Appendix C

Serial Protocol Subroutines – Visual Basic Version

This appendix covers the Visual Basic (VB) version of the Serial Protocol Subroutines. Because of the high level of commonality between the QuickBASIC version, covered in detail in Appendix B, and the VB version covered in this appendix, I'll mainly focus on the unique differences required with Visual Basic. For discussion regarding the remainder of code, that remains identical, please refer back to Appendix B. In fact, it is recommended everyone read through Appendix B before tackling Appendix C even though your plan might be to only make use of Visual Basic.

Because the Visual Basic language includes almost every statement that is provided in QuickBASIC and because both sets of subroutines are executing the same exact task, it's easy to understand the close similarity between the VB and QuickBASIC versions of the Serial Protocol Subroutines. Although the VB language also includes the GOSUB and RETURN statements, I've chosen not to provide a GOSUB version of the Serial Protocol Subroutines in VB. Basically when you get to the point of using VB, most, if not, all applications will take advantage of modular programming using CALL statements.

DIFFERENCES BETWEEN DOS AND WINDOWS-BASED OPERATION

The primary difference between the VB and QuickBASIC versions are the different statements used to pass data between the UART (see Appendix B) and the protocol subroutines. The QuickBASIC version, operating under DOS, uses a simple INP instruction to extract a data byte from the UART's receive data register and a simple OUT instruction to place a data byte into the UART's transmit data register. These access calls are basically just a couple of assembly instructions that pass data very quickly and effectively between an application program and a PC COM port. Fundamentally, DOS allows the user, via QuickBASIC, to get very close to the hardware, the PC's UART.

With the introduction of Windows 95, Microsoft stopped supporting the INP and OUT port function calls. Windows makes you work at a higher level while it, through its Application Programming Interface (API), performs the actual hardware manipulations. Basically, Windows operating with such programs as Visual Basic, Power Basic, Visual C++ and JAVA, keeps the user away from direct hardware access.

To adapt to this situation, VB makes use of "controls", one of which is called MSComm, that can be thought of as a small program module, whose entire job is to maintain the PC's serial ports for VB. It buffers data accumulated while the computer is off working on some other event and makes the data available to the VB program when the VB program is working on the I/O part of its code. MSComm operates as a separate interrupt creating event and therefore can "demand" time from the computer when it needs it. Basically, via the assistance of MSComm and many other factors, the Windows Operating System (OS) and the VB program operations complement each other.

SERIAL I/O BUFFERING REQUIREMENTS WHEN USING VISUAL BASIC

In Chapter 5 we saw how operation under Windows inserts a variety of interruptions thus preventing the smooth continuous flow desired by a real-time application program. In fact, nothing really prevents another program from operating in parallel with a VB application program. For example, if a Virus

Scanning Program is enabled when you are in the middle of an operating session, your C/MRI program's flow will be interrupted on a near continuous basis throughout the entire scanning process. The time interval between interrupts and the duration of each interrupt processing period can be on the order of several microseconds to many milliseconds. Multiplied out in time, these interruptions can become a significant portion of the normal cycle time associated with a C/MRI's real-time loop.

The basic premise of Windows is that the computer is so fast that it can carry out multi-tasking, thus appearing to perform multiple tasks or "events" simultaneously. This action can include "simultaneous operation" of word processing, printing, email, and everything else, including a C/MRI application program. Fundamentally, the computer time-slices between the different tasks as it tries to satisfy all the different requirements that are striving to "operate in parallel."

You might ask, "How can we do this when the C/MRI application needs dedicated response?" To help answer this question we need to understand that a C/MRI program operating under DOS spends a goodly portion of its time waiting for serial communications. For example, when the PC is sending data it waits for the UART's transmit buffer to empty before it can load another byte. When the PC is receiving data it continuously waits for each data byte to arrive. Windows simply says, "Let's use those available time slices to perform other tasks." Windows takes those time slices, as well as additional ones as it sees fit, to keep "everything running in parallel." For example, some of these tasks may include monitoring mouse movements, virus checking, mouse clicks, window displays and a variety of programs whose icons are located on your Windows task bar.

That's well and good but frequently, while non-C/MRI tasks are taking place, the C/MRI hardware continues to send data back to the PC. Without some special buffering, when multiple data bytes arrive during the period that other tasks are being processed, all but the last arrival byte are lost to the C/MRI program once it's reactivated. This puts the C/MRI application program out of synchronization with the C/MRI hardware.

When operating under strict DOS, the C/MRI application program stays in synchronization with the C/MRI hardware because the application program simply waits, in a tight loop, for each byte to be transmitted to and received from the C/MRI hardware. By contrast, with VB performing under Windows, the operation is said to be asynchronous meaning that the C/MRI application program runs in spurts as directed by Windows essentially independent of the arrival rate of bytes from the C/MRI hardware.

The underlying function of MSComm is to make the asynchronous operation appear as synchronous. MSComm operates as an independent interrupt driven event. Basically, the UART generates an interrupt each time it receives an incoming byte from the C/MRI hardware. Windows responds to this interrupt by activating MSComm which then retrieves the incoming byte from the UART and stores it into an input buffer array in memory. This action immediately clears the UART's internal input buffer register to receive the next input byte from the C/MRI hardware. This process is repeated with each new incoming byte.

Each time Windows activates the VB C/MRI application program for another slice of time and the application needs to read an input byte, it goes through MSComm, which retrieves the first available byte it has pre-stored into its input byte memory array buffer. This type of data buffering and retrieval is frequently referred to as FIFO standing for first-in-first-out.

Fundamentally, MSComm reads and stores the input bytes as they arrive from the C/MRI and the VB application program retrieves an input byte from the stored array whenever it needs to read an input byte. If an input byte is available, the application program takes that byte and runs. If there is no byte available, the application program simply loops, with possible interruptions from Windows, until a byte becomes available. Because of FIFO, the bytes are always being read in the order that they were received.

Things may be delayed slightly, because of the Windows induced interruptions, but everything stays in synchronization. As the VB C/MRI application program operates in its real-time loop, it first reads all the input bytes via MSComm. Then once the VB application program has all the input bytes in place, it calculates all the outputs based upon the inputs. Once all the output bytes are calculated the application program writes all the output bytes to the railroad via MSComm. Once all the output bytes are written, the program branches back to the beginning of the real-time loop to repeat the process ad infinitum.

It's vital that the above basic application loop sequencing stays intact. With VB and the MSComm buffering provisions, an "advanced programmer" can be tempted to implement multi-tasking within the application program. By this I mean letting the application program accept inputs whenever they arrive, stacking up outputs within MSComm and letting MSComm transmit them whenever it can, and continuously updating outputs based upon whatever the latest combination of inputs happens to be available. Such an approach creates an unjustified level of programming complexity and can lead to all sorts of C/MRI timing problems.

Just as an example, stacking the output bytes into the MSComm's output buffer, to be transmitted whenever MSComm can do so, **while at the same time** letting the C/MRI application program go forward on to its next real-time loop task is asking for trouble. The application program's next task is to send out a poll request to the C/MRI to read back inputs. It's very likely that the addressed PIC16F877 is still busy processing the bytes that it's continuing to receive from the PC via MSComm buffering of output bytes. Thus, the addressed PIC16F877 isn't available to respond to the poll request, collect input data and send it back to the PC. The conflict created by being asked to send back data when still receiving data results in errors detected within the 16F877 Microcontroller, causing the green status LED to flash an error code.

To keep a C/MRI system running smoothly, and correctly, it's extremely important to make sure that the actual PC hardware has completed sending all data bytes before it moves onward to requesting inputs.

Also, for data integrity sake, it's important that all the inputs are read in before starting to calculate outputs based upon inputs. Otherwise your output calculations can easily be based upon an inconsistent set of inputs resulting in miscalculated signal aspects.

The conclusions are threefold:

1. Using MSComm's capability to buffer inputs coming into the PC from the C/MRI hardware is vital to successful VB application programming.

2. Using MSComm's capability to buffer outputs going out to the C/MRI hardware is not required but is recommended to help speed up program response. However, in all cases (outputs buffered or not buffered), it is essential that the Serial Protocol Subroutines wait until all output bytes are transmitted out of the PC before allowing the VB application program to move forward to send out the poll request for receiving inputs.

3. Maintaining commonality between the real-time loop processing methods used with QuickBASIC when programming Visual Basic results in VB operation that meets my simple, straightforward and easy to understand criteria. Plus the results work perfectly each time through the real-time loop!

The VB version of the Serial Protocol Subroutines provided with this manual is set up to follow exactly these guidelines with a maximum level of commonality to the procedures used with QuickBASIC.

Remember, MSComm is set up to make an asynchronous system appear synchronous. In a like manner, the VB Serial Protocol Subroutines are set up to make a VB C/MRI application program run as smoothly as a QuickBASIC C/MRI application program. That's a worthy and achievable goal following the VB application examples provided in this manual, operating with the VB Serial Protocol Subroutines presented in this appendix.

SERIAL PROTOCOL SUBROUTINES – VISUAL BASIC VERSION

Just like with QuickBASIC, the standard VB Serial Protocol Subroutine package consists of 5 separate subroutines with the basic functions defined in Table C-1.

Table C-1. Standard Visual Basic Serial Protocol Subroutines (SPSVBM)

Subroutine	Function
INIT	Invoked by application program to initialize node
INPUTS	Invoked by application program to read input bytes IB(1) up through IB(NI) from the interface hardware, where NI = number of input ports contained within node
OUTPUTS	Invoked by application program to write output bytes OB(1) up through OB(NO) to the interface hardware, where NO = number of output ports contained within node
RXBYTE	Used by INPUTS to read a single input byte (INBYTE) from the PC receive buffer as received from the interface hardware
TXPACK	Used by INIT, INPUTS and OUTPUTS to formulate and transmit a complete data packet, a message, from the PC to the interface hardware

The three subroutines INIT, INPUTS and OUTPUTS are invoked by the user while RXBYTE and TXPACK, frequently referred to as utility subroutines, are invoked by INIT, INPUTS and OUTPUTS and thereby typically invisible to the user.

The VB Serial Protocol Subroutines are "ready-to-use" and essentially application-independent and I'll cover their content to give you a good understanding of how they work. These subroutines assume using a version of Microsoft Visual Basic V6.0 that includes Microsoft's MSComm. Users of Visual Basic V5.0 will need to download Service Pack No. 3 (SP3) corrections to MSComm or install a third-party package called XMComm. Likewise, VB5.0 and VB6.0 users with the Educational, or Learning Edition, will need to locate a copy of MSComm or alternatively install XMComm. See Chapter 15 for details regarding both SP3 and XMComm.

As we work our way through the VB Version of the Serial Protocol Subroutine listings, you can refer to Table C-2 as a handy reminder of the programming acronyms, variables, being used.

Table C-2 is identical to Table B-7 for QuickBASIC except that the PA variable is dropped and the MAXBUF variable is added with VB.

The remainder of this appendix is devoted to working our way through all five VB versions of the Serial Protocol Subroutines. As stated previously, I'll focus on the differences with VB so for a more complete discussion of the code that doesn't change, please refer back to Appendix B.

Table C-2. Programming acronyms used within the Serial Protocol Subroutines – Visual Basic version

Symbol	Variable Description
UA	USIC node address (range 0 to 127)
COMPORT	Computer's COM port being used for your C/MRI (set for your computer's configuration = 1, 2, 3 or 4)
BAUD100	Baud rate divided by 100 that must correspond to the setting on the baud rate dip switch on USIC, SUSIC or SMINI
NDP$	Node definition parameter set to: "N" when using 24-bit I/O cards with USIC or SUSIC "X" when using 32-bit I/O cards with SUSIC "M" when using SMINI
NDPD	Node definition parameter decimal equivalent of NDP$
DL	USIC transmission delay between bytes being sent back to the PC from the USIC, SUSIC and SMINI. (Adjust to different values, but approaching 0, as required for your computer; see applications instructions)
NS	**For USIC and SUSIC:** Set to the number of card sets of 4 that are plugged into the node. To calculate NS, divide the total number of I/O cards plugged into the node by 4 and round up to the nearest whole number. For example with 13 I/O cards, NS would be 4 (calculated as 13 ÷ 4 or 3.25, rounded up to 4). **For SMINI:** Set to the number of 2-lead LED searchlight signals that need yellow aspect oscillation. Equals 0 if no such signals connected.
NI	Number of input ports (set to 3 times number of 24-bit input cards or 4 times number of 32-bit input cards). NI = 3 for SMINI.
NO	Number of output ports (set to 3 times number of 24-bit output cards or 4 times number of 32-bit output cards). NO = 6 for SMINI
CT( )	**For USIC and SUSIC:** Card type array used to define which card slots (card addresses) are occupied by input cards and which are occupied by output cards. The number of defined CT( ) array elements must equal the value of NS. **For SMINI:** Card type array used to define bit positions within each card position where 2-lead LED searchlight signals are connected. Not used for SMINI applications w/o 2-lead LED searchlight signals.
IB( )	Array used to store the input data bytes read in from the USIC, SUSIC and SMINI node's input ports. Must be dimensioned to NI or greater.
OB( )	Array used to store the output data bytes to be sent to the USIC, SUSIC and SMINI node's output ports. Must be dimensioned to NO or greater.
TB( )	Transmit byte buffer used internally by the Serial Protocol Subroutines to store the complete message packet of bytes to be sent to the USIC, SUSIC or SMINI. Must be dimensioned to (NO + 6) or greater.
MAXTRIES	Maximum number of times the Serial Protocol Subroutine loops waiting for an input byte before it aborts reading inputs.
MAXBUF	Maximum number of input bytes stored in MSComm's input data buffer before RXBYTE subroutine declares PC overrun error
INBYTE	Input byte read in by RXBYTE subroutine
ABORTIN	Flag set to 1 when RXBYTE aborts reading inputs, otherwise = 0
INTRIES	Number of input tries counter
INITERR	Initialization error flag set to 1 when INIT detects an error in initialization data, otherwise = 0
MT	Message type set by Serial Protocol Subroutines (see Table B-1)
LM	Length of message used by INIT, OUTPUTS and TXPACK subroutines
TP	A software pointer, used as a subscript within the transmit buffer to point to the specific byte number to be transmitted.

A listing of the VB version of the Serial Protocol Subroutines is provided at the end of this appendix. Also, it's included on the disk supplied with this manual as file SPSVBM.BAS. In the listing, I elected to group all 5 subroutine modules (INIT, OUTPUTS, INPUTS, RXBYTE and TXPACK). This is the identical format obtained when invoking the "Print" command from the VB Menu when looking at the SPSVBM.BAS module.

In the listing, the only "separation" between the different CALLed subroutines is the END SUB statement of one subroutine and the beginning SUB *name* ( ) statement of the following subroutine. The resulting advantage of this combination is that you can look over the entire set of 5 subroutines in one straight listing. In many cases this can be an advantage, especially when examining the interactions between different subroutines. Let's now examine some of the key programming statements within each of the 5 CALLed subroutines.

Initialization Subroutine – INIT

The initialization subroutine is invoked each time a user needs to initialize a C/MRI node. For example if you have a distributed serial system with 7 nodes then the INIT subroutine is called 7 times – 1 time for each node. Except for the actual statements used to initialize the PC's serial communication port, via MSComm, the INIT subroutine for VB is close to identical to the INIT subroutine used with QuickBASIC.

One obvious initial difference is the inclusion of the list of global variables required to use the VB version of Serial Protocol Subroutines. We defined this list back in Fig. 15-1. By using the Public statement and placing this list at the Module level as part of SPSVBM.BAS, the variables are sharable between all Forms and all Modules for any VB project that includes the SPSVBM.BAS module.

Besides these added declaration statements and some title changes to reflect the VB version, all the other statements down to the point noted by the REMark "CHECK FOR VALID NUMBER FOR PC COM PORT...." are identical. At this point, comparing the SPSVBM.BAS program listing to Fig. B-5, a difference with the VB version is not setting the Port Address (PA) values corresponding to each of the COMPORT numbers. This omission is because the VB version, using MSComm, doesn't require inputting the PA values. The checks in the VB version simply validate that the user provided valid numbers for COMPORT, namely 1, 2, 3 or 4.

If invalid, the program falls through all 4 IF-THEN statements to print an error message. Because all printouts to the monitor need to be accomplished via Forms, all Print statements within a Module must be referenced back to the Form within which they are to be printed. Thus, throughout the SPSVBM Module, we'll write each Print statement as Form1.Print.

The block of code beginning with the REMark "CHECK FOR VALID BAUD RATE," has a similar simplification when compared to Fig. B-5. With VB, baud rate values are inserted directly as a property of MSComm so there is no need to define the corresponding values for the baud low-order latch (BAUDLS) as we did in QuickBASIC. The VB version simply checks that the user provides valid numbers for BAUD100, the baud rate divided by 100. If not, the program falls through all 5 IF-THEN statements to print an error message.

Following these two simplifications, the code checks to see if the user defined MAXBUF within its validity range 1 through 262. If defined outside of this range, an error message is printed.

The program then stays identical all the way down to just below the label INCHKCMP where both the QuickBASIC and VB versions set up the PC's communication port. As would be expected at this point, things are considerably different between the two approaches. Basically with QuickBASIC we can initialize the UART by directly accessing its registers using OUT statements. By contrast, with VB, we need to initialize MSComm, which in turn initializes the UART.

For convenience in our discussion, I've copied the statements used to initialize MSComm within the SPSVBM.BAS program listing and pasted them into Fig. C-1. It's important to point out that in the SPSVBM listing, and on disk, each use of MSComm is prefixed by a "Form1.", just like we did with each Print statement. This is required when MSComm is referenced outside of Form1 which contains the linkages for MSComm. Note: if the statement involving MSComm were being used within Form1, then the prefix would not be required.

```
REM**CLOSE PC COMMUNICATIONS PORT IF ALREADY OPENED SO CAN INITIALIZE PARAMETERS
   IF Form1.MSComm1.PortOpen = True Then Form1.MSComm1.PortOpen = False

REM**SET MSComm1 TO SELECTED PORT
   Form1.MSComm1.CommPort = COMPORT  'Set serial communications port to user...
                                     '...defined value

REM**SET BAUD RATE, NO PARITY CHECKING, 8 DATA BITS AND 2 STOP BITS
   IF BAUD100 = 96 THEN Form1.MSComm1.Settings = "9600,n,8,2"
   IF BAUD100 = 192 THEN Form1.MSComm1.Settings = "19200,n,8,2"
   IF BAUD100 = 288 THEN Form1.MSComm1.Settings = "28800,n,8,2"
   IF BAUD100 = 576 THEN Form1.MSComm1.Settings = "576000,n,8,2"
   IF BAUD100 = 1152 THEN Form1.MSComm1.Settings = "115200,n,8,2"

REM**INITIALIZE REMAINDER OF MSComm1 PROPERTIES
   Form1.MSComm1.InputLen = 1        'Set up to read in 1 byte at a time
   Form1.MSComm1.InputMode = 1       'Specify input bytes read as binary (0 = text)
   Form1.MSComm1.InBufferSize = 262  'Specify input buffer size at 262 bytes
   Form1.MSComm1.OutBufferSize = 262 'Specify output buffer size at 262 bytes
   Form1.MSComm1.PortOpen = True     'Open serial port
   Form1.MSComm1.InBufferCount = 0   'Clear input buffer count and content
   Form1.MSComm1.OutBufferCount = 0  'Clear output buffer count and content
```

Fig. C-1. Statements used to initialize MSComm within the SPSVBM.BAS program

The first statement checks to see if MSComm1 has been previously opened and, if it has, it is closed to enable us to initialize its parameters. The next statement defines to MSComm that the serial port being used equals the numeric value supplied by the user and stored in the variable COMPORT, which we've already validated as being a 1, 2, 3 or 4. For example, if your VB application program initialized COMPORT = 2, then the VB statement **`Form1.MSComm1.CommPort = COMPORT`** simply defines that you are setting up the MSComm1 control to use the PC's COM port number 2.

I should point out that the numeral 1 is appended to the MSComm as a default option because VB programs can make use of multiple PC serial ports. For example, a frequent application using two different PC serial communication ports, within a single application program, is to tie a C/MRI system in with a DCC command control system to support automatic control of DCC equipped locomotives following C/MRI controlled signals. To support this situation, for example, the PC's COM 2 port could be used for connecting to the C/MRI while the COM 3 port could be connected to the DCC Command Control system.

It may seem a bit confusing but MSComm1 doesn't denote using COM 1 and MSComm2 doesn't denote using COM 2, etc. Each MSComm control (MSComm1, MSComm2 and so forth) takes on the COM port that is specified by the **`Form1.MSComm`*`N`*`.CommPort = COMPORT`** statement. That is, the numeral used in place of the ***N*** is not correlated to the numerical value stored in the variable COMPORT.

The **`Form1.MSComm1.Settings`** statement is used to define the values to be used for baud rate (placed in the first argument position after the first quotation mark. Then the next three arguments define whether or not parity checking is to be implemented (in our case an "n" is inserted declaring no parity checking), the number of data bits being transmitted (in our case 8) and the number of stop bits being used (in our case 2). Five separate IF-THEN statements are used to pick up the different baud rates corresponding to each inputted value of BAUD100, which were previously checked for being valid entries. For example if BAUD100 in the application program is initialized by the user to 192, then the **`IF BAUD100 = 192 THEN Form1.MSComm1.Settings = "19200, n, 8, 2"`** statement defines the PC's COM port settings at 19200 bps, no parity, 8 data bits and 2 stop bits.

The next block of 7 statements initializes additional MSComm properties. For example, the first statement **`Form1.MSComm1.InputLen = 1`** tells the MSComm control to read inputs as single data bytes. The next statement, **`Form1.MSComm1.InputMode = 1,`** defines that the data bytes are in binary format. This is required because the MSComm I/O buffers can be set up to store string data (multiple ASCII characters like A, B, C, etc.) or direct binary data (numeric data bytes like 1, 2, 3, etc.). Changing the setting to a 0 would signify using ASCII string characters rather than numeric data. The C/MRI protocol, as detailed in Appendix B, is based upon transmitting and receiving straight binary numeric data.

The next 2 statements **Form1.MSComm1.InBufferSize = 262** and **Form1.MSComm1.OutBufferSize = 262** set the MSComm's input and output buffer size to 262 bytes. These settings provide sufficient space to store the largest possible node using 64 of the 32-bit I/O cards. The 262 number is derived from 64 cards x (4 ports per card) plus 6 bytes of overhead for the 255, 255, STX, UA +65, MT and the closing ETX.

Once the buffer sizes are established, we're ready to turn on, or open, the serial port, which is accomplished by the **`Form1.MSComm1.PortOpen = TRUE`** statement. The corresponding statement to close a port is **`From1.MSComm1.PortOpen = FALSE`**.

The 2 statements, **`Form1.MSComm1.InBufferCount = 0`** and **`Form1.MSComm1.OutBufferCount = 0`** initialize MSComm's input and output data buffer counters to zero as well as clear the buffer contents. This completes initializing the PC's communication port so that it is ready to support real-time loop operations.

With the setup of the PC's COM port complete, the remaining statements within the INIT subroutine, such as setting up the C/MRI hardware node, are identical to those used with QuickBASIC and detailed in Appendix B.

Inputs Subroutine – INPUTS

The INPUTS subroutine is invoked by the user each time it's desired to read all the C/MRI input ports from a given node. To accomplish this, INPUTS first sets up and transmits via TXPACK a poll-request, or "P", message to the C/MRI. This tells the addressed node to gather up all the data from all its input ports and transmit it back to be received by the PC. Thus, after transmitting the "P" message, INPUTS goes into a continuous loop to read in the corresponding receive, or "R", message containing the input port data from the addressed node.

Comparing the SPSVBM.BAS listing to Fig. B-8 it can be seen that the only difference in the INPUTS subroutine is an added statement to clear MSComm's receive data buffer just prior to executing CALL TXPACK to send out the poll request message. This statement, **`Form1.MSComm1.InBufferCount = 0`**, insures that MSComm's input buffer count and buffer content are cleared of any possible leftover or

extraneous data and ready to receive updated data coming back from the C/MRI in response to the poll request message**.**

Outputs Subroutine – OUTPUTS

The OUTPUTS subroutine is invoked by the user each time it's desired to send outputs to all the C/MRI output ports within a given node. It is the simplest of all the Serial Protocol Subroutines. Comparing the SPSVBM.BAS listing and Fig. B-9 shows the OUTPUTS subroutine modules to be identical between VB and QuickBASIC except for the one added VB statement, **MSComm1.OutBufferCount = 0.** Including this statement insures that MSComm's output buffer count and buffer content are cleared of any possible leftover or extraneous data and ready to receive updated data being loaded from the VB application program for transmission out to the C/MRI interface hardware.

Receive Byte Subroutine – RXBYTE

RXBYTE is a utility subroutine that receives a single Input Byte, INBYTE, as being transmitted back from the addressed C/MRI node to the PC's serial port. It's a separate subroutine because it's used repetitively by INPUTS. Because RXBYTE actually receives the input byte and because the VB version makes use of MSComm, RXBYTE's coding is significantly different from the QuickBASIC version. These differences are identifiable by comparing the SPSVBM.BAS listing and Fig. B-10.

First off, MSComm requires that the received bytes be placed into an array variable – and preferably a string array variable to keep things simple and easy to understand. The array is required even if MSComm is set up, as in our case, to receive only a single byte. This requires setting up a DIMensioned array variable, I've called it ARYBYTE(1). Because ARYBYTE(1) is used only within the RXBYTE subroutine, it can be "private" with its corresponding DIM statement placed at the beginning of the RXBYTE module. Because with our use of the DefInt A-Z statement we have implicitly defined every variable to be Integer type, we need to use the following statement to adequately define ARYBYTE:

```
Dim ARYBYTE(1) As String
```

The InputMode property, defined in the INIT subroutine, determines the type of data that is retrieved with the Input property within RXBYTE. If InputMode is set to 0, then the Input property returns text data. If InputMode is set to 1, as we've done, the Input property returns binary data in an array of bytes (in our case an array of one element).

The data being received via MSComm from the C/MRI hardware is coming in as data bytes – meaning that each element received takes up only 8-bits of memory for an effective range 0 to 255. By contrast, normal integer type variables take up 2 bytes for an effective range of -32,768 to 32,767.

The first executable statement within RXBYTE initializes a variable called INTRIES to count the number times RXBYTE loops to read inputs, before deciding to abort input tries. The second executable statement initializes the aborting reading input flag (ABORTIN) to 0. Following these initializations, the RXBYTE subroutine goes into a loop, starting at the label INLOOP, to receive an input byte.

The first statement inside the loop, a DoEvents, is an obvious addition to the VB application. As discussed previously, Windows needs to take slices of time away from the VB program execution to do the things that Windows needs to do. Including DoEvents statements within a VB program provides an effective way for the VB program to inform Windows that this is a convenient time for activating Windows activity. You might ask, "Why locate a DoEvents statement at this particular location?" At this

location the VB program is waiting in a tight loop for an input byte to arrive. As long as we are operating in a wait loop it's a better time than most to have Windows do its thing.

Operationally, execution of the DoEvents statement returns control over to Windows. If Windows has no other task waiting in line, control is returned directly back to the VB program. If other tasks are waiting in line, Windows dedicates some time slices to these tasks and then returns control to the VB program. Basically, by including DoEvents at locations where interrupts will cause the least interruption, the VB program effectively minimizes Windows interruptions at points where they are more disturbing. To speed up your application, you might want to try commenting out the DoEvents statement.

The next statement inside the loop following the DoEvents provides a check to see if the computer's serial I/O port connected to the C/MRI is experiencing overrun errors. Such errors would occur, for example, if the C/MRI were repeatedly sending data back to the PC faster than it can be processed.

Because MSComm provides input byte buffering, detecting an overrun error with VB is different than with QuickBASIC. During normal operation, MSComm is collecting and storing input bytes in its input buffer and the VB application programming is removing these bytes – all in an asynchronous manner. Proper operation requires that the VB application program, on the average, be capable of removing bytes faster than they are arriving. If the VB application can't retrieve the input bytes from MSComm fast enough, then the input buffer keeps filling until it overflows.

The default buffer size is 1024 bytes, but during INIT we've limited the buffer size to a maximum of 262 bytes. However, practice shows that you typically don't have to let the buffer contents grow to this amount before you know there's a problem. How high you want the buffer count to grow before declaring a PC overrun is established by user-defined variable MAXBUF. A typically defined user value is 50.

If the buffer count ever grows above MAXBUF, a PC overrun message is printed. Under normal circumstances, the variable InBufferCount is less than MAXBUF, the error message printout is avoided and the program proceeds directly to increment the number of input tries loop counter, INTRIES.

Directly after incrementing INTRIES a check is made to determine if INTRIES has reached the maximum allowable number of tries before aborting trying to read inputs. This test is important to avoid possible system lockups where the PC is looping waiting for inputs and the C/MRI node isn't sending back inputs to the PC because it's also sitting there waiting for inputs from the PC. This undesired looping-state is broken when the number of input tries reaches the value MAXTRIES. If and when this ever happens, the aborting inputs flag is set to 1, the INTRIES counter is reinitialized to 0, a corresponding error message is printed, input byte is defaulted to a value of 0, the input buffer count and the buffer itself are cleared and the program branches to label RXRET to exit the subroutine via the END SUB statement.

Assuming the general case where the INTRIES counter has not reached MAXTRIES, the IF-THEN-END-IF statement block is skipped over and the program checks MSComm to see if the input buffer count is 0, meaning that there is no data currently in the input buffer. For the case when the count is zero, input data isn't available, so the subroutine loops back to INLOOP. With correct operation, this loop keeps functioning until the buffer count becomes greater than 0, showing that data is available; the subroutine then falls through the IF-THEN statement to input the data via MSComm and place it into the variable location INBYTE as accomplished by the following statements:

```
ARYBYTE(1) = Form1.MSComm1.Input  'Transfer received input byte available from...
                                  '...MSComm to required input byte array of 1 element
                                  '...if using XMComm, change .Input to .InputData
INBYTE = AscB(ARYBYTE(1))         'Move byte from required array to desired INBYTE
                                  'Note: AscB function returns the first byte within...
                                  '...string, which is the inputted byte from the C/MRI
```

The first statement moves the received byte (with range 0-255), contained in the PC's UART (see Appendix B) and places it in the memory location denoted by ARYBYTE(1), which is defined as a string. The second statement uses the AscB function to return the first byte within the string – which is the inputted byte – and places it in the desired INBYTE location as a 2-byte integer.

The RXBYTE subroutine completes its job once the inputted data byte is placed into INBYTE. However, to complete the subroutine processing, as with QuickBASIC, I've included the same optional printout for debugging purposes, just prior to the effective return to the calling program via the END SUB statement.

Transmit Packet Subroutine - TXPACK

TXPACK is a utility subroutine that assembles the entire string of bytes that need to be transmitted and then transmits them out to the C/MRI via MSComm. TXPACK is a separate subroutine because it is used by the INIT, INPUTS and OUTPUTS subroutines. The string of bytes to be transmitted, in sequence order, is set up in an array within the Serial Protocol Subroutines referred to as the Transmit Buffer or TB( ) array.

Formulating the TB( ) array in Visual Basic is identical to QuickBASIC. However, how the array is transmitted to the C/MRI hardware is considerably different. The first thing with VB is the need to move the TB( ) array content to MSComm's output buffer. This is accomplished by setting up a FOR-NEXT loop from 1 to TP with the `Form1.MSComm1.Output = CHR$(TB(I))` statement placed within the loop. The CHR$ function is required to change the TB( ) array element from straight binary to ASCII as required by MSComm.

The next requirement with VB is to wait until MSComm has transmitted all the output bytes. This is accomplished by simply waiting in a loop until MSComm's output buffer count reaches 0. For the case when the count is greater than zero, i.e. data to be transmitted by MSComm is still in its output buffer, the subroutine simply loops back to the label EMTY to check again. A DoEvents statement is included within this loop to inform Windows that within this wait loop is a good time to take those slices of time so that Windows can do what Windows needs to do. To speed up your application, you might want to try commenting out the DoEvents statement.

Once OutBufferCount reaches 0, MSComm has transmitted all output data to the C/MRI hardware, so the TXBYTE wait is over and control effectively returns to the calling program via the END SUB statement.

That's the end of our discussion covering the Visual Basic version of the Serial Protocol Subroutines. What's really nice is that the Serial Protocol Subroutines are provided in a ready-to-use format that is easy to apply and essentially application independent. Therefore to apply the subroutines you really don't need to understand every detail concerning how they function internally.

However, if you would like to pursue additional information about Microsoft's MSComm control, I recommend checking out the web site:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/comm98/html/vbobjComm.asp

Under a link called properties, the site provides a pretty good explanation covering all the different options that can be set up.

PROGRAM SOURCE LISTING (SPSVBM.BAS)

The remainder of this appendix contains the source listing of the SPSVBM.BAS program set up for use as a separate module under the file name SPSVBM.BAS.

```
      DefInt A-Z       'Make variables Integer by default
   REM**GLOBALIZED ARRAY VARIABLES
   REM**PUBLIC is used in place of DIM so that the variables are globally
        '...accessible throughout the entire Visual Basic Project
      Public OB(60)           'Output byte array
      Public IB(60)           'Input byte array
      Public CT(15)           'Card type array
      Public TB(80)           'Transmit buffer array

   REM**USER DEFINED GLOBALIZED NON-ARRAY VARIABLES
      Public UA               'USIC address (range 0 to 127)
      Public COMPORT          'PC Communications port = 1, 2, 3 or 4
      Public BAUD100          'PC baud rate divided by 100
      Public DL               'USIC transmission delay
      Public NDP As String    'Mode definition parameter = "M", "N" or "X"
                              '... We want this to be a string, not an
                              '... integer so we add 'As String'
      Public NS               'Number of card sets of 4 for SUSIC and USIC...
                              '...applications and number of 2-lead...
                              '...searchlight signals for SMINI applications
      Public NI               'Number on input ports
      Public NO               'Number of output ports
      Public MAXTRIES         'Maximum number of PC tries to read input...
                              '...bytes prior to PC aborting inputs
      Public MAXBUF           'Maximum number of input bytes allowed in MSComm's...
                              '...input data buffer before declaring PC overrun error

   REM**SUBPROGRAM DEFINED GLOBALIZED NON-ARRAY VARIABLES
      Public INBYTE              'Input byte read in by RXBYTE subprogram
      Public ABORTIN             'Flag set to 1 when by RXBYTE subprogram...
                                 '...aborts reading inputs, otherwise = 0
      Public INTRIES             'Number of input tries counter
      Public INITERR             'Flag set by INIT subprogram when detects...
                                 '...and error in initialization input data
      Public LM                  'Length of message used by INIT, OUTPUTS...
                                 '...and TXPACK subprograms
      Public MT                  'Message type set Serial Protocol Subroutines

Sub INIT()
    Rem************************************************
    Rem**********SERIAL PROTOCOL SUBROUTINES*************
    Rem********VISUAL BASIC VERSION USING MSComm**********
    Rem     SPSVBM.BAS    DATED: August 30, 2003        **
    Rem    FOR USIC, SUSIC AND SMINI Applications       **
    Rem************************************************

    Rem**SPSVBM PACKAGE CONTAINS 5 SEPARATE SUBROUTINES DEFINED AS:
        '1)  INIT    -- Invoked by application program to initialize node
        '2)  INPUTS  -- Invoked by application program to read input bytes
                         'IB(1) up through IB(NI) from the interface hardware
                         'where NI = number of input ports
        '3)  OUTPUTS -- Invoked by application program to write output bytes
                         'OB(1) up through OB(NO) to the interface hardware
                         'where NO = number of output ports
        '4)  RXBYTE  -- Used by INPUTS to read an input byte (INBYTE) into
                         'the PC receive buffer from the interface hardware
        '5)  TXPACK  -- Used by INIT, INPUTS and OUTPUTS to formulate and
```

```
                        'transmit data packet from PC to interface hardware

    Rem*********************************************
    Rem**               ***INIT***                 **
    Rem*******VISUAL BASIC VERSION USING MSComm******
    Rem**     NODE INITIALIZATION SUBROUTINE       **
    Rem** for use with USIC, SUSIC and SMINI nodes **
    Rem*********************************************

        ' **NOTE: This subroutine must be executed correctly prior to...
        '         ...invoking INPUTS and OUTPUTS subroutines
        '  Following parameters must be defined in the application...
        '    ...program prior to invoking INIT:
        '       UA       = USIC address (range 0 to 127) unless using...
        '                  ...Classic USIC with 68701 then range is 0 - 15
        '       COMPORT  = PC COM port = 1, 2, 3 or 4
        '       BAUD100  = PC baud rate divided by 100
        '       DL       = USIC transmission delay
        '       NDP$     = Node definition parameter = "M", "N" or "X"
        '       MAXTRIES = Maximum number of PC tries to read input...
        '                  ... bytes prior to PC aborting inputs
        '       MAXBUF   = Maximum number of input bytes allowed in MSComm's...
        '                  ...input data buffer before declare PC overrun error

        'For SMINI applications:
        '       NS       = Number of 2-lead searchlight signals...
        '                      ...requiring yellow oscillation feature
        'Only for SMINI case when NS > 0 i.e. signals are present
        '       CT(1) through CT(6) = Card type definition elements...
        '       ...defining signal bit locations within each...
        '       ...of the SMINI's 6 output ports
        'For SUSIC and USIC applications:
        '       NS       = Number of I/O card sets of 4
        '       CT(1) through CT(NS) = Card type definition elements...
        '       ...defining the card type arrangement within...
        '       ...each set of 4 cards


    Rem**BEGIN INITIALIZATION OF USIC, SUSIC OR SMINI
    Rem**Initialize intries counter and initialization error flag
       INTRIES = 0     'Initialize INTRIES counter to zero
       INITERR = 0     'Initialize error flag to zero

    Rem**CHECK FOR VALID RANGE OF USIC ADDRESS
         'NOTE: range 0 - 15 is not checked for Classic USIC with 68701
       If UA > 127 Then
          Form1.Print "***ERROR*** UA = "; UA; " Out of range 0 to 127"
          INITERR = 1
       End If

    Rem**CHECK FOR VALID NUMBER FOR PC COM PORT (COM1 THRU COM4)
       If COMPORT = 1 Then GoTo COMOK
       If COMPORT = 2 Then GoTo COMOK
       If COMPORT = 3 Then GoTo COMOK
       If COMPORT = 4 Then GoTo COMOK
       Form1.Print "***ERROR*** COMPORT = "; COMPORT; " MUST = 1, 2, 3 OR 4"
       INITERR = 1
COMOK:

    Rem**CHECK FOR VALID BAUD RATE
       If BAUD100 = 96 Then GoTo BAUDOK
       If BAUD100 = 192 Then GoTo BAUDOK
       If BAUD100 = 288 Then GoTo BAUDOK
       If BAUD100 = 576 Then GoTo BAUDOK
```

```
      If BAUD100 = 1152 Then GoTo BAUDOK
      Form1.Print "***ERROR*** BAUD100 = "; BAUD100
      Form1.Print "Valid BAUD100 values are 96, 192, 228, 576 and 1152"
      INITERR = 1
BAUDOK:

   Rem**CHECK FOR VALID RANGE OF MAXBUF
      If MAXBUF < 1 Or MAXBUF > 262 Then
         Form1.Print "***ERROR*** MAXBUF = "; MAXBUF
         Form1.Print "Valid MAXBUF range is 1 through 262"
      End If

   Rem**CHECK FOR VALID NODE DEFINITION PARAMETER AND BRANCH ACCORDINGLY
      If NDP$ = "M" Then GoTo CHKMINI
      If NDP$ = "N" Then GoTo CHKMAXI
      If NDP$ = "X" Then GoTo CHKMAXI
      Form1.Print "***ERROR*** NDP$ = "; NDP$; " must be set to M, N, OR X "
      Form1.Print "****INITIALIZATION TERMINATED DUE TO INVALID NDP$ ****"
      INITERR = 1
      GoTo INITRET

CHKMINI:
   Rem*************BEGIN SMINI SPECIFIC PARAMETER CHECKING************
   Rem**CHECK FOR VALID NI, NO AND NS USING SMINI
      If NI <> 3 Then
        Form1.Print "INVALID NI = "; NI; "MUST BE NI = 3 FOR SMINI"
        INITERR = 1
      End If

      If NO <> 6 Then
        Form1.Print "INVALID NO = "; NO; "MUST BE NO = 6 FOR SMINI"
        INITERR = 1
      End If

      If NS = 0 Then GoTo INCHKCMP   'No signals so branch to initialization...
                                     '...checking complete

   Rem**SMINI HAS 2-LEAD OSCILLATING SIGNAL SO CHECK IF NS IN RANGE
      If NS > 24 Then
        Form1.Print "**ERROR** NS = "; NS; " OUT OF RANGE 0 TO 24 FOR SMINI"
        INITERR = 1
      End If

   Rem**CHECK FOR VALID CT ARRAY ELEMENTS WHILE COUNTING SIGNALS TO EQUAL NS
      NSCNT = 0        'Initialize signal count to zero
      For I = 1 To 6   'Loop through 6 SMINI CT elements
         If CT(I) = 0 Then GoTo NEXTCT
         If CT(I) = 3 Then NSCNT = NSCNT + 1: GoTo NEXTCT
         If CT(I) = 6 Then NSCNT = NSCNT + 1: GoTo NEXTCT
         If CT(I) = 12 Then NSCNT = NSCNT + 1: GoTo NEXTCT
         If CT(I) = 24 Then NSCNT = NSCNT + 1: GoTo NEXTCT
         If CT(I) = 48 Then NSCNT = NSCNT + 1: GoTo NEXTCT
         If CT(I) = 96 Then NSCNT = NSCNT + 1: GoTo NEXTCT
         If CT(I) = 192 Then NSCNT = NSCNT + 1: GoTo NEXTCT
         If CT(I) = 15 Then NSCNT = NSCNT + 2: GoTo NEXTCT
         If CT(I) = 27 Then NSCNT = NSCNT + 2: GoTo NEXTCT
         If CT(I) = 51 Then NSCNT = NSCNT + 2: GoTo NEXTCT
         If CT(I) = 99 Then NSCNT = NSCNT + 2: GoTo NEXTCT
         If CT(I) = 195 Then NSCNT = NSCNT + 2: GoTo NEXTCT
         If CT(I) = 30 Then NSCNT = NSCNT + 2: GoTo NEXTCT
         If CT(I) = 54 Then NSCNT = NSCNT + 2: GoTo NEXTCT
         If CT(I) = 102 Then NSCNT = NSCNT + 2: GoTo NEXTCT
         If CT(I) = 198 Then NSCNT = NSCNT + 2: GoTo NEXTCT
         If CT(I) = 60 Then NSCNT = NSCNT + 2: GoTo NEXTCT
```

```
         If CT(I) = 108 Then NSCNT = NSCNT + 2: GoTo NEXTCT
         If CT(I) = 204 Then NSCNT = NSCNT + 2: GoTo NEXTCT
         If CT(I) = 120 Then NSCNT = NSCNT + 2: GoTo NEXTCT
         If CT(I) = 216 Then NSCNT = NSCNT + 2: GoTo NEXTCT
         If CT(I) = 240 Then NSCNT = NSCNT + 2: GoTo NEXTCT
         If CT(I) = 63 Then NSCNT = NSCNT + 3: GoTo NEXTCT
         If CT(I) = 123 Then NSCNT = NSCNT + 3: GoTo NEXTCT
         If CT(I) = 243 Then NSCNT = NSCNT + 3: GoTo NEXTCT
         If CT(I) = 111 Then NSCNT = NSCNT + 3: GoTo NEXTCT
         If CT(I) = 207 Then NSCNT = NSCNT + 3: GoTo NEXTCT
         If CT(I) = 219 Then NSCNT = NSCNT + 3: GoTo NEXTCT
         If CT(I) = 126 Then NSCNT = NSCNT + 3: GoTo NEXTCT
         If CT(I) = 222 Then NSCNT = NSCNT + 3: GoTo NEXTCT
         If CT(I) = 246 Then NSCNT = NSCNT + 3: GoTo NEXTCT
         If CT(I) = 252 Then NSCNT = NSCNT + 3: GoTo NEXTCT
         If CT(I) = 255 Then NSCNT = NSCNT + 4: GoTo NEXTCT
         Form1.Print "**ERROR** INVALID CT("; I; ") = "; CT(I); " FOR SMINI"
         INITERR = 1
NEXTCT:
       Next I

       If NSCNT <> NS Then
          Form1.Print "**ERROR** SIGNAL COUNT FROM CTs <> NS FOR SMINI"
          INITERR = 1
       End If
       GoTo INCHKCMP   'Proceed to initialization checking complete

   Rem*************BEGIN SUSIC SPECIFIC PARAMETER CHECKING************
   Rem**CHECK FOR VALID NS AND CTs USING USIC AND SUSIC
CHKMAXI:
      If NS = 0 Or NS > 16 Then 'Maximum value NS = 16 for 64 cards
         Form1.Print "**ERROR** NS ="; NS; " OUT OF RANGE 1 TO 16 FOR SUSIC NODE"
         INITERR = 1
      End If

      NICT = 0: NOCT = 0    'Initialize I/O type card counters to zero

   Rem**GO THROUGH CT ELEMENT TO CHECK IF VALID AND COUNT I/O CARDS
      For I = 1 To NS
         If I = NS Then 'Note: These CT( ) elements valid only for last card set
           If CT(I) = 2 Then NOCT = NOCT + 1: GoTo NEXTCTU
           If CT(I) = 1 Then NICT = NICT + 1: GoTo NEXTCTU
           If CT(I) = 10 Then NOCT = NOCT + 2: GoTo NEXTCTU
           If CT(I) = 6 Then NOCT = NOCT + 1: NICT = NICT + 1: GoTo NEXTCTU
           If CT(I) = 9 Then NOCT = NOCT + 1: NICT = NICT + 1: GoTo NEXTCTU
           If CT(I) = 5 Then NICT = NICT + 2: GoTo NEXTCTU
           If CT(I) = 42 Then NOCT = NOCT + 3: GoTo NEXTCTU
           If CT(I) = 26 Then NOCT = NOCT + 2: NICT = NICT + 1: GoTo NEXTCTU
           If CT(I) = 38 Then NOCT = NOCT + 2: NICT = NICT + 1: GoTo NEXTCTU
           If CT(I) = 22 Then NOCT = NOCT + 1: NICT = NICT + 2: GoTo NEXTCTU
           If CT(I) = 41 Then NOCT = NOCT + 2: NICT = NICT + 1: GoTo NEXTCTU
           If CT(I) = 25 Then NOCT = NOCT + 1: NICT = NICT + 2: GoTo NEXTCTU
           If CT(I) = 37 Then NOCT = NOCT + 1: NICT = NICT + 2: GoTo NEXTCTU
           If CT(I) = 21 Then NICT = NICT + 3: GoTo NEXTCTU
         End If
         If CT(I) = 170 Then NOCT = NOCT + 4: GoTo NEXTCTU
         If CT(I) = 106 Then NOCT = NOCT + 3: NICT = NICT + 1: GoTo NEXTCTU
         If CT(I) = 154 Then NOCT = NOCT + 3: NICT = NICT + 1: GoTo NEXTCTU
         If CT(I) = 90 Then NOCT = NOCT + 2: NICT = NICT + 2: GoTo NEXTCTU
         If CT(I) = 166 Then NOCT = NOCT + 3: NICT = NICT + 1: GoTo NEXTCTU
         If CT(I) = 102 Then NOCT = NOCT + 2: NICT = NICT + 2: GoTo NEXTCTU
         If CT(I) = 150 Then NOCT = NOCT + 2: NICT = NICT + 2: GoTo NEXTCTU
         If CT(I) = 86 Then NOCT = NOCT + 1: NICT = NICT + 3: GoTo NEXTCTU
         If CT(I) = 169 Then NOCT = NOCT + 3: NICT = NICT + 1: GoTo NEXTCTU
```

```
         If CT(I) = 105 Then NOCT = NOCT + 2: NICT = NICT + 2: GoTo NEXTCTU
         If CT(I) = 153 Then NOCT = NOCT + 2: NICT = NICT + 2: GoTo NEXTCTU
         If CT(I) = 89 Then NOCT = NOCT + 1: NICT = NICT + 3: GoTo NEXTCTU
         If CT(I) = 165 Then NOCT = NOCT + 2: NICT = NICT + 2: GoTo NEXTCTU
         If CT(I) = 101 Then NOCT = NOCT + 1: NICT = NICT + 3: GoTo NEXTCTU
         If CT(I) = 149 Then NOCT = NOCT + 1: NICT = NICT + 3: GoTo NEXTCTU
         If CT(I) = 85 Then NICT = NICT + 4: GoTo NEXTCTU
         Form1.Print " **ERROR**INVALID CT("; I; ") = "; CT(I); " OR CT POSITIONING"
NEXTCTU:
      Next I

   Rem**CONVERT I/O CARD COUNTS TO PORT COUNTS
      If NDP$ = "N" Then NOCT = NOCT * 3: NICT = NICT * 3
      If NDP$ = "X" Then NOCT = NOCT * 4: NICT = NICT * 4

   Rem**CHECK IF PORT COUNTS EQUAL NUMBER OF PORTS INPUTTED
      If NOCT <> NO Then
        Form1.Print "**ERROR**NUMBER OF OUTPUT PORTS COUNTED IN CT NOT EQUAL TO NO"
        INTERR = 1
      End If
      If NICT <> NI Then
        Form1.Print "**ERROR**NUMBER OF INPUT PORTS COUNTED IN CT NOT EQUAL TO NI"
        INTERR = 1
      End If

INCHKCMP:
   Rem*********INITIALIZATION PARAMETER CHECKING IS COMPLETE**********
   Rem**SET UP COMMUNICATIONS PORT IN PC USING MSComm
   Rem**Note: User needs to add MSComm to Form1 before program execution

   Rem**SET MScomm1 TO SELECTED PORT
    Form1.MSComm1.CommPort = COMPORT  'Set PC serial communications port to...
                                      '...user defined value
   Rem**Close PC communications port if already opened so can initialize parameters
      If Form1.MSComm1.PortOpen = True Then Form1.MSComm1.PortOpen = False

   Rem**SET BAUD RATE, NO PARITY CHECKING, 8 DATA BITS AND 2 STOP BITS
      If BAUD100 = 96 Then Form1.MSComm1.Settings = "9600,n,8,2"
      If BAUD100 = 192 Then Form1.MSComm1.Settings = "19200,n,8,2"
      If BAUD100 = 288 Then Form1.MSComm1.Settings = "28800,n,8,2"
      If BAUD100 = 576 Then Form1.MSComm1.Settings = "576000,n,8,2"
      If BAUD100 = 1152 Then Form1.MSComm1.Settings = "115200,n,8,2"

   Rem**INITIALIZE REMAINDER OF MSComm1 PROPERTIES
      Form1.MSComm1.InputLen = 1        'Set up to read in 1 byte at a time
      Form1.MSComm1.InputMode = 1       'Specify input bytes read as binary (0 = text)
      Form1.MSComm1.InBufferSize = 262  'Specify input buffer size at 262 bytes
      Form1.MSComm1.OutBufferSize = 262 'Specify output buffer size at 262 bytes
      Form1.MSComm1.PortOpen = True     'Open serial port
      Form1.MSComm1.InBufferCount = 0   'Clear input buffer count and content
      Form1.MSComm1.OutBufferCount = 0  'Clear output buffer count and content

   Rem**DEFINE INITIALIZATION MESSAGE PARAMETERS
      MT = Asc("I")              'Define message type = "I" (decimal 73)
      OB(1) = Asc(NDP$)          'Define node definition parameter
      OB(2) = Int(DL / 256)      'Set USIC delay high-order byte
      OB(3) = DL - (OB(2) * 256) 'Set USIC delay low-order byte
      OB(4) = NS                 'Define number of card sets of 4 for...
                                 'USIC and SUSIC cases and the...
                                 '...number of 2-lead yellow aspect...
                                 '...oscillating signals for the SMINI.
     LM = 4  'Initialize length of message to start of loading CT elements

  Rem**CHECK TYPE OF NODE TO CONTINUE SPECIFIC INITIALIZATION
```

```
     If NDP$ = "M" Then GoTo INITSMINI  'SMINI node so branch accordingly

INITUSIC:
  Rem**SUSIC-NODE (either "N" or "X") SO LOAD CT( ) ARRAY ELEMENTS
     For I = 1 To NS      '...loop through number of card sets...
        LM = LM + 1       '...accumulating message length while...
        OB(LM) = CT(I)    '...loading card type definition CT...
     Next I               '...array elements into output byte array.
     GoTo TXMSG  'CT( )s complete so branch to transmit initialization...
                 '...message to interface

INITSMINI:
  Rem**SMINI-NODE ("M") SO CHECK IF REQUIRES 2-LEAD OSCILLATION...
                   '...SEARCHLIGHT SIGNALS
     If NS = 0 Then GoTo TXMSG 'No signals so hold message length at...
                               '...LM = 4 and branch to transmit packet

  Rem**SMINI CASE WITH SIGNALS (NS > 0) SO LOOP THROUGH TO LOAD...
     For I = 1 To 6       '...signal location CT array elements...
        LM = LM + 1       '...into output byte array while...
        OB(LM) = CT(I)    '...accumulating message length
     Next I

  Rem**FORM INITIALIZATION PACKET AND TRANSMIT TO INTERFACE
TXMSG: Call TXPACK       'Invoke transmit packet subroutine

  Rem**COMPLETED USE OF OUTPUT BYTE ARRAY SO CLEAR IT BEFORE EXIT SUBROUTINE
     For I = 1 To NO: OB(I) = 0: Next I

INITRET:         'Return to application program
End Sub


Sub INPUTS()
  Rem******************************************************
  Rem**                ***INPUTS***                      **
  Rem***********VISUAL BASIC VERSION USING MSComm*********
  Rem** SUBROUTINE TO READ INPUT BYTES IB(1) THRU IB(NI) **
  Rem**   for use with USIC, SUSIC and SMINI nodes       **
  Rem******************************************************
  Rem**TRANSMIT POLL REQUEST TO INITIATE RECEIVING DATA BACK FROM...
                    '...INTERFACE HARDWARE
REPOL:
     MT = Asc("P") 'Define message type as poll request "P" (decimal 80)
     Form1.MSComm1.InBufferCount = 0 'Clear input buffer count and content...
                               '...to insure clean start for receiving input bytes
     Call TXPACK   'Invoke transmit packet subroutine to transmit...
                   '...poll request message to external hardware

  Rem**LOOP ON RECEIVING INPUT BYTES UNTIL RECEIVE A Start-of-Text (STX)
GETSTX:
     Call RXBYTE                       'Receive input byte
     If ABORTIN = 1 Then GoTo INPUTRET
     If INBYTE <> 2 Then GoTo GETSTX    'Input byte not STX so branch...
                                        '...back to read another byte

  Rem**RECEIVED STX SO READ NEXT BYTE AND SEE IF RETURNED DATA IS...
                      '...COMING FROM THE CORRECT USIC ADDRESS
     Call RXBYTE      'Receive input byte which should be USIC address
     If ABORTIN = 1 Then GoTo INPUTRET
     INBYTE = INBYTE - 65   'Subtract offset of 65 from address and...
                            '...check that matches node that was polled
     If INBYTE <> UA Then Form1.Print "ERROR; Received bad UA = "; IBYTE: GoTo REPOL
```

```
  Rem**CORRECT UA IS REPLYING BACK TO PC SO CHECK IF "R" MESSAGE
     Call RXBYTE      'Receive input byte which should be "R"
     If ABORTIN = 1 Then GoTo INPUTRET
  If INBYTE <> 82 Then Form1.Print "Error received not = R for UA = "; UA: GoTo GETSTX

  Rem**MESSAGE IS "R" SO LOOP THROUGH READING INPUTS FROM INPUT PORTS...
                       '...ON INTERFACE HARDWARE
     For I = 1 To NI    'Loop through number of input ports
        Call RXBYTE     'Receive input byte
        If ABORTIN = 1 Then GoTo INPUTRET
  If INBYTE = 2 Then Form1.Print "ERROR: No DLE ahead of 2 for UA = "; UA: GoTo REPOL
  If INBYTE = 3 Then Form1.Print "ERROR: No DLE ahead of 3 for UA = "; UA: GoTo REPOL
        If INBYTE = 16 Then Call RXBYTE    'DLE so next byte read and...
        IB(I) = INBYTE                     '...stored as valid input byte
     Next I

  Rem**RECEIVED ALL NI INPUT BYTES SO CHECK FOR End-of-Text (ETX = 3)
     Call RXBYTE     'Receive input byte which should be ETX
     If ABORTIN = 1 Then GoTo INPUTRET
     If INBYTE <> 3 Then Form1.Print "ERROR: ETX NOT PROPERLY RECEIVED FOR UA = "; UA

INPUTRET:       'Receive message complete so execute RETURN via ENDSUB
                'Note: if aborted inputs then ABORTIN = 1 else = 0
End Sub

Sub OUTPUTS()
  Rem*******************************************************
  Rem**                  ***OUTPUTS***                     **
  Rem************VISUAL BASIC VERSION USING MSComm***********
  Rem** SUBROUTINE TO WRITE OUTPUT BYTES OB(1) THRU OB(NO) **
  Rem**     for use with USIC, SUSIC and SMINI nodes       **
  Rem*******************************************************
  Rem**TRANSMIT DATA FROM PC TO OUTPUT PORTS ON INTERFACE HARDWARE
     MT = Asc("T")  'Define message type = "T" (decimal 84)
     LM = NO        'Define message length as number of output ports
     Form1.MSComm1.OutBufferCount = 0 'Clear output buffer count and output buffer...
                                '...content to insure clean start for...
                                '...transmitting output bytes
     Call TXPACK    'Invoke transmit packet subroutine
                    'Transmit message complete so return to calling program

End Sub

Sub RXBYTE()
  Rem***********************************************
  Rem**                ***RXBYTE***                **
  Rem********VISUAL BASIC VERSION USING MSComm*******
  Rem** SUBROUTINE TO RECEIVE BYTE AT PC FROM NODE **
  Rem** for use with USIC, SUSIC and SMINI nodes   **
  Rem***********************************************
     Dim ARYBYTE(1) As String  'Private array byte required for MSComm reception
  Rem**INITIALIZE INPUT TRIES COUNTER AND ABORTING INPUT FLAG
     INTRIES = 0  'Initialize input tries counter to 0
     ABORTIN = 0  'Initialize aborting input flag to 0

  Rem**SET UP A MAJOR LOOP STARTING AT INLOOP FOR PC TO RECEIVE AN...
              '...INPUT BYTE FROM THE INTERFACE HARDWARE
  Rem**START LOOP BY CHECKING FOR PC OVERRUN ERROR
INLOOP:
     DoEvents    'Comment out to reduce background activities (if...
                 '...required for overall satisfactory VB operation)
     If Form1.MSComm1.InBufferCount > MAXBUF Then 'If more than maximum...
                                                  '...allowable bytes in buffer...
                                                  '...then PC considered being...
```

```
                                                    '...overrun by C/MRI hardware
    Form1.Print "PC overrun at node = "; UA; " with MAXBUF = "; MAXBUF
    End If

  Rem**CHECK IF INPUT TRIES EXCEED MAXIMUM ALLOWED
    INTRIES = INTRIES + 1       'Increment input tries counter
    If INTRIES > MAXTRIES Then    'If counter exceeds maximum tries...
                                  '    ...then abort reading inputs
      ABORTIN = 1: INTRIES = 0
     Form1.Print "INPUT TRIES EXCEEDED "; MAXTRIES; " NODE = "; UA; " ABORTING INPUT"
      INBYTE = 0                  'Aborted input so set input byte value to 0
      Form1.MSComm1.InBufferCount = 0  'Clear input buffer count and content...
      GoTo RXRET                   '...and branch to receive input byte return
    End If

  Rem**READING INPUTS NOT ABORTED SO CHECK TO SEE IF INPUT BUFFER...
                       '...IS LOADED WITH ONE OR MORE INPUT BYTES
    If Form1.MSComm1.InBufferCount = 0 Then GoTo INLOOP  'If input buffer...
                       '...not yet loaded then branch back to beginning...
                       '...of input loop to try read again

  Rem**PC SERIAL INPUT BUFFER IS LOADED SO RECEIVE INPUT BYTE
    ARYBYTE(1) = Form1.MSComm1.Input  'Transfer received input byte available from...
                                  '...MSComm to required input byte array of 1 element
                                  '...if using XMComm, change .Input to .InputData
    INBYTE = AscB(ARYBYTE(1))   'Move byte from required array to desired INBYTE
                                  'Note: AscB function returns the first byte within...
                                  '...string, which is the inputted byte from the C/MRI

    'Form1.Print INBYTE    '!!!!Optional printout of input byte for test and debug
                          'Note: Invoking this print slows PC...
                          '...significantly so will most likely need...
                          '...to significantly increase USIC delay (DL)

RXRET:    'Received byte complete so execute return to calling program
End Sub


Sub TXPACK()
  Rem**************************************************
  Rem**                ***TXPACK***                    **
  Rem*********VISUAL BASIC VERSION USING MSComm*********
  Rem** SUBROUTINE TO TRANSMIT PACKET FROM PC TO NODE **
  Rem**   for use with USIC, SUSIC and SMINI nodes    **
  Rem**************************************************

  Rem**FORM PACKET TO SEND TO USIC, SUSIC OR SMINI
    TB(1) = 255         'Set 1st synchronization byte to all 1's
    TB(2) = 255         'Set 2nd synchronization byte to all 1's
    TB(3) = 2           'Define start-of-text (STX = 2)
    TB(4) = UA + 65     'Add 65 offset to USIC address
    TB(5) = MT          'Define message type
    TP = 6              'Define next position for transmit pointer
    If MT = 80 Then GoTo ENDMSG   'Poll request so branch to end message

  Rem**LOOP TO SET UP OUTPUT DATA IN TRANSMIT BUFFER INCLUDING THE...
        '...ADDING IN OF DLE BYTES WHERE NEEDED
    For I = 1 To LM     'Loop to set up output data...
                        '...including DLE processing
      If OB(I) = 2 Then TB(TP) = 16: TP = TP + 1: GoTo DATABYT

      If OB(I) = 3 Then TB(TP) = 16: TP = TP + 1: GoTo DATABYT
      If OB(I) = 16 Then TB(TP) = 16: TP = TP + 1
DATABYT:
```

```
        TB(TP) = OB(I)    'Move actual data byte to transmit buffer
        TP = TP + 1       'Increment pointer to next byte position
     Next I

  Rem**END MESSAGE FORMATION WITH End-of-text
ENDMSG:
     TB(TP) = 3  'Add end-of-text (ETX = 3)

  Rem**LOOP TO LOAD PACKET TO BE TRANSMITTED INTO MSComm's OUTPUT BUFFER
     For I = 1 To TP    'Loop through transmit buffer to move items to...
                        '...MSComm's output buffer
        Form1.MSComm1.Output = Chr$(TB(I)) 'Load byte to be transmitted into
MSComm's...
                                           '...output buffer as ASCII character
        'Form1.Print TB(I);   '!!!!Optional printout of transmitted byte for...
                             '...test and debug. Note: Invoking this print...
                             '...slows PC significantly so will most likely...
                             '...need to significantly increase USIC delay (DL)
     Next I        'Increment to pick up next output byte

  Rem**WAIT IN LOOP UNTIL MSComm's OUTPUT BUFFER IS EMPTY
EMTY:
     DoEvents          'Return control to Windows for doing Windows activity
     If Form1.MSComm1.OutBufferCount > 0 Then GoTo EMTY 'If transmit buffer...
                       '...is not empty then branch back to wait for buffer to empty
          'Transmission of output bytes is complete so execute RETURN via END SUB
End Sub
```

This completes the listing of the source code for the Visual Basic version of the Serial Protocol Subroutines using MSCOM. A copy is contained on the disk provided with this manual as file SPSVBM.BAS.