgrammer(Command_LoadConfiguration)
    FixedTimeDelay(10)          ' Wait 10us before sending first dummy data byte
    SendOneByteToProgrammer(&H00)
    FixedTimeDelay(10)          ' Wait 10us before sending second dummy data byte
    SendOneByteToProgrammer(&H00)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BulkEraseProgramMemory)
    AwaitPingFromProgrammer()
    FixedTimeDelay(20000)       ' Round TERA up from 6ms (max) to 20ms
End Sub

Public Sub BurnPIC16F87XDefaultConfigWord()
    ' This routine can be called if a PIC to be programmed has a Configuration Word that
    ' requires Low-Voltage Programming.  This routine will write the default configuration,
    ' which allows for High-Voltage Programming.
    ' Reset the PIC being programmed, and set the address pointer to &H0000
    Main.labelStatusLine.Text = "Now resetting the PIC being programmed ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_ResetPIC)
    AwaitPingFromProgrammer()
    ' Relocate the address pointer to the start of configuration memory
    Main.labelStatusLine.Text = "Now re-locating to configuration memory ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_LoadConfiguration)
    FixedTimeDelay(10)      ' Wait 10us before sending first dummy data byte
    SendOneByteToProgrammer(&H00)
    FixedTimeDelay(10)      ' Wait 10us before sending second dummy data byte
    SendOneByteToProgrammer(&H00)
    AwaitPingFromProgrammer()
    ' Increment up to the Configuration Word.
    Main.labelStatusLine.Text = "Now skipping in configuration memory ..."
    Main.labelStatusLine.Refresh()
    Dim NumSkips As Int32
    NumSkips = ConfigWordsStartAddress - UserIDStartAddress
    For I As Int32 = 1 To NumSkips Step 1
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
    Next I
    ' Burn the default Configuration Word
```

```
        Main.labelStatusLine.Text = "Now burning Configuration Word address &&H" &
            ConvertIntegerTo16BitHexString(ConfigWordsStartAddress) & " ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_LoadDataForProgramMemory)
        FixedTimeDelay(10)      ' Wait 10us before sending first data byte
        SendOneByteToProgrammer(Default16F87XConfigWord And &HFF)
        FixedTimeDelay(10)      ' Wait 10us before sending second data byte
        SendOneByteToProgrammer(CInt((Default16F87XConfigWord And &H3F00) / 256))
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
        AwaitPingFromProgrammer()
        FixedTimeDelay(5000)    ' Round tprog1 up from 1ms (min) to 5ms
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
        ' Reset the PIC being programmed
        Main.labelStatusLine.Text = "Now resetting the PIC being programmed ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_ResetPIC)
        AwaitPingFromProgrammer()
        ' All finished
        Main.labelStatusLine.Text = "Burn is complete."
        Main.labelStatusLine.Refresh()
    End Sub

    Public Sub BurnPIC16F87XADefaultConfigWord()
        ' This routine can be called if a PIC to be programmed has a Configuration Word that
        ' requires Low-Voltage Programming.  This routine will write the default configuration,
        ' which allows for High-Voltage Programming.
        ' Reset the PIC being programmed, and set the address pointer to &H0000
        Main.labelStatusLine.Text = "Now resetting the PIC being programmed ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_ResetPIC)
        AwaitPingFromProgrammer()
        ' Relocate the address pointer to the start of configuration memory
        Main.labelStatusLine.Text = "Now re-locating to configuration memory ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_LoadConfiguration)
        FixedTimeDelay(10)      ' Wait 10us before sending first dummy data byte
        SendOneByteToProgrammer(&H00)
        FixedTimeDelay(10)      ' Wait 10us before sending second dummy data byte
        SendOneByteToProgrammer(&H00)
        AwaitPingFromProgrammer()
        ' Increment up to the Configuration Word.
        Main.labelStatusLine.Text = "Now skipping in configuration memory ..."
        Main.labelStatusLine.Refresh()
        Dim NumSkips As Int32
        NumSkips = ConfigWordsStartAddress - UserIDStartAddress
        For I As Int32 = 1 To NumSkips Step 1
            SendOneByteToProgrammer(Command_IncrementAddress)
            AwaitPingFromProgrammer()
        Next I
        ' Burn the default Configuration Word
        Main.labelStatusLine.Text = "Now burning Configuration Word address &&H" &
            ConvertIntegerTo16BitHexString(ConfigWordsStartAddress) & " ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_LoadDataForProgramMemory)
        FixedTimeDelay(10)      ' Wait 10us before sending first data byte
        SendOneByteToProgrammer(Default16F87XConfigWord And &HFF)
```

```
    FixedTimeDelay(10)      ' Wait 10us before sending second data byte
    SendOneByteToProgrammer(CInt((Default16F87XConfigWord And &H3F00) / 256))
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
    AwaitPingFromProgrammer()
    FixedTimeDelay(5000)    ' Round tprog1 up from 1ms (min) to 5ms
    SendOneByteToProgrammer(Command_EndProgramming_16F87XA)
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_IncrementAddress)
    AwaitPingFromProgrammer()
    ' Reset the PIC being programmed
    Main.labelStatusLine.Text = "Now resetting the PIC being programmed ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_ResetPIC)
    AwaitPingFromProgrammer()
    ' All finished
    Main.labelStatusLine.Text = "Burn is complete."
    Main.labelStatusLine.Refresh()
End Sub

Public Sub BurnPIC16F88XDefaultConfigWords()
    ' This routine can be called if a PIC to be programmed has Configuration Words that
    ' require Low-Voltage Programming.  This routine will write the default configuration,
    ' which allows for High-Voltage Programming.
    ' Reset the PIC being programmed, and set the address pointer to &H0000
    Main.labelStatusLine.Text = "Now resetting the PIC being programmed ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_ResetPIC)
    AwaitPingFromProgrammer()
    ' Relocate the address pointer to the start of configuration memory
    Main.labelStatusLine.Text = "Now re-locating to configuration memory ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_LoadConfiguration)
    FixedTimeDelay(10)      ' Wait 10us before sending first dummy data byte
    SendOneByteToProgrammer(&H00)
    FixedTimeDelay(10)      ' Wait 10us before sending second dummy data byte
    SendOneByteToProgrammer(&H00)
    AwaitPingFromProgrammer()
    ' Increment up to the Configuration Word.
    Main.labelStatusLine.Text = "Now skipping in configuration memory ..."
    Main.labelStatusLine.Refresh()
    Dim NumSkips As Int32
    NumSkips = ConfigWordsStartAddress - UserIDStartAddress
    For I As Int32 = 1 To NumSkips Step 1
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
    Next I
    ' Burn the default Configuration Words
    Main.labelStatusLine.Text = "Now burning Configuration Word address &&H" &
        ConvertIntegerTo16BitHexString(ConfigWordsStartAddress) & " ..."
    Main.labelStatusLine.Refresh()
    SendOneByteToProgrammer(Command_LoadDataForProgramMemory)
    FixedTimeDelay(10)      ' Wait 10us before sending first data byte
    SendOneByteToProgrammer(Default16F88XConfigWord1 And &HFF)
    FixedTimeDelay(10)      ' Wait 10us before sending second data byte
    SendOneByteToProgrammer(CInt((Default16F88XConfigWord1 And &H3F00) / 256))
    AwaitPingFromProgrammer()
    SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
```

```
        AwaitPingFromProgrammer()
        FixedTimeDelay(5000)    ' Round tprog1 up from 1ms (min) to 5ms
        SendOneByteToProgrammer(Command_EndProgramming_16F88X)
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_LoadDataForProgramMemory)
        FixedTimeDelay(10)      ' Wait 10us before sending first data byte
        SendOneByteToProgrammer(Default16F88XConfigWord2 And &HFF)
        FixedTimeDelay(10)      ' Wait 10us before sending second data byte
        SendOneByteToProgrammer(CInt((Default16F88XConfigWord2 And &H3F00) / 256))
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_BeginProgrammingOnlyCycle)
        AwaitPingFromProgrammer()
        FixedTimeDelay(5000)    ' Round tprog1 up from 1ms (min) to 5ms
        SendOneByteToProgrammer(Command_EndProgramming_16F88X)
        AwaitPingFromProgrammer()
        SendOneByteToProgrammer(Command_IncrementAddress)
        AwaitPingFromProgrammer()
        ' Reset the PIC being programmed
        Main.labelStatusLine.Text = "Now resetting the PIC being programmed ..."
        Main.labelStatusLine.Refresh()
        SendOneByteToProgrammer(Command_ResetPIC)
        AwaitPingFromProgrammer()
        ' All finished
        Main.labelStatusLine.Text = "Burn is complete."
        Main.labelStatusLine.Refresh()
    End Sub

    '*************************************************************************************
    '*** D. Timing routines
    '*************************************************************************************

    Public Sub FixedTimeDelay(ByVal MicroSeconds As Int32)
        ' This subroutine uses a stopwatch, which measures time in TICKs which are equal
        ' in duration to the reciprocal of the stopwatch's frequency.
        DelayInTicks = MicroSeconds * TicksPerMicroSecond
        Dim SW As Stopwatch = New Stopwatch
        SW.Start()
        Do
            If (SW.ElapsedTicks > DelayInTicks) Then
                Exit Do
            End If
        Loop
        SW.Stop()
    End Sub

End Module
```

Appendix "E5"

VisualBasic program for the Host computer – Module Display.vb

```
Option Strict On
Option Explicit On

Public Module Display

    ' ********************************************************************************
    ' *** List of display routines ***************************************************
    '       DisplayHexFileContents()
    '       DisplayOutputBuffers()
    '       DisplayInputBuffers()

    Public Sub DisplayHexFileContents()
        Dim DisplayString As String = ""
        If (OutputBuffersHaveBeenFilledFromHexFile = False) Then
            Main.labelErrorLine.Text = "A hex file has not been opened."
            Main.labelErrorLine.Refresh()
            Exit Sub
        End If
        Main.labelStatusLine.Text = "Now formatting hex file.  Please wait ..."
        Main.labelStatusLine.Refresh()
        Main.labelContents.Visible = True
        Main.labelContents.Text = "Contents of hex file in Low byte / High byte order:"
        Main.labelContents.Refresh()
        Main.textboxContents.Visible = True
        Main.textboxContents.Text = DisplayString
        Main.textboxContents.Refresh()
        For I As Int32 = 1 To HexFileLength Step 1
            DisplayString = DisplayString & Chr(HexFileContents(I - 1))
            Application.DoEvents()
        Next I
        Main.textboxContents.Text = DisplayString
        Main.textboxContents.Refresh()
        Main.labelStatusLine.Text = ""
        Main.labelStatusLine.Refresh()
    End Sub

    Public Sub DisplayOutputBuffers()
        Dim DisplayString As String = ""
        If (OutputBuffersHaveBeenFilledFromHexFile = False) Then
            Main.labelErrorLine.Text = "Output buffers have not been filled."
            Main.labelErrorLine.Refresh()
            Exit Sub
        End If
        Main.labelContents.Visible = True
        Main.labelContents.Text = "Contents of output buffers in High byte / Low byte order:"
        Main.labelContents.Refresh()
        Main.textboxContents.Visible = True
        ' Display program memory
        Main.labelStatusLine.Text = "Now formatting program memory.  Please wait ..."
        Main.labelStatusLine.Refresh()
        DisplayString = "Program memory" & vbCrLf
        Main.textboxContents.Text = DisplayString
        Main.textboxContents.Refresh()
```

```
    For I As Int32 = 0 To (NumProgMemWords - 1) Step 8
        DisplayString = DisplayString & "&H" &
            ConvertIntegerTo16BitHexString(ProgMemStartAddress + I) & "   "
        For J As Int32 = 0 To 7 Step 1
            DisplayString = DisplayString &
                ConvertIntegerTo16BitHexString(OutputBufferProgMemory(I + J)) & " "
        Next J
        DisplayString = DisplayString & vbCrLf
        Main.textboxContents.Text = DisplayString
        Main.textboxContents.Refresh()
        Application.DoEvents()
    Next I
    DisplayString = DisplayString & vbCrLf
    ' Display EEPROM memory
    Main.labelStatusLine.Text = "Now formatting EEPROM memory.  Please keep waiting ..."
    Main.labelStatusLine.Refresh()
    DisplayString = DisplayString & "EEPROM memory" & vbCrLf
    For I As Int32 = 0 To (NumEEPROMBytes - 1) Step 8
        DisplayString = DisplayString & "&H" &
            ConvertIntegerTo16BitHexString(EEPROMStartAddress + I) & "   "
        For J As Int32 = 0 To 7 Step 1
            DisplayString = DisplayString &
                ConvertIntegerTo8BitHexString(OutputBufferEEPROMMemory(I + J)) & " "
        Next J
        DisplayString = DisplayString & vbCrLf
        Main.textboxContents.Text = DisplayString
        Main.textboxContents.Refresh()
        Application.DoEvents()
    Next I
    DisplayString = DisplayString & vbCrLf
    'Display all four UserID words on one line
    DisplayString = DisplayString & "UserID words" & vbCrLf & "&H" &
        ConvertIntegerTo16BitHexString(UserIDStartAddress) & "   "
    For I As Int32 = 0 To (NumUserIDWords - 1) Step 1
        DisplayString = DisplayString &
            ConvertIntegerTo16BitHexString(OutputBufferUserIDWords(I)) & " "
    Next I
    DisplayString = DisplayString & vbCrLf & vbCrLf
    'Display the Configuration Word(s) on one line
    DisplayString = DisplayString & "Configuration Word(s)" &
        vbCrLf & "&H" & ConvertIntegerTo16BitHexString(ConfigWordsStartAddress) & "   "
    For I As Int32 = 0 To (NumConfigWords - 1) Step 1
        DisplayString = DisplayString &
            ConvertIntegerTo16BitHexString(OutputBufferConfigWords(I)) & " "
    Next I
    DisplayString = DisplayString & vbCrLf & vbCrLf
    ' All finished
    DisplayString = DisplayString & "End of output buffers"
    Main.textboxContents.Text = DisplayString
    Main.textboxContents.Refresh()
    Main.labelStatusLine.Text = ""
    Main.labelStatusLine.Refresh()
End Sub

Public Sub DisplayInputBuffers()
    Dim DisplayString As String = ""
    If (InputBuffersHaveBeenFilledFromPIC = False) Then
        Main.labelErrorLine.Text = "Input buffers have not been filled."
```

```
    Main.labelErrorLine.Refresh()
    Exit Sub
End If
Main.labelContents.Visible = True
Main.labelContents.Text = "Contents of input buffers in High byte / Low byte order:"
Main.labelContents.Refresh()
Main.textboxContents.Visible = True
' Display program memory
Main.labelStatusLine.Text = "Now formatting program memory.  Please wait ..."
Main.labelStatusLine.Refresh()
DisplayString = "Program memory" & vbCrLf
Main.textboxContents.Text = DisplayString
Main.textboxContents.Refresh()
For I As Int32 = 0 To (NumProgMemWords - 1) Step 8
    DisplayString = DisplayString & "&H" &
        ConvertIntegerTo16BitHexString(ProgMemStartAddress + I) & "   "
    For J As Int32 = 0 To 7 Step 1
        DisplayString = DisplayString &
            ConvertIntegerTo16BitHexString(InputBufferProgMemory(I + J)) & " "
    Next J
    DisplayString = DisplayString & vbCrLf
    Main.textboxContents.Text = DisplayString
    Main.textboxContents.Refresh()
    Application.DoEvents()
Next I
DisplayString = DisplayString & vbCrLf
' Display EEPROM memory
Main.labelStatusLine.Text = "Now formatting EEPROM memory.  Please keep waiting ..."
Main.labelStatusLine.Refresh()
DisplayString = DisplayString & "EEPROM memory" & vbCrLf
For I As Int32 = 0 To (NumEEPROMBytes - 1) Step 8
    DisplayString = DisplayString & "&H" &
        ConvertIntegerTo16BitHexString(EEPROMStartAddress + I) & "   "
    For J As Int32 = 0 To 7 Step 1
        DisplayString = DisplayString &
            ConvertIntegerTo8BitHexString(InputBufferEEPROMMemory(I + J)) & " "
    Next J
    DisplayString = DisplayString & vbCrLf
    Main.textboxContents.Text = DisplayString
    Main.textboxContents.Refresh()
    Application.DoEvents()
Next I
DisplayString = DisplayString & vbCrLf
'Display all four UserID words on one line
DisplayString = DisplayString & "UserID words" & vbCrLf &
    "&H" & ConvertIntegerTo16BitHexString(UserIDStartAddress) & "   "
For I As Int32 = 0 To (NumUserIDWords - 1) Step 1
    DisplayString = DisplayString &
        ConvertIntegerTo16BitHexString(InputBufferUserIDWords(I)) & " "
Next I
DisplayString = DisplayString & vbCrLf & vbCrLf
'Display the DeviceID word on a separate line
DisplayString = DisplayString & "DeviceID word" & vbCrLf &
    "&H" & ConvertIntegerTo16BitHexString(DeviceIDStartAddress) & "   " &
    ConvertIntegerTo16BitHexString(InputBufferDeviceIDWord)
DisplayString = DisplayString & vbCrLf & vbCrLf
'Display the Configuration Word(s) on one line
DisplayString = DisplayString & "Configuration Word(s)" &
```

```
            vbCrLf & "&H" & ConvertIntegerTo16BitHexString(ConfigWordsStartAddress) & "   "
        For I As Int32 = 0 To (NumConfigWords - 1) Step 1
            DisplayString = DisplayString &
                ConvertIntegerTo16BitHexString(InputBufferConfigWords(I)) & " "
        Next I
        DisplayString = DisplayString & vbCrLf & vbCrLf
        'Display the Calibration Word on a separate line
        If (NumCalibWords = 1) Then
            DisplayString = DisplayString &
                "Calibration Word" & vbCrLf & "&H" &
                ConvertIntegerTo16BitHexString(CalibWordStartAddress) & "   " &
                ConvertIntegerTo16BitHexString(InputBufferCalibWord)
            DisplayString = DisplayString & vbCrLf & vbCrLf
        End If
        ' All finished
        DisplayString = DisplayString & "End of input buffers"
        Main.textboxContents.Text = DisplayString
        Main.textboxContents.Refresh()
        Main.labelStatusLine.Text = ""
        Main.labelStatusLine.Refresh()
    End Sub

End Module
```

Appendix "E6"

VisualBasic program for the Host computer – Module Debugging.vb

```
Option Strict On
Option Explicit On

' This module contains controls and procedures which can be useful when debugging this device
' on a breadboard.  None of these tools is used during normal operation.

Public Module Debugging

    ' ***********************************************************************************************
    ' *** Subroutines used for debugging purposes only ********************************************
    ' CommunicationsTest1()
    ' CommunicationsTest2()
    ' CommunicationsTest3()

    Public Sub CommunicationsTest1()
        ' This subroutine tests reception from the programmer.  The programmer runs a
        ' continuous loop, sending an incrementing byte to the Host.  The programmer
        ' sends the bytes in groups of eight, with one millisecond between bytes and
        ' one-half second between groups.  The Host should be able to keep pace and,
        ' once it is receiving, can self-detect a break in the sequence.
        Dim ByteReceived As Int32
        Dim ByteExpected As Int32
        Dim IterationNumber As Int32 = 0
        Dim NumberOfErrors As Int32 = 0
        ' Wait for the first byte
        ReceiveOneByteFromProgrammer(ByteReceived)
        ByteExpected = ByteReceived
        Do
            IterationNumber = IterationNumber + 1
            ByteExpected = ByteExpected + 1
            If (ByteExpected >= 256) Then
                ByteExpected = 0
            End If
            Main.labelStatusLine.Text =
                "Iteration #" & Trim(IterationNumber.ToString) &
                " with expected byte &&H" & ConvertIntegerTo8BitHexString(ByteExpected)
            Main.labelStatusLine.Refresh()
            ReceiveOneByteFromProgrammer(ByteReceived)
            If (ByteReceived <> ByteExpected) Then
                NumberOfErrors = NumberOfErrors + 1
                Main.labelErrorLine.Text =
                    "Number of errors = " & Trim(NumberOfErrors.ToString)
                Main.labelErrorLine.Refresh()
                ' Reset the expectation in the event of error
                ByteExpected = ByteReceived
            End If
            ' Check to see if the user has clicked on the Exit button
            Application.DoEvents()
        Loop
    End Sub

    Public Sub CommunicationsTest2()
        ' This subroutine tests transmission to the programmer.  This subroutine runs a
```

```
        ' continuous loop, sending an incrementing byte to the programmer.  The bytes are sent
        ' in groups of eight, with one millisecond between bytes and one-half second between
        ' groups.  The programmer should be able to keep pace and, once it is receiving, can
        ' self-detect a break in the sequence.
        Dim ByteToSend As Int32 = 0
        Dim IterationNumber As Int32 = 0
        Do
            For I As Int32 = 0 To 7 Step 1
                IterationNumber = IterationNumber + 1
                ByteToSend = ByteToSend + 1
                If (ByteToSend >= 256) Then
                    ByteToSend = 0
                End If
                Main.labelStatusLine.Text =
                    "Iteration #" & Trim(IterationNumber.ToString) &
                    " sending byte &&H" & ConvertIntegerTo8BitHexString(ByteToSend)
                Main.labelStatusLine.Refresh()
                SendOneByteToProgrammer(ByteToSend)
                FixedTimeDelay(1000)
            Next I
            ' Delay 500ms between groups
            FixedTimeDelay(500000)
            ' Check to see if the user has clicked on the Exit button
            Application.DoEvents()
        Loop
    End Sub

    Public Sub CommunicationsTest3()
        ' This subroutine tests communications with the programmer.  It is a continuous
        ' loop which sends an incrementing single byte to the programmer.  The programmer
        ' waits one millisecond and then sends the complement of the byte back to the Host.
        ' After receiving the byte, this subroutine checks for an error, then waits one
        ' millisecond before sending the next byte to the programmer.
        Dim ByteToSend As Int32 = 0
        Dim ByteReceived As Int32
        Dim IterationNumber As Int32 = 0
        Dim NumberOfErrors As Int32 = 0
        Do
            IterationNumber = IterationNumber + 1
            ByteToSend = ByteToSend + 1
            If (ByteToSend >= 256) Then
                ByteToSend = 0
            End If
            FixedTimeDelay(1000)
            Main.labelStatusLine.Text =
                "Iteration #" & Trim(IterationNumber.ToString) &
                " sending byte &&H" & ConvertIntegerTo8BitHexString(ByteToSend)
            Main.labelStatusLine.Refresh()
            SendOneByteToProgrammer(ByteToSend)
            ReceiveOneByteFromProgrammer(ByteReceived)
            If (ByteReceived <> (ByteToSend Xor &HFF)) Then
                NumberOfErrors = NumberOfErrors + 1
                Main.labelErrorLine.Text =
                    "Number of errors = " & Trim(NumberOfErrors.ToString)
                Main.labelErrorLine.Refresh()
            End If
            ' Check to see if the user has clicked on the Exit button
            Application.DoEvents()
```

```
        Loop
    End Sub

End Module
```