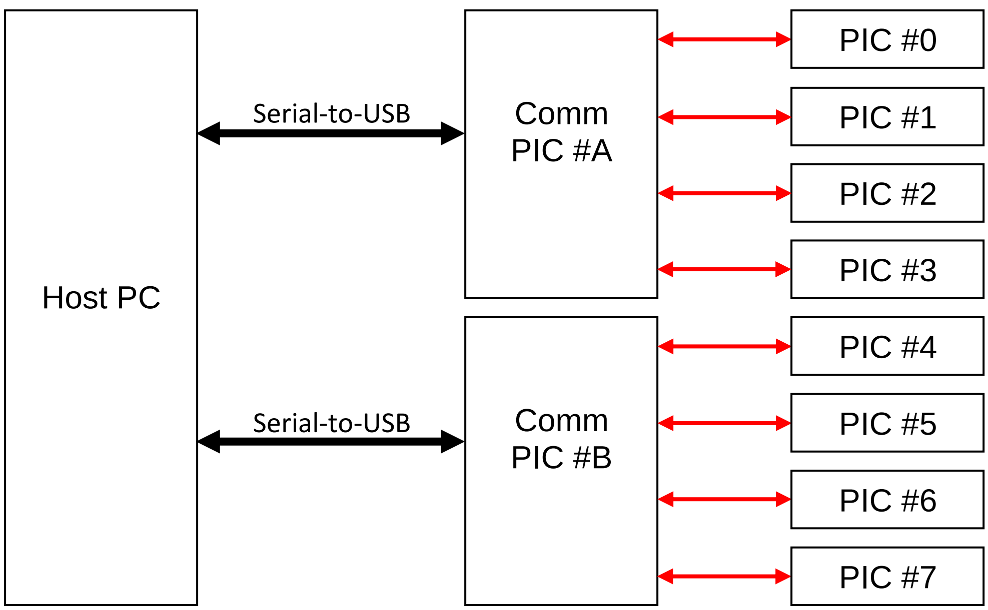
PIC-to-Host PC communication test for my post-War Lionel control system

The control system for my post-War Lionel layout uses many Microchip 16F882 microcontrollers ("PICs"). Control of the hardware is divided into eight separate channels, each controlled by one PIC. For example, Channel #0 is managed by PIC #0, and collects data about the sections of track which are occupied. Channel #1 is managed by PIC #1, and closes and opens turnouts. Channel #4 is managed by PIC #4, and receives speed and e-Unit commands from the locomotive handsets.

Two separate PICs, called CommPIC #A and CommPIC #B, are the intermediaries between the eight channel-control PICs and the Host PC. Each manages four of the channel-control PICs. CommPIC #A and CommPIC #B only handle communications - they do not do any "thinking". They relay commands sent out by the Host PC to the appropriate channels, and relay information coming in from the channels to the Host PC.

CommPIC #A and CommPIC #B are connected to the Host PC through serial-to-USB adapter cables. The protocol is full-duplex asynchronous RS232 at 19,200 baud.

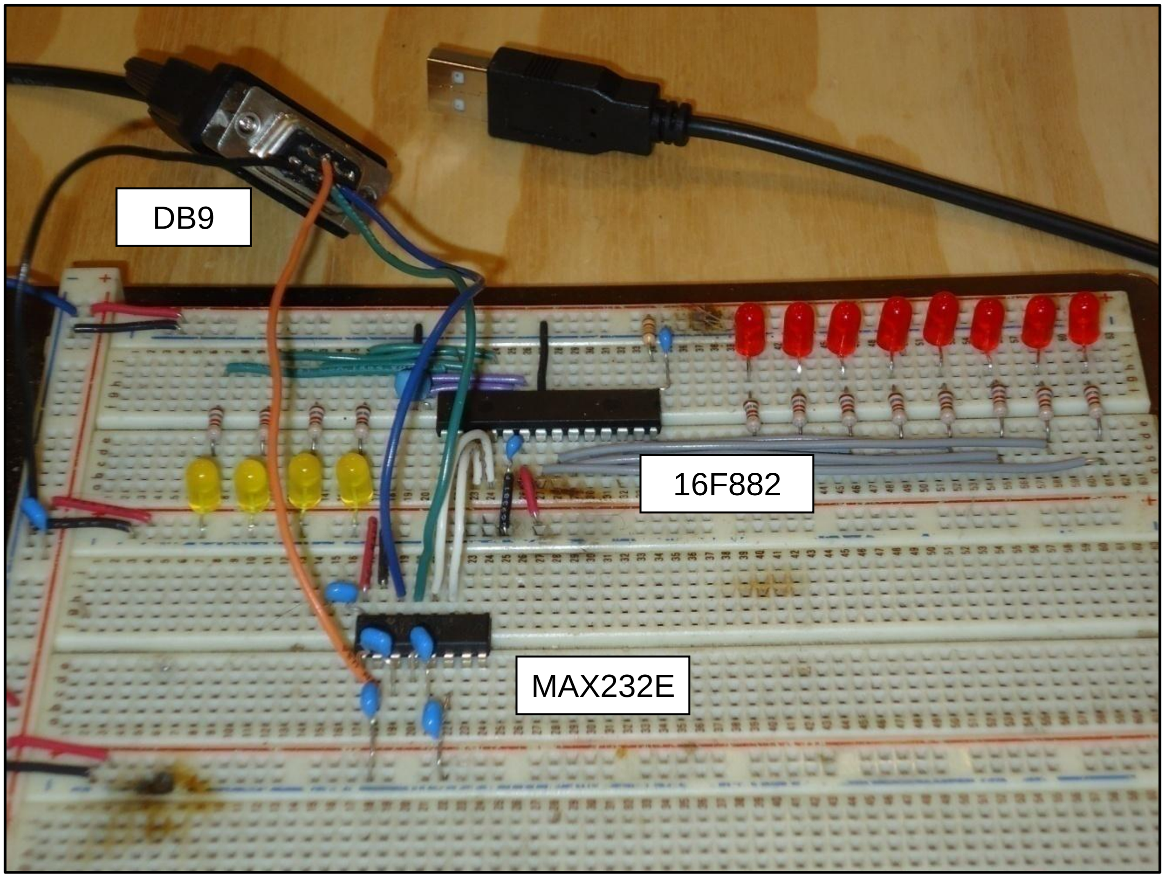


The hardware and software for CommPICA and CommPICB are almost identical. (The only difference is in register names -- whereas CommPIC #A makes reference to Channels 0-3, CommPIC #B makes reference to Channels 4-7. This difference does not arise in the test program described below.) I used two separate PICs because the 16F882 chip has only 22 I/O pins, a limitation which makes it cumbersome to connect to more than four channel PICs. Of course, the Host PC has to remember which CommPIC supervises which channels and to address the appropriate CommPIC through its dedicated USB port.

This paper deals solely with a test of the communication between the Host PC and one of the CommPICs. A separate test, of the communication between the CommPICs and the individual channel-control PICs will be described in a subsequent paper.

The breadboard used for the test

The following photograph shows the breadboard used for the test. The schematic diagram is set out in Appendix "A" below.



Eight red LEDs are driven by portB. They display data being exchanged during the test. Four yellow are used to display error codes. They are driven by the low nibble of portC.

The program in the PIC uses Microchip's built-in EUSART module, which reserves pin 18 (RC7) for reception from the Host PC and pin 17 (RC6) for transmission to the Host. A MAX232E chip is used as a line driver. The MAX232e has two channels for input signals and two for output signals, but only one of each is used here. I have confirmed that the chip has internal protection against floating inputs, so have not added pull-up resistors at the chip's unused inputs.

Signals received from the Host PC run along the green wire from pin 3 of the DB9 connector. In a pinout of the DB9 connector, pin 3 is called the TX pin because it handles transmissions from the Host. Signals sent by the PIC to the Host run along the blue wire to pin 2 of the DB9 connector. Pin 2 is called the RX pin because it handles reception by the Host.

The MAX232E chip requires five capacitors. Four of them are used by the voltage-level shifters in the chip. This chip is a convenient way to interface the 5V logic used by the PIC to the ±8.5V levels required by the RS232 protocol.

The photograph shows four wires soldered to the DB9 connector. There are the TX and RX lines, of course. Ground is connected to pin 5 of the connector through the black wire. The last wire -- the orange one -- is soldered to the CTS line (pin 8) of the DB9 connector. CTS stands for "Clear-to-send" and was once widely used by printers and terminals to tell the boss (the mainframe computer) that they were ready to receive something. In this circuit, the CTS line is connected directly to pin 2 of the MAX232E chip. That pin is the +8.5V voltage which the chip generates internally. The Host PC checks the CTS line for this voltage, and uses it as a test of whether or not the PIC is powered up.

I need to say a word about the USB-to-serial port cable. I spent several frustrating hours before I discovered that the USB-to-serial port cables I had been using were not compatible with Windows 10. It seems that many such cables became obsolete when Microsoft migrated up to Windows 10. The packaging or datasheet for your new cables should state explicitly that they will work with Windows 10.

Packets contain 24 bits

All information throughout the control system, including communication between PICs, is sent in 24-bit packets. When necessary, as in the RS232 protocol, the 24 bits are sent as three consecutive bytes. The three bytes are always transmitted in this order: the high byte first, then the middle byte, with the low byte last.

Test programs and test results

The CommPIC runs the Microchip Assembly code listed in Appendix "B". The Host PC runs the Visual Basic program listed in Appendix "C".

In the test, the Host PC sends a packet to the CommPIC. The CommPIC complements the packet and sends it back to the Host PC. Once the Host PC verifies that the packet it received is correct, it increments the value in the outgoing packet, and sends the incremented value to the CommPIC. In the test, the initial packet is &H000000. The two computers exchange packets until the final packet -- &H07FFFF -- has been exchanged.

During the test, the Host PC sends out a total of 8 * 256 * 256, or 524,288 packets. The CommPIC sends one packet back for each packet received. Since each packet contains 24 information bits, a total of 524,288 * 2 * 24, or 25,165,824 information bits are sent during the test.

With the RS232 configuration set to 19,200 baud, the test takes 4,681 seconds, representing a speed of 5,376 information bits per second. This is so far below the nominal 19,200 baud rate that it deserves investigation. See below.

Initialization sequence

The production versions of CommPIC #A and CommPIC #B do not have on/off switches or reset pushbuttons. The CommPICs begin running as soon as the master power switch for the complete control system is flipped on. It is therefore imperative that the CommPICs and the Host PC are able to begin talking to each other automatically and without outside intervention.

When a CommPIC is powered up, it begins running a infinite loop waiting to receive a single ping byte 0xF5. Interrupts are not enabled and time-outs are not recognized. The loop simply tests bit `PIR1<rcif>`, which will go high when the EUSART receives one byte. The CommPIC ignores any errors and ignores any other bytes until it receives 0xF5. When it gets that ping, the CommPIC sends a single byte -- 0xF6 -- back to the Host PC. The CommPIC then enables receive-complete interrupts and begins running a second loop, waiting (indefinitely, if need be) for the Host PC to send a specific ping packet 0xF00505. When it's received, the PIC sends a specific response ping packet 0xF00606.

The Host PC is under the control of the human User, who starts the Visual Basic program by clicking on a button on the main form. The Host PC first checks to see whether or not the serial-to-USB cable is plugged in. If the cable is not plugged in, the program prompts the User and waits (indefinitely, if need be) until the cable is plugged in. The Host PC then checks to see whether or not the CommPIC is powered up. It uses the CTS line for this purpose. If the CommPIC is not powered up, the program prompts the User and begins to wait (indefinitely, if need be) until the power is turned on. At this point, the Host PC enters an infinite loop, sending ping byte &HF5 to the CommPIC and waiting 25ms for a reply.

Eventually, the Host PC will receive the expected response ping byte: &HF6. After a short wait, long enough to allow the PIC to configure itself for packet transmission, the Host PC sends ping packet &HF00505. When it receives ping response packet &HF00606, the Host PC knows that initialization and synchronization are complete and begins running the main program, which in this case is the test program.

Details about the PIC program

The Assembly code for the PIC test program is listed in Appendix "B". The program uses interrupts to send and receive packets. Let me describe reception first.

When the EUSART module receives a byte, interrupt flag bit `PIR1<rcif>` will go high. Once the initialization process is complete, receive-complete interrupts are enabled by setting interrupt enable bit `PIE1<rcie>` high. Receive-complete interrupts remain enabled as long as the program runs.

Six User-registers are used for reception:

- `HostInByte` holds the byte just received
- `HostInPktH, HostInPktM, HostInPktL` hold the three bytes in the packet
- `HostInNumByte` holds the numbers of bytes received so far
- `IntTrig<HostPktRecd>` is set high when a whole packet is ready

When an interrupt occurs, the ISR (Interrupt Service Routine) "reads" the byte by moving it from the EUSART register `RCREG` into User-register `HostInByte`. By referring to `HostInNumByte`, the ISR can tell whether this byte is the first (high), second (middle) or third (low) byte in the packet. Unless this byte is the last one in the packet, the ISR simply stores the byte received, increments the count in `HostInNumByte` and returns. If the byte is the last one, then the ISR does two more things: (i) it clears the count in `HostInNumByte` in readiness for the next packet to be received, and (ii) it sets the flag bit `IntTrig<HostPktRecd>`. Note that the ISR does not need to, and cannot, clear the `PIR1<rcif>` flag which triggered these interrupts.

Throughout this period, the main program has been in a loop watching the `IntTrig<HostPktRecd>` bit. When it goes high, the main program moves on. It complements the bytes in the packet and transmits them back to the Host PC.

Let me now describe transmission. The interrupt flag bit used for transmission needs special treatment. This bit `PIR1<txif>` goes low one instruction cycle after the byte to be transmitted is moved into EUSART register `TXREG`. It goes high after the EUSART module has processed the byte (which does not necessarily mean that transmission is finished, just that another byte can now be moved into register `TXREG`). This has two consequences. Firstly, the main program must start the transmission process, which it does by moving the first (high) byte of the packet into register `TXREG`. Secondly, transmission-complete interrupts cannot be enabled until after this first byte has been moved into `TXREG`. They are enabled by setting bit `PIE1<txie>` high.

Seven User-registers are used for transmission:

- `HostOutByte` holds the byte to be transmitted
- `HostOutPktH, HostOutPktM, HostOutPktL` hold the three bytes in the packet
- `HostOutNumByte` holds the numbers of bytes transmitted so far
- `IntTrig<HostPktSent>` is set high when the whole packet has been sent
- `IntDoISR` is a clue to the ISR about what to do

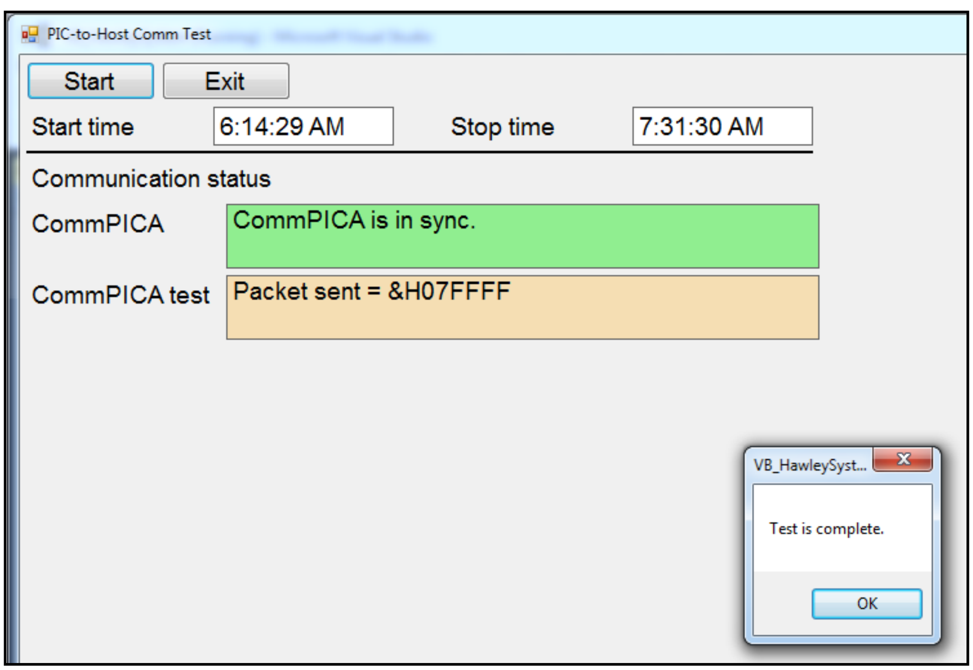
When the ISR is called by a transmission-complete interrupt, the ISR refers to the count in `HostOutNumByte` to determine which byte has been sent. It then loads the next byte into register `TXREG` and returns. When the last byte has been sent, the ISR does four things: (i) it clears the count in `HostOutNumByte` in readiness for the next packet to be sent, (ii) it sets the flag bit `IntTrig<HostPktSent>`, (iii) it disables transmission-complete interrupts and (iv) it clears the "clue" register `IntDoISR`.

What is the clue register? In many programs, including this test program, there are multiple sources of interrupts. When an interrupt occurs, the ISR has to figure out what caused the interrupt. It usually does this by examining the interrupt flags bits to find out which one is high. But this process can fail if a particular source of interrupts is disabled. Such is the case in the test program. The transmission-interrupt flag bit `PIR1<txif>` will almost always be high -- it is low only during the short periods of time a byte is being transmitted. If an interrupt occurs for some other reason, such as a receive-complete, the transmit-complete flag will be high. The ISR would be misled into thinking that a transmission was underway.

I use register `IntDoISR` as a mask to tell the ISR whether or not to process transmission-complete interrupts. If bit `IntDoISR<HostPktSend>` is high, the ISR will process transmissions; if it is low the ISR will not process transmissions. (As an alternative, the ISR could test the interrupt enable bit `PIE1<txie>` to determine if it should be processing these interrupts. But system-register `PIE1` is located in bank 1, and it is more convenient to use User-register `IntDoISR` located in bank 0.)

Details about the Visual Basic program

The Visual Basic code for the Host PC test program is listed in Appendix "C". It consists of a main form `Form1.vb` and one module `PICComm.vb`. The main form contains text boxes which display information to the User. It displays the times at which the test started and finished. It has a text box to display the status of communication. The background colour of the status textbox is set to light red or light green as appropriate. Lastly, there is a textbox which displays the contents of the packet currently being sent to the PIC. The following screenshot shows the main form when the test program is done.



The real work of the test is carried out in module `PICComm.vb` using the SerialPort object. The properties of the serial port must be defined before the port is opened. I used the following declaration:

```
CommPICASerialPort = New SerialPort
CommPICASerialPort.PortName = DefaultCommPICASerialPort
CommPICASerialPort.BaudRate = 19200
CommPICASerialPort.DataBits = 8
CommPICASerialPort.StopBits = StopBits.One
CommPICASerialPort.Parity = Parity.None
CommPICASerialPort.Handshake = Handshake.None
CommPICASerialPort.ReadBufferSize = 2048
CommPICASerialPort.WriteBufferSize = 2048
CommPICASerialPort.ReceivedBytesThreshold = 1
CommPICASerialPort.ReadTimeout = -1
```

There are several things to note:

1. The protocol does not check parity, even though it would be better practice to use at least one parity bit. The limitation here is in the PIC. While the EUSART module of the PIC does have a procedure to inject a ninth bit into every byte, and that ninth bit can be used as a parity bit, the procedure to do so is a little bit clumsy and time-consuming. Because the USB-to-serial port cable is relatively short and will be used a considerable distance away from the electrical noise underneath the layout, I elected to dispense with the parity check.

2. `ReceivedBytesThreshold` is set to one. That means that the operating system will alert the program as soon as it can after every single byte is received.

3. `ReadTimeout` is set to minus one to disable time-outs.

Reception is carried out by an asynchronous reader which runs constantly throughout the test. The asynchronous reader is defined as:

```
Private Async Sub CommPICAAsyncReadBytes()
```

This subroutine contains an infinite loop which reads any and all bytes which have become available since the previous pass through the loop. The following function call returns the number of bytes which are now available and moves them into a temporary byte buffer called `ReceiveBuffer`.

```
NumBytesRead =
      Await CommPICASerialPort.BaseStream.ReadAsync(
      ReceiveBuffer, 0, BytesToRead, CancellationToken.None)
```

If one or more bytes have become available since the last pass through the loop, they are then moved from the `ReceiveBuffer` into the main input buffer `CommPICAInBytes()` by the following `For-Next` block.

```
For I As Int32 = 1 To NumBytesRead Step 1
      CommPICAInBytes_NumBytesAdded =
```

```
                CommPICAInBytes_NumBytesAdded + 1
            CommPICAInBytes(CommPICAInBytes_NumBytesAdded) =
                ReceiveBuffer(I - 1)
    Next I
```

`CommPICAInBytes_NumBytesAdded` is a counter which keeps track of the number of bytes which have been moved into the main storage vector.

Note that I keep using the phrase "bytes which have become available". That is because the Windows operating system does not instantly report every time it receives a byte. It processes serial ports on an every-now-and-then basis, so bytes do not become available on a regular basis.

The main program runs a loop calling function `GetNextPacket()` to determine if a packet has been received. This function uses a second counter, named `CommPICAInBytes_NumBytesUsed`, to keep track of the number of bytes which have been "read" from the main storage vector. Whenever counter `CommPICAInBytes_NumBytesAdded` is greater than counter `CommPICAInBytes_NumBytesUsed` by at least three, then the three "unused" bytes constitute the next packet.

That's reception. Transmission is handled differently. When a packet is ready to send to the PIC, a background worker named `BWCommPICASendPkt_DoWork()` is invoked. It uses the following instruction to send the three bytes in the packet:

```
    CommPICASerialPort.Write(lOutPkt, 0, 3)
```

When the packet has been sent, the operating system calls a handler named `BWCommPICASendPkt_RunWorkerCompleted()`. This handler simply sets a Boolean flag `CommPICASendPktComplete`. The main program runs a loop waiting for this flag to turn True. Once that happens, it moves on and begins to wait for the PIC to respond.

The asynchronous reader used for reception and the background worker used for transmission are useful because they run on separate threads from the main program. Therefore, they do not prevent the main program from doing other things. In the test program here, there is not much for the main program to do, other than wait for reception or transmission to finish. But in the Lionel control system, there is a great deal for the main program to do, and there is not enough free time available for it to "blocked" just to allow communication to take place.

Input and output buffers and thread safety

In the test program, the input buffer used by the Host PC is the vector `CommPICAInBytes()` I mentioned above. It is neither circular nor resettable. I simply made it long enough to hold all 3 * 8 * 256 * 256 = 1,572,864 bytes which will be received during the test. This may seem wasteful, but it is not outrageously so. And, it makes it easy to guarantee thread safety.

The asynchronous reader, running on its own thread, adds byte to the input buffer and moves its counter `CommPICAInBytes_NumBytesAdded` monotonically out to higher indices in the buffer.

The main program and its function `GetNextPacket()` run on a different thread. The function has its own counter `CommPICAInBytes_NumBytesUsed`, which also moves out monotonically to higher indices in the buffer.

The sole point of intersection between the two threads, and the only point of potential conflict, occurs when function `GetNextPacket()` subtracts the "used" counter from the "added" counter to calculate how many unprocessed bytes there are in the buffer. The worst that can possibly happen is that asynchronous reader adds a third byte immediately after the function has done its calculation, with the result that the main program misses the opportunity to process a packet. However, the main program will catch this packet during the next pass through its main loop.

Why is the realized speed so low?

It should be understood that the back-and-forth exchange of packets between the two computers takes no advantage whatsoever of the full-duplex capability that is available. Each computer must wait to receive a full packet, and then process it before sending a full packet out. The turn-around time at each end is the most significant reason why the realized speed is so much less than the nominal baud rate. Let me describe the various uses of time during the test.

1. At 19,200 baud, each bit has a theoretical duration of 1 / 19,200 = 52.083µs. A total of 25,165,824 information bits were sent during the test, which requires a total theoretical time of 52.083µs * 25,165,824 = 1,310.720 seconds.

2. The RS232 protocol used has one start bit and one stop bit per byte. That means that ten bits are sent for each eight information bits. This adds 25%, or 327.680 seconds, to the total theoretical time required.

3. It takes the PIC time to realize that it has received a packet. Although a separate interrupt occurs at the end of each byte received, it is only the time required after the last byte that is of interest here. The time taken to collect the first two bytes does not matter because they are being received at a rate determined by the Host PC at the other end. Let's say that it takes the PIC one instruction cycle to move the last byte into system-register `RCREG`, another instruction cycle to set the `PIR1<rcif>` flag high, two more instruction cycles to push the PC address onto the stack and another two instruction cycles to jump to the start of the Interrupt Service Routine. It takes 25 instruction cycles to work through from the start of the ISR to label `ISR3`, where processing of the third byte received takes place. It takes another 13 instruction cycles to complete the ISR and another four, say, to pop the PC address off the stack and jump back into the main program. All told, 48 instruction cycles are needed before control returns to the main program. The main program is running a tight loop, testing

`IntTrig<HostPktRecd>`. At most, it will take the main program four instruction cycles to detect the bit high and to fall out of the loop into the next step. Thus, a total of 52 instruction cycles elapses between reception of the last bit of the last byte in a packet and starting the next step. Since the PIC is clocked with a 20MHz crystal, each instruction cycle takes 200ns. The 52 instruction cycles take 52 * 0.2µs = 10.4µs. This delay is incurred for each of the 524,288 packets received, for a total delay of 10.4µs * 524,288 = 5.453 seconds.

4. The next step for the PIC after reception is to complement the three bytes in the packet. This takes six instruction cycles, or 6 * 0.2µs = 1.2µs, per packet, for a total delay during the test of 1.2µs * 524,288 = 0.629 seconds.

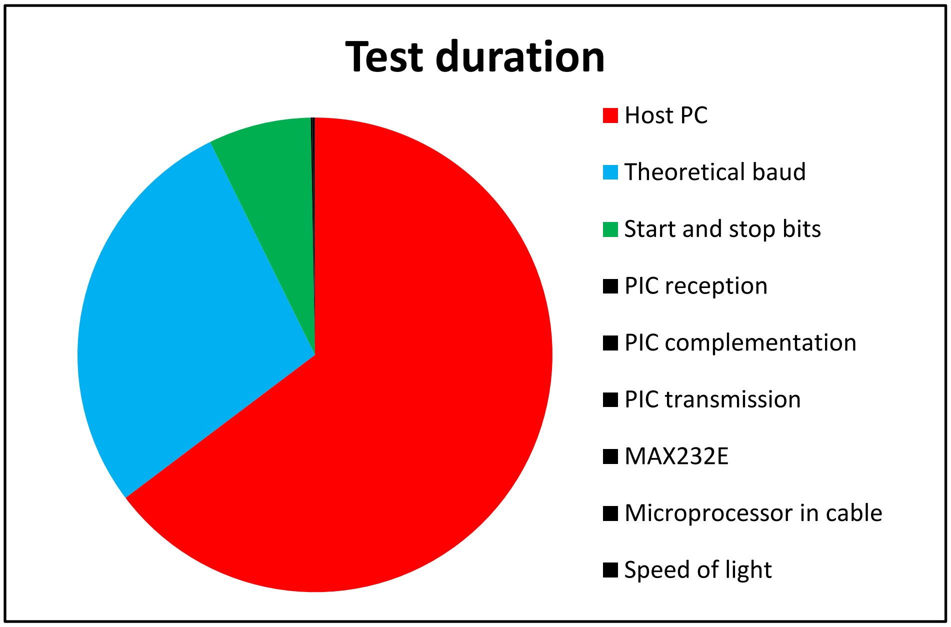
5. Now, for the transmission of the complemented packet back to the Host PC. We need to examine each byte separately. The main program starts the transmission of the first byte. It takes the main program two instruction cycles to load the first byte into register `TXREG`. That step launches the first byte down the cable. We do not really care how long it takes the PIC to enable transmission-complete interrupts and so forth because the first byte is already on its way. The delay we are interested in only starts when the first interrupt occurs. Let's say that it takes the PIC one instruction cycle to set the `PIR1<txif>` flag high, two more instruction cycles to push the PC address onto the stack and another two instruction cycles to jump to the start of the Interrupt Service Routine. It takes eight instruction cycles to work through from the start of the ISR to label `ISR4`, where processing of transmission interrupts begins. It takes another 11 instruction cycles to load the second byte into register `TXREG`. What happens after that does not matter because the second byte is now on its way. When the second interrupt occurs, it once again takes five instruction cycles to get the ISR going, and another eight to get down to label `ISR4`. This time, it takes 17 instruction cycles until the third byte of the packet is loaded into register `TXREG`. That's it -- the last byte is now on its way down the cable and we can ignore the time it takes the PIC to sort itself out and get ready to receive the next packet. All in, it takes 56 instructions to get the packet out the door. These 56 instruction cycles take 56 * 0.2µs = 11.2µs. This delay is incurred for each of the 524,288 packets transmitted, for a total delay of 11.2µs * 524,288 = 5.872 seconds.

6. The MAX232E line driver introduces some delay. Its datasheet says that the propagation delay of a bit passing through is 500ns. There are 24 information bits, or 30 physical bits, in each packet, but the 500ns delay affects each bit equally. The propagation delay does not increase the spacing between the bits -- it simply delays the entire packet by 500ns. Each packet that passes through, in either direction, is delayed by 0.5µs. Thus, the total delay during the test is 0.5µs * 2 directions * 524,288 = 0.524 seconds.

7. The microprocessor embedded in the USB-to-serial port cable is also a source of delay. The cable I have does not a datasheet. I estimate that its propagation delay will be comparable to that of the MAX232E chip. To be conservative, let's say that it adds one second to the duration of the test.

8. The USB-to-serial port cable I use is 36 inches long. It takes time for electrical pulses to travel up and down the cable. Let's assume that the signals travel through the cable at 80% of the speed of light. That is 0.8 * $3x10^8$ = $2.4x10^8$ meters per second. A 36 inch of cable is 0.914 meters long. It therefore takes a pulse 0.914 / $2.4x10^8$ = 0.00381µs to travel from one end to the other. The total travel time for the 1,048,576 packets which travel up or down the cable is 0.00381µs * 1,048,576 = 3995µs, or 0.004 seconds. Note that this factor is one that affects all bits in a packet in the same way, with the result that it does not increase the spacing between the individual bits, but simply delays the entire packet as a whole.

9. The sum of the eight items above is 1,652 seconds. Activity by the Host PC takes up the rest of the time -- 3,029 seconds -- bringing the total time required for the test up to 4,681 seconds. The following pie chart shows where all the time went.



The red slice is the Host PC. The blue slice is the speed required to send 24 bits per packet at the theoretical speed of 19,200 bits per second. The green slice is the extra time required to send start and stop bits at the theoretical speed. All of the other delays, due to the PIC, the MAX232E chip, and so on, are concentrated in the black slice, which is vanishingly small.

To confirm that this analysis is sound, I re-ran the test with the computers at both ends configured to send and receive at 9,600 baud. This time around, the test took a total of 6,350 seconds. At 9,600 baud, each bit has a theoretical duration of 1 / 9,600 = 104.167µs. Since the same number of information bits -- 25,165,824 -- was sent during this test, the total theoretical time required is 104.167µs * 25,165,824 = 2,621.440 seconds. Once again, the start and stop bits add 25%, or 655.360 seconds in this case, to the theoretical time required.

The following table compares the time required for the two tests.

	9,600 baud	19,200 baud
Elapsed time for test	6,350.00	4,681.00
Less :Theoretical time for 24 bits	(2,621.44)	(1,310.72)
Less: Extra time for start and stop bits	(655.36)	(327.68)
"Overhead"	3,073.20	3,042.60

I have called the difference between the actual time required and the theoretical time required the "overhead". That's the time required by the computers to turn around the packets they receive. Note that it is almost exactly the same -- within one percent -- for the two tests. This makes sense since the same packets were sent in both tests and the same programs and hardware were used in both.

The conclusion

The Host PC is, by a huge margin, the biggest source of delay. About two-thirds of the time needed to communicate with the PIC is taken up by the Host PC playing around with itself. It's not because the Host PC runs slowly, but because it runs a great many instructions on multiple threads. There is the Visual Basic program, which interacts with the SerialPort APIs, which in turn interact with the Windows operating system. All of this commotion requires time.

The Windows setup is geared towards sending large files or receiving large files. When a lot of bytes are being sent in one direction, this issue of "turn around" time is much less important. The turn-around and its inherent delay occur only once, after the entire file has been processed. Windows is just not very effective for sending and receiving little bits of information like our 24-bit packets.

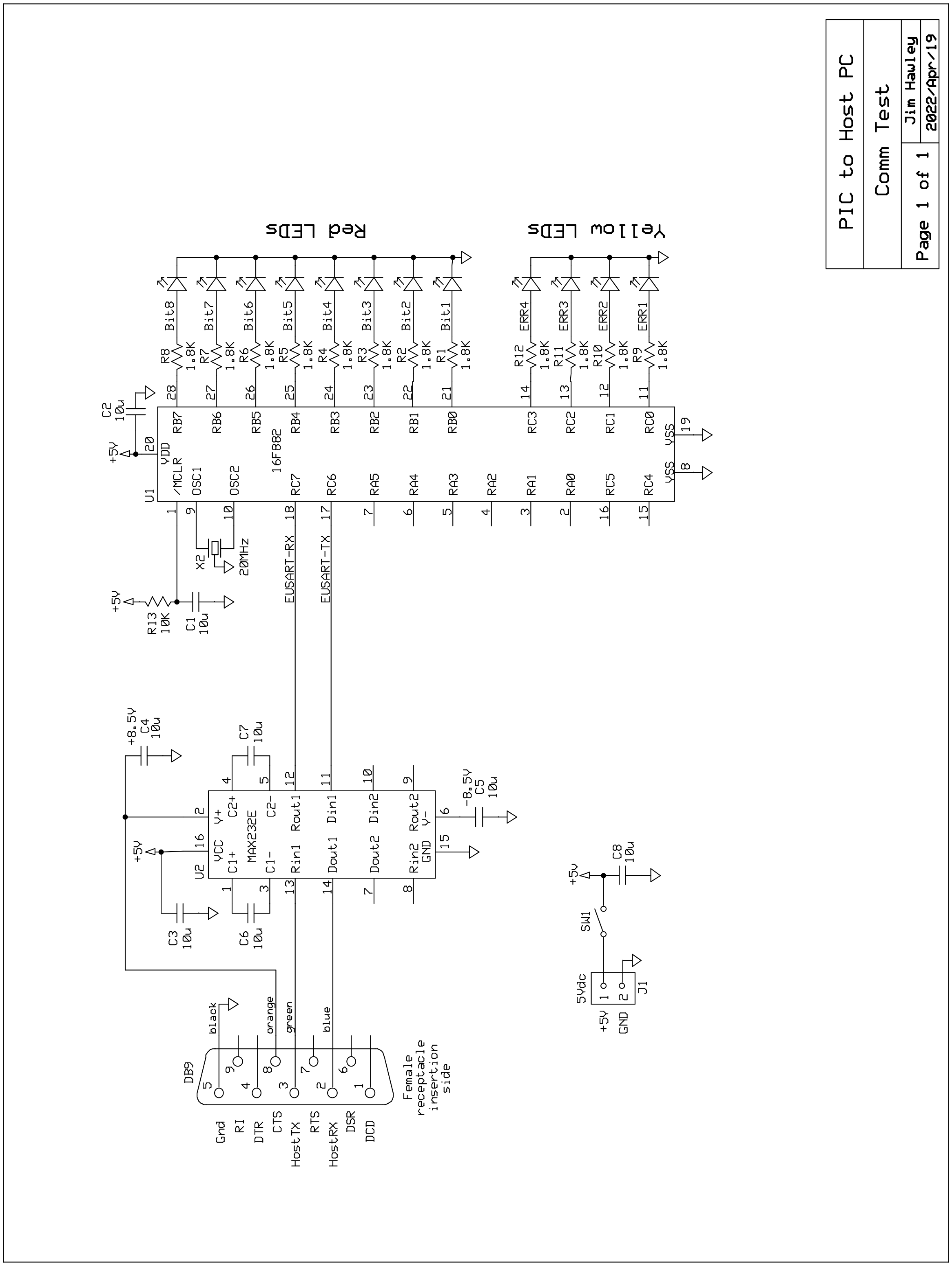
If one needed faster communication between the Host PC and the CommPICs, then one would probably need to migrate to another operating system. I have not done so. One of the biggest reasons is my preference for Visual Basic. It is ideal for programming all of the other tasks which the Lionel control system needs to operate. I find that Visual Basic is an excellent compromise between the absence of structure in C and the anal micromanagement demanded by Python. While there is much not to like about Microsoft, I do have to give credit where it is deserved.

Jim Hawley
June 2022

(As always, an e-mail pointing out errors and omissions would be appreciated.)

Appendix "A"

Schematic diagram of the circuit used to test PIC to Host PC communication



Appendix "B"

Assembly code for the 16F882 PIC

```
; Program for Hawley's Lionel train system
;   PICtoHostCommTest: Communications test between Host PC and CommPIC #A
;   For 16F882 microprocessor
;
; 1. There are two sources of interrupts: (i) UART receive-complete interrupts
;    and (ii) UART transmit-complete interrupts.
;
; 2. User-register IntTrig has a bit for each source of interrupts.  These flags
;    are set by the ISR when it has finished processing an interrupt.  For
;    example, when the ISR has finished reading a three-byte packet sent by the
;    Host PC, it will set bit IntTrig<HostPktRecd>.
;
; 3. A second User-register IntDoISR also has a bit for each source of
;    interrupts.  These bits are set by the main program to tell the ISR whether
;    or not to process a particular type of interrupt.  This is necessary
;    because of the way the ISR (any ISR, not just this one) works.  Once the
;    ISR starts running, it examines the interrupt flag bits.  But the interrupt
;    flag bit for a particular source of interrupts can be set even if
;    interrupts from that source are not enabled.  This is a serious issue for
;    UART transmission.  The PIR1<txif> flag is ALWAYS high except in the very
;    special case when a byte is moved into system register TXREG.  Even then,
;    this flag goes low after one instruction cycle.  Even if PIE1<txie>
;    interrupts are disabled, the PIR1<txif> flag will almost always be set.  If
;    the ISR is called for some other reason, it will be misled by the
;    PIR1<txif> flag into thinking that a transmission has occurred.  To prevent
;    this and similar errors, the ISR checks the appropriate IntDoISR bit for
;    each source of interrupt before processing it.
;
; 4. The ISR handles reception of all three bytes (24 bits) in a packet sent by
;    the Host PC.  User-register HostInNumByte is a counter of the number of
;    bytes which have been received from the Host PC.  It is cleared at power-
;    on.  Once things get started, UART receive-complete interrupts are
;    enabled and they continue to be enabled as long as the program runs.  When
;    a byte is received, the PIC places it in system-register RCREG and triggers
;    a PIR1<rcif> interrupt.  Since HostInNunByte is zero, the ISR knows that
;    this is the first byte in the packet, and moves TXREG into User-register
;    HostInPktH, which is the first (high) byte in the packet.  The ISR then
;    increments the count in HostInNumByte and returns, leaving bit
;    IntTrig<HostPktRecd> low.  The main program can continue to do other stuff.
;    When another byte is received, another PIR1<rcif> interrupt occurs.  Since
;    HostInNumByte is one, the ISR knows that this is the second byte in the
;    packet,  It transfers RCREG into User-register HostInPktM, increments
;    HostInNumByte and returns with IntTrig<HostPktRecd> still low.  When a
;    third byte is received, a third PIR1<rcif> interrupt occurs.  It transfers
;    RCREG into User-register HostInPktL.  This time, though, the ISR knows the
;    packet is complete.  It now sets bit IntTrig<HostPktRecd>.  It also clears
;    the count in User-register HostInNumByte to get things ready for the next
;    packet.  Now, when control returns to the main program, its polling of
;    flag IntTrig<HostPktRecd> will cause it to begin processing the full
;    packet, which is stored in triplet HostInPktH, HostInPktM and HostInPktL.
;
; 5. The ISR also handles transmission of all three bytes (24 bits) in a packet
```

```
;    being sent to the Host PC.  User-register HostOutNumByte is a counter of
;    the number of bytes which have been sent (past tense).  Before starting
;    the transmission, the three bytes are loaded into User-registers
;    HostOutByteH, HostOutByteM and HostOutByteL.  They will be sent in that
;    order.  UART transmission interrupts PIE1<txie> are enabled only AFTER byte
;    HostOutByteH is moved into system-register TXREG.  Note that it takes one
;    instruction cycle for flag bit PIR1<txif> to go low after that move.  The
;    first PIR1<txif> interrupt will occur after the UART module has removed the
;    first byte from system-register TXREG.  (Note that it will take the UART
;    module further time to send the byte even after it has been removed from
;    TXREG.)  Even so, the ISR can tell from the count in HostOutNumByte that
;    the first byte is well on its way.  It therefore loads the second byte
;    (HostOutPktM) into TXREG, increments HostOutNumByte and returns, with
;    completion flag IntTig<HostPktSent> still low.  When the second byte has
;    been removed from TXREG, a second interrupt will occur.  The ISR will then
;    load the third byte (HostOutPktL) into TXREG, increment HostOutNumByte and
;    return, with completion flag IntTrig<HostPktSent> still low.  When the
;    third and last byte has been sent, the ISR will disable transmission-
;    complete interrupts by clearing bit PIE1<txie>.  It will clear the count
;    in HostOutNumByte and return with flag IntTrig<HostPktSent> high.  Now,
;    when control returns to the main program, its polling of flag
;    IntTrig<HostPktSent> will cause it to move on to whatever it does after a
;    packet has been sent.  Note that interrupt flag PIR1<txif> will be high
;    and it will stay high.  This is why User-register IntDoISR is needed -- we
;    do not want the ISR to be misled, if it is called for something else, into
;    thinking that another transmission is under way.
;
; 6. UART interrupts are not used during the initialization of communication
;    with the Host PC.  Only after the Host PC has sent ping byte 0xF5 and this
;    PIC has replied with ping response byte 0xF6 are UART interrupts enabled
;    to allow the exchange of complete packets.
;
; 7. The routines are grouped into the following blocks:
;     A. Definition of system registers
;     B. Definition of user registers
;     C. Interrupt Service Routine
;     D. Initialization of system registers
;     E. Initialization of user registers
;     F. Initialize communication with the Host PC
;     G. Main program
;     H. Subroutine Error_Flash
;     I. Miscellaneous and timing subroutines
;
; 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.
;
; *****************************************************************************
; Block A - Definition of PIC 16F882 system registers
; *****************************************************************************
;
; System registers in Bank 0
TMR0            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             equ         0x07    ; global interrupt enable
peie            equ         0x06    ; peripheral interrupt enable
tmr0ie          equ         0x05    ; Timer0 interrupt enable
tmr0if          equ         0x02    ; Timer0 interrupt flag
PIR1            equ     0x0C        ; Peripheral interrupt flags reg 1
tmr1if          equ         0x00    ; Timer1 interrupt flag
txif            equ         0x04    ; UART transmit interrupt flag
rcif            equ         0x05    ; UART receive interrupt flag
TMR1L           equ     0x0E        ; Timer1 count register low byte
TMR1H           equ     0x0F        ; Timer1 count register high byte
T1CON           equ     0x10        ; Timer1 control register
SSPCON          equ     0x14        ; Synch serial port control reg 1
CCP1CON         equ     0x17        ; Capture/Compare/PWM control reg 1
RCSTA           equ     0x18        ; Receive status and control register
spen            equ         0x07    ; serial port enable
rx9             equ         0x06    ; 9-bit receive enable
sren            equ         0x05    ; single receive enable
cren            equ         0x04    ; continuous receive enable
adden           equ         0x03    ; address detect enable
ferr            equ         0x02    ; framing error
oerr            equ         0x01    ; overrun error
rx9d            equ         0x00    ; 9th bit of received data
TXREG           equ     0x19        ; UART transmit data register
RCREG           equ     0x1A        ; UART receive data register
CCP2CON         equ     0x1D        ; Capture/Compare/PWM control reg 2
ADCON0          equ     0x1F        ; Analogue-to-digital control reg 0
;
; System registers in Bank 1
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
PIE1            equ     0x8C        ; Peripheral interrupt enable reg 1
tmr1ie          equ         0x00    ; Timer1 interrupt enable
txie            equ         0x04    ; UART transmit interrupt enable flag
rcie            equ         0x05    ; UART receive interrupt enable flag
PCON            equ     0x8E        ; Power control register
```

```
WPUB            equ     0x95        ; portB weak pull-up resistors
IOCB            equ     0x96        ; portB interrupt-on-change
TXSTA           equ     0x98        ; Transmit status and control register
csrc            equ         0x07    ; clock source select
tx9             equ         0x06    ; 9-bit transmit enable
txen            equ         0x05    ; transmit enable
synch           equ         0x04    ; UART mode select
sendb           equ         0x03    ; send break character
brgh            equ         0x02    ; high baud rate select
trmt            equ         0x01    ; transmit shift register status
tx9d            equ         0x00    ; 9th bit of transmit data
SPBRG           equ     0x99        ; Serial port baud rate generator
SPBRGH          equ     0x9A        ; Serial port baud rate generator (high)
PSTRCON         equ     0x9D        ; pulse steering control register
;
; System registers in Bank 2
CM1CON0         equ     0x107       ; Comparator C1 control register 0
CM2CON0         equ     0x108       ; Comparator C2 control register 0
CM2CON1         equ     0x109       ; Comparator C2 control register 1
;
; System registers in Bank 3
BAUDCTL         equ     0x187       ; Baud rate control register
abdovf          equ         0x07    ; auto-baud detect overflow
rcidl           equ         0x06    ; receive idle flag
sckp            equ         0x04    ; synchronous clock polarity select
brg16           equ         0x03    ; 16-bit baud rate generator
wue             equ         0x01    ; wake-up enable
abden           equ         0x00    ; auto-baud detect enable
ANSEL           equ     0x188       ; Analogue select channels 0-7
ANSELH          equ     0x189       ; Analogue select channels 8-13
;
f               equ         0x01    ; f and w identify destination register
w               equ         0x00
;
; ****************************************************************************
; Block B - Definition of user registers - Accessible only in bank 0
; ****************************************************************************
;
; I/O ports
portAmirror     equ     0x20
ncRA0           equ         0x00    ; Output - not connected
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
Bit1            equ         0x00    ; Output - Display LED - LSB
Bit2            equ         0x01    ; Output - Display LED
Bit3            equ         0x02    ; Output - Display LED
Bit4            equ         0x03    ; Output - Display LED
Bit5            equ         0x04    ; Output - Display LED
Bit6            equ         0x05    ; Output - Display LED
Bit7            equ         0x06    ; Output - Display LED
Bit8            equ         0x07    ; Output - Display LED - MSB
;
```

```
portCmirror     equ     0x22
ERR1            equ         0x00    ; Output - Error LED - LSB
ERR2            equ         0x01    ; Output - Error LED
ERR3            equ         0x02    ; Output - Error LED
ERR4            equ         0x03    ; Output - Error LED - MSB
Byte1           equ         0x04    ; Output - not connected
Byte2           equ         0x05    ; Output - not connected
TX              equ         0x06    ; Output - UART asynchronous transmit
RX              equ         0x07    ; Input - UART asynchronous receive
;
; Flags to indicate which types of interrupts should be processed by the ISR.
; If a bit is zero, the corresponding interrupts are not processed
IntDoISR                equ     0x23
HostPktRecv     equ         0x00    ; A packet is being received from Host PC
HostPktSend     equ         0x01    ; A packet is being sent to Host PC
TimerZero       equ         0x02    ; 1ms Timer0 expiry
TimerOne        equ         0x03    ; 10ms Timer1 expiry
;
; Flags to indicate which types of interrupts have occurred, and been processed
; by the ISR
IntTrig         equ     0x24        ; Flags for interrupt identification
HostPktRecd     equ         0x00    ; A packet has been received from Host PC
HostPktSent     equ         0x01    ; A packet has been sent to Host PC
TimerZero       equ         0x02    ; 1ms Timer0 expiry
TimerOne        equ         0x03    ; 10ms Timer1 expiry
;
; Any byte received from or sent to the Host PC
HostInByte      equ     0x25
HostOutByte     equ     0x26
;
; Three-byte packet being received from the host PC
HostInNumByte   equ     0x27        ; Number of bytes received so far
HostInPktH      equ     0x28        ; Most-significant byte in packet
HostInPktM      equ     0x29
HostInPktL      equ     0x2A        ; Least-significant byte in packet
;
; Three-byte packet being sent to the host PC
HostOutNumByte  equ     0x2B        ; Number of bytes sent so far
HostOutPktH     equ     0x2C        ; Most-significant byte in packet
HostOutPktM     equ     0x2D
HostOutPktL     equ     0x2E        ; Least-significant byte in packet
;
; Register used to display error codes
ErrorCode       equ     0x2F
;
; Temporary registers used in the subroutines indicated
tempDus         equ     0x30        ; del10us(), del50us() and del100us()
;
; Registers used for saves before executing ISR
w_temp          equ     0x70
status_temp     equ     0x71
;
; ****************************************************************************
; Hard start
; ****************************************************************************
;
        org     0x0000
```

```
HardStart
        bcf     INTCON,gie
        goto    InitializeSystemRegisters
;
; ******************************************************************************
; Block C - Interrupt Service Routine
; ******************************************************************************
;
        org     0x0004
ISR
        ; Disable global interrupts
        bcf     INTCON,gie
        ; Save current status and w-reg.  Swaps do not affect status bits.
        movwf   w_temp
        swapf   STATUS,w
        movwf   status_temp
ISR1
        ; Branch based on the UART receive-complete interrupt flag
        btfss   PIR1,rcif
        goto    ISR4                    ; Goto since not a receive interrupt
        ; Do we want to process receive-complete interrupts?  This test is
        ; not really necessary since we always want to process packets received
        ; from the Host PC.
        ;btfss  IntDoISR,HostPktRecv
        ;goto   ISR_Finish              ; Goto since we do not want these interrupts
        ; **********************************************************************
        ; *** Process a byte received from the Host PC ************************
        ; **********************************************************************
        ; Before reading the byte, check the error flags in register RCSTA
        btfsc   RCSTA,ferr              ; Bit=1 after a framing error
        goto    Error_FrameErr          ; Goto since a framing error occurred
        btfsc   RCSTA,oerr              ; Bit=1 after an overrun error
        goto    Error_OverrunErr        ; Goto since an overrun error occurred
        ; No errors were detected, so read the byte received
        movf    RCREG,w
        movwf   HostInByte              ; Save the byte in register HostInByte
        ; These interrupts occur after a byte has been read, so register
        ; HostInNumByte needs to be incremented.  Note that HostInNumByte is
        ; reset to zero after the third byte in a packet has been received.
        incf    HostInNumByte,f
        ; If appropriate, save the first (high) byte of the incoming packet
        movf    HostInNumByte,w
        xorlw   0x01                    ; Z=1 if number of bytes received = 1
        btfss   STATUS,zero
        goto    ISR2                    ; Goto since number of bytes received > 1
        movf    HostInByte,w
        movwf   HostInPktH              ; Save high (first) byte of packet
        goto    ISR_Finish
ISR2
        ; If appropriate, save the second (middle) byte of the incoming packet
        movf    HostInNumByte,w
        xorlw   0x02                    ; Z=1 if number of bytes received = 2
        btfss   STATUS,zero
        goto    ISR3                    ; Goto since number of bytes received > 2
        movf    HostInByte,w
        movwf   HostInPktM              ; Save middle byte of packet
        goto    ISR_Finish
```

```
ISR3
        ; The only choice left is to save the third (low) byte of the packet
        movf    HostInByte,w
        movwf   HostInPktL          ; Save low (last) byte of packet
        ; Reset HostInNumByte to zero in preparation for the next packet
        clrf    HostInNumByte       ; Set number of bytes received = 0
        ; Tell the MainProgram that a packet has been received
        bsf     IntTrig,HostPktRecd
        goto    ISR_Finish
ISR4
        ; Branch based on the UART transmit-complete interrupt flag
        btfss   PIR1,txif
        goto    ISR_Finish          ; Goto since not a transmit interrupt
        ; Do we want to process transmit-complete interrupts?
        btfss   IntDoISR,HostPktSend
        goto    ISR_Finish          ; Goto since we do not want these interrupts
        ; ***********************************************************************
        ; *** Process completion of sending a byte to the Host PC **************
        ; ***********************************************************************
        ; These interrupts occur after a byte has been transferred out, so
        ; register HostOutNumByte needs to be incremented.  HostOutNumByte is
        ; reset to zero after the third byte has been sent.
        incf    HostOutNumByte,f
        ; If appropriate, send the second (middle) byte of the outgoing packet
        movf    HostOutNumByte,w
        xorlw   0x01                ; Z=1 if number of bytes sent = 1
        btfss   STATUS,zero
        goto    ISR5                ; Goto since number of bytes sent > 1
        movf    HostOutPktM,w
        movwf   TXREG               ; Send the middle byte of the packet
        goto    ISR_Finish
ISR5
        ; If appropriate, send the third (low) byte of the outgoing packet
        movf    HostOutNumByte,w
        xorlw   0x02                ; Z=1 if number of bytes sent = 2
        btfss   STATUS,zero
        goto    ISR6                ; Goto since number of bytes sent > 2
        movf    HostOutPktL,w
        movwf   TXREG               ; Send the low byte of the packet
        goto    ISR_Finish
ISR6
        ; Three bytes have been sent, so the packet transmission is complete
        bsf     STATUS,page0        ; Select register bank 1
        bcf     PIE1,txie           ; Disable UART transmit-complete interrupts
        bcf     STATUS,page0        ; Re-select register bank 0
        ; Record that we no longer want the ISR to process interrupts arising
        ; from transmission to the Host PC
        bcf     IntDoISR,HostPktSend
        ; Reset HostOutNumByte to zero in preparation for the next packet
        clrf    HostOutNumByte
        ; Tell the MainProgram that the packet has been sent
        bsf     IntTrig,HostPktSent
        goto    ISR_Finish
ISR_Finish
        ; End-of-interrupt
        swapf   status_temp,w       ; Retrieve the original status and w-reg
        movwf   STATUS
```

```
        swapf   w_temp,f
        swapf   w_temp,w
        bsf     INTCON,gie          ; Re-enable global interrupts
        retfie
;
; ******************************************************************************
; Block D - Initialization of system registers
; ******************************************************************************
;
InitializeSystemRegisters
        ; Select register bank 0
        bcf     STATUS,page0
        bcf     STATUS,page1
        ; Reset the Timer0 counter
        clrf    TMR0
        ; INTCON=0 disables all interrupt activity (affects portB)
        clrf    INTCON
        ; Reset the Timer1 counters
        clrf    TMR1L
        clrf    TMR1H
        ; PIR1=0 clears all peripheral interrupt flags
        clrf    PIR1
        ; Configure Timer1
        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
        ; Configure RCSTA: UART receive status and control register
        ; Default configuration is for both Transmit and Receive modes enabled,
        ; with the UART module not enabled yet.
        ;   RCSTA<spen>=0/1         ; Serial port disabled/enabled
        ;   RCSTA<rx9>=0            ; 8-bit transmission
        ;   RCSTA<sren>=0           ; Don't care (Asynchronous mode)
        ;   RCSTA<cren>=0/1         ; Disable/enable receiver
        ;   RCSTA<adden>=0          ; Disable address detection
        ;   RCSTA<ferr>=0           ; No framing error
        ;   RCSTA<oerr>=0           ; No overrun error
        ;   RCSTA<rx9D>=0           ; 9th bit is not used
        ;   To receive:  b'10010000'
        movlw   B'00010000'
        movwf   RCSTA
        ; Clear the UART transmit and receive registers
        clrf    TXREG
        clrf    RCREG
        ; CCP2CON=0 disables C/C/P module (affects portC)
        clrf    CCP2CON
        ; ADCON0=0 disables the A/D module (affects portA)
        clrf    ADCON0
        ;
        ; Select register bank 1
        bsf     STATUS,page0
        bcf     STATUS,page1
        ; Configure OPTION_REG (affects portB)
        ; OPTION_REG<7>=1           ; Disable PortB pull-up resistors
        ; OPTION_REG<6>=0           ; RB0 interrupt on falling edge
        ; OPTION_REG<5>=0           ; Internal clock (Fosc/4) drives Timer0
```

```
; OPTION_REG<4>=0           ; Increment Timer0 on low-to-high
; OPTION_REG<3>=0           ; Assign prescalar to Timer0
; OPTION_REG<2-0>=100       ; Set Timer0 prescalar 32:1
movlw   0x84
movwf   OPTION_REG
; Configure portA for output
clrf    TRISA
; Configure all pins of portB for output
movlw   0x00
movwf   TRISB
; Configure RC7 for input; all other pins of portC for output
movlw   0x80
movwf   TRISC
; PIE1=0 disables all peripheral interrupt activity
clrf    PIE1
; PCON<4-5>=0 disables wake-up and brown-out resets
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
; Configure TXSTA: UART transmit status and control register
; Default configuration is for both Transmit and Receive modes enabled,
; with the UART module not enabled yet.
; Note: For 9600 baud in asynchronous mode when using a 20MHz clock, the
; best baud rate setting is achieved with: (i) brgh=1, (ii) brg16=1 and
; spbrg=d'520'.  The resulting timing error is -0.03%.
; Note: For 19,200 baud with a 20MHz clock, set (i) brgh=1, (ii) brg16=1
; and spbrg=d'259'.  The resulting error is +0.16%.
;   TXSTA<csrc>=0           ; Don't care (Asynchronous mode)
;   TXSTA<tx9>=0            ; 8-bit transmission
;   TXSTA<txen>=1/0         ; Transmit enabled/disabled
;   TXSTA<synch>=0          ; Asynchronous mode
;   TXSTA<sendb>=0          ; Synch break transmission completed
;   TXSTA<brgh>=1           ; High-speed baud rate selected
;   TXSTA<trmt>=0           ; TSR register is full
;   TXSTA<tx9d>=0           ; 9th bit is not used
;   To transmit: b'10100100'
movlw   B'00100100'
movwf   TXSTA
; Configure SPBRG and SPBRGH for 9600 baud as described above, where
; d'520' = 0x0208, or configure for 19,200 baud with d'259' = 0x0103.
movlw   0x03
movwf   SPBRG
movlw   0x01
movwf   SPBRGH
; PSTRCON=0 zeroes the steering pin assignments (affects portC)
clrf    PSTRCON
;
; Select register bank 2
bcf     STATUS,page0
bsf     STATUS,page1
; CM1CON0=0 disables Comparator 1 module (affects portA)
clrf    CM1CON0
; CM2CON0=0 disables Comparator 2 module (affects portA)
clrf    CM2CON0
```

```
        ; CM2CON1=0 disables Comparator 2 module (affects portA and portB)
        clrf    CM2CON1
        ;
        ; Select register bank 3
        bsf     STATUS,page0
        bsf     STATUS,page1
        ; Configure BAUDCTL: Baud rate control register
        ;   BAUDCTL<abdovf>=0       ; Read-only
        ;   BAUDCTL<rcidl>=0        ; Read-only
        ;   BAUDCTL<sckp>=0         ; Transmit non-inverted
        ;   BAUDCTL<brg16>=1        ; Use 16-bit baud rate generator
        ;   BAUDCTL<wue>=0          ; Receiver is operating normally
        ;   BAUDCTL<abden>=0        ; Disable auto-baud detect
        movlw   0x08
        movwf   BAUDCTL
        ; Ensure that all pins are digital I/O, not analogue
        clrf    ANSEL               ; Set portA pins as digital I/O
        clrf    ANSELH              ; Set portB pins as digital I/O
        ;
        ; Select register bank 0 for main program
        bcf     STATUS,page0
        bcf     STATUS,page1
;
; ******************************************************************************
; Block E - Initialization of user registers
; ******************************************************************************
;
InitializeUserRegisters
        ; Clear all Display LEDs
        clrf    portBmirror
        movf    portBmirror,w
        movwf   portB
        ; Clear all Error LEDs, but be careful not to touch RC7 and RC8
        movf    portC,w
        andlw   0xC0
        movwf   portC
        ; For the time being, the ISR should ignore all types of interrupts
        clrf    IntDoISR
        ; Since no interrupts have occurred, clear the IntTrig register
        clrf    IntTrig
;
; ******************************************************************************
; Block F - Initialize communication with the Host PC
; ******************************************************************************
;
        ; Wait 100ms before starting the initialization process.  This will
        ; de-bounce the power-on or reset procedure and prevent the
        ; initialization process from re-starting, something which will confuse
        ; the Host PC.
        call    del100ms
        ; Enable the UART module (Note that all interrupts are disabled)
        bsf     RCSTA,spen
        ; Make sure the receive buffer is completely empty
        movf    RCREG,w
        movf    RCREG,w
ICWHP1
        ; Wait until a byte is received from the Host PC
```

```
        btfss   PIR1,rcif           ; Bit=1 when a byte has been received
        goto    ICWHP1              ; Goto since nothing has been received
        ; Read the error flags in register RCSTA
        btfsc   RCSTA,ferr          ; Bit=1 after a framing error
        goto    Error_FrameErr      ; Goto since a framing error occurred
        btfsc   RCSTA,oerr          ; Bit=1 after an overrun error
        goto    Error_OverrunErr    ; Goto since an overrun error occurred
        ; No errors were detected, so read the byte received
        movf    RCREG,w
        movwf   HostInByte          ; Save the byte in register HostInByte
        ; Display the byte received on portB
        movwf   portB
        ; Test if the byte received is ping byte 0xF5.  It is assumed, but it's
        ; not vital, that the Host PC will be turned on before this PIC.  When
        ; the Host PC is turned on, it may send some spurious signals to the
        ; serial port.  The following code ignores all bytes received which are
        ; not the ping byte 0xF5.  Of course, if one of the spurious bytes
        ; happens to be 0xF5, then the initialization process will likely stall
        ; and the Host PC will prompt he User to re-start this PIC.
        xorlw   0xF5
        btfss   STATUS,zero         ; Z=1 if the byte is the ping byte
        goto    ICWHP1              ; Wait for a correct ping byte
        ; Transmit ping response byte 0xF6 to the Host PC
        movlw   0xF6
        movwf   TXREG
        ; Display the byte sent on portB.  Note: It takes one instruction cycle
        ; before the PIR1 register can be tested for completion.  It is not
        ; necessary to include a nop instruction since the following movwf
        ; instruction takes the same length of time.
        movwf   portB
ICWHP2
        ; Wait until the byte has been transferred out
        btfss   PIR1,txif           ; Bit=1 when the byte has been transferred
        goto    ICWHP2              ; Goto since the transfer is not complete
        ; The Host PC waits 10ms after it receives ping response byte 0xF6.
        ; That delay gives this PIC time to re-configure to receive packets
        ; instead of individual bytes.
        ; Wait 1ms in case the Host PC happens to send another ping byte 0xF5
        ; before it was able to process this PIC's ping response byte.
        call    del1ms
        ; Just in case the Host PC does send another ping byte or two before it
        ; can process this PIC's ping response byte, make sure the receive
        ; buffer is completely empty.
        movf    RCREG,w
        movf    RCREG,w
        ; Set the number of bytes received in the first packet to zero.  Note
        ; that the ISR re-sets this number to zero just after receiving the
        ; third byte of a packet.  Therefore, the following instruction does not
        ; need to be repeated once this PIC begins to receive complete packets.
        clrf    HostInNumByte
        ; Clear the IntTrig receive-complete interrupt flag in preparation for
        ; receiving the first packet
        bcf     IntTrig,HostPktRecd
        ; Record that we want the ISR to start processing reception from the
        ; Host PC
        bsf     IntDoISR,HostPktRecd
        ; Enable UART receive-complete interrupts.  Do not enable UART transmit-
```

```
        ; complete interrupts.
        bsf     STATUS,page0        ; Select register bank 1
        bsf     PIE1,rcie           ; Enable UART receive-complete interrupts
        bcf     STATUS,page0        ; Re-select register bank 0
        bsf     INTCON,peie         ; Enable peripheral interrupts
        bsf     INTCON,gie          ; Enable global interrupts
ICWHP3
        ; Wait until the first packet is received from the Host PC
        btfss   IntTrig,HostPktRecd ; Bit=1 when a packet has been received
        goto    ICWHP3              ; Goto since no packet has been received
        ; Clear the IntTrig receive-complete interrupt flag in preparation for
        ; the next packet
        bcf     IntTrig,HostPktRecd
        ; Display the low byte of the packet received on portB
        movf    HostInPktL,w
        movwf   portB
        ; Test if the packet received is ping packet 0xF00505
        movf    HostInPktH,w
        xorlw   0xF0
        btfss   STATUS,zero         ; Z=1 if the high byte is 0xF0
        goto    Error_NotPingPkt    ; Goto since not the correct ping packet
        movf    HostInPktM,w
        xorlw   0x05
        btfss   STATUS,zero         ; Z=1 if the middle byte is 0x05
        goto    Error_NotPingPkt    ; Goto since not the correct ping packet
        movf    HostInPktL,w
        xorlw   0x05
        btfss   STATUS,zero         ; Z=1 if the low byte is 0x05
        goto    Error_NotPingPkt    ; Goto since not the correct ping packet
        ; Set up the ping response packet 0xF00606
        movlw   0xF0
        movwf   HostOutPktH
        movlw   0x06
        movwf   HostOutPktM
        movlw   0x06
        movwf   HostOutPktL
        movwf   portB               ; Display the Low byte on portB
        ; Set the number of bytes already sent in the first packet to zero.
        ; Note that the ISR re-sets this number to zero just after sending the
        ; third byte of a packet.  Therefore, the following instruction does not
        ; need to be repeated once this PIC begins to send complete packets.
        clrf    HostOutNumByte
        ; Clear the IntTrig transmit-complete interrupt flag in preparation for
        ; sending the first packet
        bcf     IntTrig,HostPktSent
        ; Send the first (High) byte.  The ISR will send the last two bytes
        ; automatically.
        movf    HostOutPktH,w
        movwf   TXREG
        ; Record that we want the ISR to start processing interrupts arising
        ; from the transmission of three bytes to the Host PC.  This flag is
        ; turned off by the ISR once it has sent the third byte.
        bsf     IntDoISR,HostPktSent
        ; Enable UART transmit-complete interrupts.  Note that txie interrupts
        ; may be enabled only after loading register TXREG has started the
        ; process.  If interrupts are enabled before moving a byte into register
        ; TXREG, an interrupt will fire immediately.
```

```
        bsf     STATUS,page0        ; Select register bank 1
        bsf     PIE1,txie           ; Enable UART transmit-complete interrupts
        bcf     STATUS,page0        ; Re-select register bank 0
ICWHP4
        ; Wait until the entire packet 0xF00606 has been transferred out
        btfss   IntTrig,HostPktSent
        goto    ICWHP4
        ; Clear the IntTrig transmit-complete interrupt flag in preparation for
        ; the next packet
        bcf     IntTrig,HostPktSent
        ; Initialization is complete; start the main program
;
; ******************************************************************************
; Block G - Main program
; ******************************************************************************
;
MainProgram

ReceiveFromHost
        ; Wait until the next packet is received from the Host PC
        btfss   IntTrig,HostPktRecd
        goto    ReceiveFromHost
        ; Clear the IntTrig receive-complete interrupt flag in preparation for
        ; the next packet
        bcf     IntTrig,HostPktRecd
        ; Display the low byte of the packet on portB
        movf    HostInPktL,w
        movwf   portB
        ; Complement the packet received
        comf    HostInPktH,w
        movwf   HostOutPktH
        comf    HostInPktM,w
        movwf   HostOutPktM
        comf    HostInPktL,w
        movwf   HostOutPktL
        ; Send the complemented packet
        movf    HostOutPktH,w
        movwf   TXREG
        ; Record that we want the ISR to start processing interrupts arising
        ; from the transmission of three bytes to the Host PC.  This flag is
        ; turned off by the ISR once it has sent the third byte.
        bsf     IntDoISR,HostPktSend
        ; Enable UART transmit-complete interrupts.  Note that txie interrupts
        ; may be enabled only after loading register TXREG has started the
        ; process.  If interrupts are enabled before moving a byte into register
        ; TXREG, an interrupt will fire immediately.
        bsf     STATUS,page0        ; Select register bank 1
        bsf     PIE1,txie           ; Enable UART transmit-complete interrupts
        bcf     STATUS,page0        ; Re-select register bank 0
MP1
        ; Wait until the entire packet has been transferred out
        btfss   IntTrig,HostPktSent
        goto    MP1
        ; Clear the IntTrig transmit-complete interrupt flag in preparation for
        ; the next packet
        bcf     IntTrig,HostPktSent
        ; Start waiting to receive another packet
```

```
        goto    ReceiveFromHost
;
; ******************************************************************************
; Block H - Subroutine Error_Flash flashes a unique non-zero error code on the
;           four low-order bits of portC.  The error code is lit up for one-half
;           second, alternating with half-second blanks.  The error code is
;           passed into this subroutine in User register ErrorCode.  Care must
;           be taken when using portC, which is also used for UART
;           communication.
; Error codes while receiving a byte from the Host:-
;   Code 0x01: Framing error detected by UART module
;   Code 0x02: Overrun error detected by UART module
; Error codes while receiving a packet from the Host PC:-
;   Code 0x03: First packet is not ping packet 0xF00505
; ******************************************************************************
;
Error_FrameErr
        movlw   0x01
        movwf   ErrorCode
        goto    Error_Flash
Error_OverrunErr
        movlw   0x02
        movwf   ErrorCode
        goto    Error_Flash
Error_NotPingPkt
        movlw   0x03
        movwf   ErrorCode
        goto    Error_Flash
Error_Flash
EF1
        movwf   portC
        andlw   0xC0
        xorwf   ErrorCode,w
        movwf   portC
        call    del500ms
        movwf   portC
        andlw   0xC0
        movwf   portC
        call    del500ms
        goto    EF1
;
; ******************************************************************************
; Block I - Miscellaneous and timing subroutines
; Subroutines:-
;   del1us - timed delay of exactly 1.00us
;   del10us - timed delay of exactly 10.0us
;   del50us - timed delay of exactly 50us
;   del100us - timed delay of exactly 100us
;   del1ms - timed delay of approximately 1ms
;   del10ms - timed delay of approximately 10ms
;   del100ms - timed delay of approximately 100ms
;   del500ms - timed delay of approximately 500ms
; ******************************************************************************
;
del1us
; This subroutine is a timed 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 cycles.  The
; "return" takes 2 instruction cycles.  The "nop" takes 1 instruction cycle.
        nop
        return
;
del10us
; This subroutine is a timed delay of exactly 10 microseconds, including the
; invoking "call".  This is equal to 50 instruction cycles at 20MHz.
        call    del1us                  ; 5 cycles
        movlw   0x0A                    ; 1 cycle; 0x0A = d'10'
        movwf   tempDus                 ; 1 cycle
D10us
        nop                             ; 10 cycles
        decfsz  tempDus,f               ; 9 interim tests + 2 final = 11 cycles
        goto    D10us                   ; 9 x 2 cycles = 18 cycles
        return                          ; 2 cycles
;
del50us
; This subroutine is a timed 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 final = 62 cycles
        goto    D50us                   ; 60 x 2 cycles = 120 cycles
        return                          ; 2 cycles
;
del100us
; This subroutine is a timed 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 final = 99 cycles
        goto    D100us                  ; 97 x 2 cycles = 194 cycles
        return                          ; 2 cycles
;
del1ms
; This subroutine is a timed delay of about one millisecond.  It calls
; subroutine del100us() ten times.
        call    del100us
        call    del100us
        call    del100us
        call    del100us
        call    del100us
        call    del100us
        call    del100us
        call    del100us
```

```
        call    del100us
        call    del100us
        return
;
del10ms
; This subroutine is a timed delay of about ten milliseconds.  It calls
; subroutine delay100us() ten times.
        call    del1ms
        call    del1ms
        call    del1ms
        call    del1ms
        call    del1ms
        call    del1ms
        call    del1ms
        call    del1ms
        call    del1ms
        call    del1ms
        return
;
del100ms
; This subroutine is a timed delay of about 100 milliseconds.  It calls
; subroutine del10ms() ten times.
        call    del10ms
        call    del10ms
        call    del10ms
        call    del10ms
        call    del10ms
        call    del10ms
        call    del10ms
        call    del10ms
        call    del10ms
        call    del10ms
        return
;
del500ms
; This subroutine is a timed delay of about 500 milliseconds.  It calls
; subroutine del100ms() five times.
        call    del100ms
        call    del100ms
        call    del100ms
        call    del100ms
        call    del100ms
        return
;
        END                                 ; end assembly
```

Appendix "C"

Visual Basic program for the Host PC

Main form Form1.vb

```
Option Strict On
Option Explicit On

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

Public Class Form1
    Inherits System.Windows.Forms.Form

    Public Sub New()
        ' Set parameters of the screen and the display
        Name = "Main"
        Text = "PIC-to-Host Comm Test"
        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
    End Sub

    ' ******************************************************************************
    ' *** Controls for program flow ************************************************
    ' ******************************************************************************
    Private ButtonRow As Int32 = 5
    Private TimeRow As Int32 = 40
    Private CommStatusRow As Int32 = TimeRow + 40

    Private WithEvents buttonStart As New System.Windows.Forms.Button With
        {.Size = New Drawing.Size(100, 30), .Location = New Drawing.Point(5, ButtonRow),
        .Text = "Start", .TextAlign = ContentAlignment.MiddleCenter,
        .Font = New Font("Arial", 14), .Visible = True, .Parent = Me}

    Private Sub buttonStart_Click() Handles buttonStart.MouseClick
        PICComm.InitializeCommWithCommPICA()
        PICComm.TestProgram()
    End Sub

    Private WithEvents buttonExit As New System.Windows.Forms.Button With
        {.Size = New Drawing.Size(100, 30), .Location = New Drawing.Point(110, ButtonRow),
        .Text = "Exit", .TextAlign = ContentAlignment.MiddleCenter,
        .Font = New Font("Arial", 14), .Visible = True, .Parent = Me}
```

```
Public Sub buttonExit_Click() Handles buttonExit.MouseClick
    ' Close the serial port
    Try
        If (CommPICASerialPort.IsOpen = True) Then
            CommPICASerialPort.Close()
            CommPICASerialPort.Dispose()
        End If
    Catch
    End Try
    ' Exit from the application
    Application.Exit()
    End
End Sub

' ******************************************************************************************
' *** Controls for start and stop times *****************************************************
' ******************************************************************************************

Private labelStartTime As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(140, 30), .Location = New Drawing.Point(5, TimeRow),
    .Text = "Start time", .TextAlign = ContentAlignment.MiddleLeft,
    .Font = New Font("Arial", 14), .Visible = True, .Parent = Me}

Public tbStartTime As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(140, 30), .Location = New Drawing.Point(150, TimeRow),
    .Text = "", .TextAlign = ContentAlignment.MiddleLeft, .BackColor = Color.White,
    .Font = New Font("Arial", 14), .BorderStyle = BorderStyle.FixedSingle,
    .Visible = True, .Parent = Me}

Private labelStopTime As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(140, 30), .Location = New Drawing.Point(330, TimeRow),
    .Text = "Stop time", .TextAlign = ContentAlignment.MiddleLeft,
    .Font = New Font("Arial", 14), .Visible = True, .Parent = Me}

Public tbStopTime As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(140, 30), .Location = New Drawing.Point(475, TimeRow),
    .Text = "", .TextAlign = ContentAlignment.MiddleLeft, .BackColor = Color.White,
    .Font = New Font("Arial", 14), .BorderStyle = BorderStyle.FixedSingle,
    .Visible = True, .Parent = Me}

' ******************************************************************************************
' *** Controls for communication status *****************************************************
' ******************************************************************************************

Private labelBreakLine1 As New System.Windows.Forms.Panel With
    {.Size = New Drawing.Size(610, 2), .Location = New Drawing.Point(5, TimeRow + 35 - 1),
    .BackColor = Color.Black, .Visible = True, .Parent = Me}

Private labelCommStatus As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(200, 30), .Location = New Drawing.Point(5, CommStatusRow),
    .Text = "Communication status", .TextAlign = ContentAlignment.MiddleLeft,
    .Font = New Font("Arial", 14), .Visible = True, .Parent = Me}

Private labelCommPICAStatusHdr As New System.Windows.Forms.Label With
    {.Size = New Drawing.Size(150, 30), .Location = New Drawing.Point(5, CommStatusRow + 35),
    .Text = "CommPICA", .TextAlign = ContentAlignment.MiddleLeft,
    .Font = New Font("Arial", 14), .Visible = True, .Parent = Me}
```

```
    Public WithEvents labelCommPICAStatus As New System.Windows.Forms.Label With
        {.Size = New Drawing.Size(460, 50), .Location = New Drawing.Point(160, CommStatusRow + 35),
        .Text = "", .TextAlign = ContentAlignment.TopLeft, .BackColor = Color.White,
        .Font = New Font("Arial", 14), .BorderStyle = BorderStyle.FixedSingle,
        .Visible = True, .Parent = Me}

    Private labelCommPICATestHdr As New System.Windows.Forms.Label With
        {.Size = New Drawing.Size(150, 30), .Location = New Drawing.Point(5, CommStatusRow + 90),
        .Text = "CommPICA test", .TextAlign = ContentAlignment.MiddleLeft,
        .Font = New Font("Arial", 14), .Visible = True, .Parent = Me}

    Public WithEvents labelCommPICATest As New System.Windows.Forms.Label With
        {.Size = New Drawing.Size(460, 50), .Location = New Drawing.Point(160, CommStatusRow + 90),
        .Text = "", .TextAlign = ContentAlignment.TopLeft, .BackColor = Color.White,
        .Font = New Font("Arial", 14), .BorderStyle = BorderStyle.FixedSingle,
        .Visible = True, .Parent = Me}

End Class
```

Module PICComm.vb

```
Option Strict On
Option Explicit On

' This module contains all of the routines needed to communicate with CommPICA.

' List of subroutines:-
' RS-232 communication with CommPICA
'   InitializeCommWithCommPICA()
'   TestProgram()
'   BWCommPICASendPkt_DoWork(Int32)
'   BWCommPICASendPkt_RunWorkerCompleted()
'   CommPICAAsyncReadBytes()
'   CommPICAGetNextPacket(Int32)

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

Public Module PICComm

    ' Serial port for CommPICA
    Public WithEvents CommPICASerialPort As SerialPort
    Private DefaultCommPICASerialPort As String = "COM5"

    ' One-based buffer which stores ***ALL*** bytes received from CommPICA.  NumBytesAdded
    ' is the number of bytes placed into the buffer by the read-procedure.  NumBytesUsed
    ' is the number of bytes which have been taken out of the buffer by the packet-
    ' generation-procedure.
    Private CommPICAInBytes(3 * 256 * 256 * 256) As Byte
    Private CommPICAInBytes_NumBytesAdded As Int32
    Private CommPICAInBytes_NumBytesUsed As Int32
```

```
' A packet to be sent to CommPICA
Private CommPICAPktToSend As Int32

' Background worker for transmission
Private WithEvents BWCommPICASendPkt As BackgroundWorker

' Flag to mark completion of transmission
Private CommPICASendPktComplete As Boolean = False

' Variables to display the start and stop times
Private StartDateTime As Date
Private StartTimeStr As String
Private StopDateTime As Date
Private StopTimeStr As String

Public Sub InitializeCommWithCommPICA()
    ' This subroutine runs through all of the steps to initialize communication with
    ' CommPICA.  Since communication with CommPICA is vital, this is a "blocking"
    ' subroutine which does not allow anything else to happen until communication
    ' has been established successfully.
    ' Step #1: ' Close any CommPICA serial port which is already open
    Try
        CommPICASerialPort.Close()
        CommPICASerialPort.Dispose()
        Threading.Thread.Sleep(25)
        Application.DoEvents()
    Catch
    End Try
    ' Step #2: Set the properties for CommPICA before it is opened
    CommPICASerialPort = New SerialPort
    CommPICASerialPort.PortName = DefaultCommPICASerialPort
    CommPICASerialPort.BaudRate = 19200
    CommPICASerialPort.DataBits = 8
    CommPICASerialPort.StopBits = StopBits.One
    CommPICASerialPort.Parity = Parity.None
    CommPICASerialPort.Handshake = Handshake.None
    CommPICASerialPort.ReadBufferSize = 2048
    CommPICASerialPort.WriteBufferSize = 2048
    CommPICASerialPort.ReceivedBytesThreshold = 1
    CommPICASerialPort.ReadTimeout = -1
    ' Step #3: Check every 25ms until a USB-to-serial port adapter cable is inserted
    ' into the default CommPICA USB socket.
    Do
        Try
            ' Try to open the selected serial port
            CommPICASerialPort.Open()
            ' Empty CommPICA's input and output buffers
            CommPICASerialPort.DiscardInBuffer()
            CommPICASerialPort.DiscardOutBuffer()
            Exit Do
        Catch
            Form1.labelCommPICAStatus.Text =
                "Could not open CommPICA serial port." & vbCrLf &
                "Make sure that CommPICA is plugged in."
            Form1.labelCommPICAStatus.BackColor = Color.HotPink
            Form1.labelCommPICAStatus.Refresh()
        End Try
```

```
        Threading.Thread.Sleep(25)
        Application.DoEvents()
    Loop
    Form1.labelCommPICAStatus.Text = ""
    Form1.labelCommPICAStatus.BackColor = Color.White
    Form1.labelCommPICAStatus.Refresh()
    ' Step 4: Check every 25ms until CommPICA is powered up
    Do
        If (CommPICASerialPort.CtsHolding = True) Then
            Exit Do
        Else
            Form1.labelCommPICAStatus.Text =
                "CTS from CommPICA is low." & vbCrLf &
                "Make sure that CommPICA is turned on."
            Form1.labelCommPICAStatus.BackColor = Color.HotPink
            Form1.labelCommPICAStatus.Refresh()
        End If
        Threading.Thread.Sleep(25)
        Application.DoEvents()
    Loop
    Form1.labelCommPICAStatus.Text = ""
    Form1.labelCommPICAStatus.BackColor = Color.White
    Form1.labelCommPICAStatus.Refresh()
    ' Step #6: Send ping byte &HF5 to CommPICA
    Dim CommPICAOutputByte(0) As Byte
    Try
        CommPICAOutputByte(0) = &HF5
        CommPICASerialPort.Write(CommPICAOutputByte, 0, 1)
    Catch ex As Exception
        ' Treat failure to send as a fatal error.  Display an error message for
        ' five seconds and then abort.
        Form1.labelCommPICAStatus.Text =
            "Failed to send ping byte &&HF5 to CommPICA." & vbCrLf &
            "ex = " & ex.ToString
        Form1.labelCommPICAStatus.BackColor = Color.HotPink
        Form1.labelCommPICAStatus.Refresh()
        Threading.Thread.Sleep(5000)
        Application.Exit()
        Exit Sub
    End Try
    ' Step #7: Every 25ms, check to see if CommPICA has sent anything.  If CommPICA
    ' has not sent anything, then wait 1ms and then resend ping byte &HF5.  If
    ' CommPICA has sent a byte, check to see if it is the ping response byte &HF6.
    ' If it is, then proceed to the next step.  Otherwise, wait 1ms and resend ping
    ' byte &HF5.
    Dim CommPICAInputByte(0) As Int32
    Do
        'Wait 25ms
        Threading.Thread.Sleep(25)
        Application.DoEvents()
        ' Check if a reply has been received
        If (CommPICASerialPort.BytesToRead >= 1) Then
            CommPICAInputByte(0) = CommPICASerialPort.ReadByte
            ' Check if the reply is ping response byte &HF6
            If (CommPICAInputByte(0) = &HF6) Then
                Exit Do
            Else
                ' If the byte received is not the ping response byte &HF6, then
```

```
            ' display the byte received.
            Form1.labelCommPICAStatus.Text =
                "CommPICA responded to ping byte with &&H" &
                CommPICAInputByte(0).ToString("X2")
            Form1.labelCommPICAStatus.BackColor = Color.HotPink
            Form1.labelCommPICAStatus.Refresh()
        End If
    Else
        Form1.labelCommPICAStatus.Text =
            "CommPICA has not responded to ping byte &&HF5."
        Form1.labelCommPICAStatus.BackColor = Color.HotPink
        Form1.labelCommPICAStatus.Refresh()
    End If
    ' Resend ping byte &HF5 to CommPICA
    Try
        CommPICAOutputByte(0) = &HF5
        CommPICASerialPort.Write(CommPICAOutputByte, 0, 1)
    Catch ex As Exception
        ' Treat failure to send as a fatal error.  Display an error message for
        ' five seconds and then abort.
        Form1.labelCommPICAStatus.Text =
            "Failed to send ping byte &&HF5 to CommPICA." & vbCrLf &
            "ex = " & ex.ToString
        Form1.labelCommPICAStatus.BackColor = Color.HotPink
        Form1.labelCommPICAStatus.Refresh()
        Threading.Thread.Sleep(5000)
        Application.Exit()
        Exit Sub
    End Try
Loop
' Step #8: Wait 10ms for CommPICA to set the interrupt enables for packet
' reception.  Use this period to start the asynchronous reader.
' Empty CommPICA's input and output buffers
CommPICASerialPort.DiscardInBuffer()
CommPICASerialPort.DiscardOutBuffer()
' Set the timeout for the base stream
CommPICASerialPort.BaseStream.ReadTimeout = SerialPort.InfiniteTimeout
' Start the asynchronous reader
CommPICAAsyncReadBytes()
' Wait 10ms
Threading.Thread.Sleep(10)
' Step #9: Send the ping packet &HF00505 to CommPICA
CommPICAPktToSend = &HF00505
' Update the GUI
Form1.labelCommPICAStatus.Text =
    "Sending ping packet &&HF00505 to CommPICA."
Form1.labelCommPICAStatus.BackColor = Color.White
Form1.labelCommPICAStatus.Refresh()
' Clear the completion flag CommPICASendPktComplete.  When the transmission
' is complete, the RunWorkerCompleted subroutine will set the flag high.
CommPICASendPktComplete = False
' Start the transmit background worker
BWCommPICASendPkt = New BackgroundWorker
BWCommPICASendPkt.RunWorkerAsync(CommPICAPktToSend)
' Wait for the transmission to be completed
Do
    If (CommPICASendPktComplete = True) Then
        CommPICASerialPort.DiscardOutBuffer()
```

```
            Exit Do
        Else
            Application.DoEvents()
        End If
    Loop
    ' Step #11: Wait for ping response packet &HF00606 to be received
    Do
        Application.DoEvents()
        Dim lNextPacket As Int32
        If (GetNextPacket(lNextPacket) = True) Then
            Form1.labelCommPICATest.Text = lNextPacket.ToString("X6")
            Form1.labelCommPICATest.Refresh()
            ' Test if lNextPacket is the ping response packet &HF00606
            If (lNextPacket = &HF00606) Then
                Form1.labelCommPICAStatus.Text = "CommPICA is in sync."
                Form1.labelCommPICAStatus.BackColor = Color.LightGreen
                Form1.labelCommPICAStatus.Refresh()
                Exit Do
            Else
                Form1.labelCommPICAStatus.Text =
                    "CommPICA responded to the ping packet with &&H" &
                    lNextPacket.ToString("X6")
                Form1.labelCommPICAStatus.BackColor = Color.HotPink
                Form1.labelCommPICAStatus.Refresh()
                Threading.Thread.Sleep(5000)
                Application.Exit()
                Exit Sub
            End If
        Else
            Form1.labelCommPICAStatus.Text =
                "CommPICA has not responded to ping packet &&HF00505."
            Form1.labelCommPICAStatus.BackColor = Color.HotPink
            Form1.labelCommPICAStatus.Refresh()
        End If
    Loop
End Sub

Public Sub TestProgram()
    ' Display the starting time
    StartDateTime = Date.Now
    StartTimeStr = StartDateTime.ToString("h:mm:ss tt")
    Form1.tbStartTime.Text = StartTimeStr
    ' Initialize the first packet to be sent
    CommPICAPktToSend = &H000000
    Do
        ' Clear the completion flag CommPICASendPktComplete.  When the transmission
        ' is complete, the RunWorkerCompleted subroutine will set the flag high.
        CommPICASendPktComplete = False
        ' Send the packet
        BWCommPICASendPkt.RunWorkerAsync(CommPICAPktToSend)
        ' Wait for the transmission to be completed
        Do
            If (CommPICASendPktComplete = True) Then
                Exit Do
            Else
                Application.DoEvents()
            End If
        Loop
```

```
        ' Wait for a packet to be received
        Dim lPacketReceived As Int32
        Do
            If (GetNextPacket(lPacketReceived) = True) Then
                Exit Do
            Else
                Application.DoEvents()
            End If
        Loop
        ' Validate the packet received
        Dim XORResult As Int32 = CommPICAPktToSend Xor lPacketReceived
        If (XORResult <> &H00FFFFFF) Then
            MsgBox(
                "Exchange Error:" & vbCrLf &
                "Packet sent = &&H" & CommPICAPktToSend.ToString("X6") & vbCrLf &
                "Packet received = &&H" & lPacketReceived.ToString("X6"))
        End If
        ' Display the packet sent
        Form1.labelCommPICATest.Text =
            "Packet sent = &&H" & CommPICAPktToSend.ToString("X6")
        Form1.labelCommPICATest.BackColor = Color.Wheat
        Form1.labelCommPICATest.Refresh()
        ' Increment the packet value
        CommPICAPktToSend = CommPICAPktToSend + 1
        ' Test for completion
        If (CommPICAPktToSend >= &H080000) Then
            ' Display the stopping time
            StopDateTime = Date.Now
            StopTimeStr = StopDateTime.ToString("h:mm:ss tt")
            Form1.tbStopTime.Text = StopTimeStr
            MsgBox("Test is complete.")
            Form1.buttonExit_Click()
        End If
    Loop
End Sub

Private Sub BWCommPICASendPkt_DoWork(
    ByVal sender As System.Object,
    ByVal e As System.ComponentModel.DoWorkEventArgs) Handles _
    BWCommPICASendPkt.DoWork
    ' Retrieve the packet to send
    Dim lCommPICAPktToSend As Int32 = CInt(e.Argument)
    ' Parse the packet into three bytes
    Dim lOutPkt(2) As Byte
    lOutPkt(0) = CByte((lCommPICAPktToSend And &H00FF0000) >> 16)
    lOutPkt(1) = CByte((lCommPICAPktToSend And &H0000FF00) >> 8)
    lOutPkt(2) = CByte(lCommPICAPktToSend And &H000000FF)
    ' Start to send the packet
    CommPICASerialPort.Write(lOutPkt, 0, 3)
End Sub

Private Sub BWCommPICASendPkt_RunWorkerCompleted(
    ByVal sender As Object,
    ByVal e As System.ComponentModel.RunWorkerCompletedEventArgs) Handles _
    BWCommPICASendPkt.RunWorkerCompleted
    ' Set the completion flag
    CommPICASendPktComplete = True
End Sub
```

```
    Private Async Sub CommPICAAsyncReadBytes()
        ' Initialize the number of bytes added and used
        CommPICAInBytes_NumBytesAdded = 0
        CommPICAInBytes_NumBytesUsed = 0
        ' Infinite Do-loop runs continuously during the application
        Do While (CommPICASerialPort.IsOpen)
            ' Create a temporary buffer
            Dim BytesToRead As Int32 = 1024
            Dim ReceiveBuffer(1024) As Byte
            Dim NumBytesRead As Int32 = 0
            Try
                ' Read all available bytes -- returns zero if EOF
                NumBytesRead =
                    Await CommPICASerialPort.BaseStream.ReadAsync(
                    ReceiveBuffer, 0, BytesToRead, CancellationToken.None)
                If (NumBytesRead > 0) Then
                    ' Add the bytes to the buffer
                    For I As Int32 = 1 To NumBytesRead Step 1
                        CommPICAInBytes_NumBytesAdded =
                            CommPICAInBytes_NumBytesAdded + 1
                        CommPICAInBytes(CommPICAInBytes_NumBytesAdded) =
                            ReceiveBuffer(I - 1)
                    Next I
                End If
            Catch ex As Exception
                MsgBox(
                    "Error in CommPICAAsyncReadBytes()" & vbCrLf &
                    ex.ToString())
            End Try
        Loop
    End Sub

    Private Function GetNextPacket(ByRef lNextPacket As Int32) As Boolean
        ' This function returns True if there are enough unprocessed bytes in buffer
        ' CommPICAInBytes to create a packet.  The packet is returned in ByRef variable
        ' lNextPacket, but is not meaningful if the function returns False.
        ' Calculate the number of bytes which have not been processed yet
        Dim lNumUnprocessedBytes As Int32 =
            CommPICAInBytes_NumBytesAdded - CommPICAInBytes_NumBytesUsed
        ' Return failure unless there are at least three unprocessed bytes
        If (lNumUnprocessedBytes <= 2) Then
            Return False
        End If
        ' Grab the next three unprocessed bytes
        CommPICAInBytes_NumBytesUsed = CommPICAInBytes_NumBytesUsed + 1
        Dim lInByteH As Int32 = CommPICAInBytes(CommPICAInBytes_NumBytesUsed)
        CommPICAInBytes_NumBytesUsed = CommPICAInBytes_NumBytesUsed + 1
        Dim lInByteM As Int32 = CommPICAInBytes(CommPICAInBytes_NumBytesUsed)
        CommPICAInBytes_NumBytesUsed = CommPICAInBytes_NumBytesUsed + 1
        Dim lInByteL As Int32 = CommPICAInBytes(CommPICAInBytes_NumBytesUsed)
        ' Calculate the value of the packet
        lNextPacket = (lInByteH << 16) + (lInByteM << 8) + lInByteL
        ' Return success
        Return True
    End Function

End Module
```