Column #147, January 2008 by Jon Williams:

The Power of Networking

For an actor attempting to make his way in Hollywood the word "Networking" takes on a whole host of meanings. It's a crazy business, really, and what most of us find is that those with the same goals, e.g., becoming an established actor, are not abundantly helpful to each other (a few are downright malicious). So, "networking" – actor to actor, that is – is mostly bupkis in my book. Now, I do have a "Hollywood" network, but the only actors in it are very well established, if not particularly well known. Most of my friends in the business do other things: make-up, special FX, etc., and the person that I network with most is a guy named Peter who, like me, is one of those techno-artistic types. Peter directs TV commercials, has worked for a major studio directing a TV series and doing special effects and oh, by the way, just happens to be a fantastic electronics engineer who uses the SX in many of his projects. He even maintains the SX-Key IDE for Parallax – how could we not get along?!

About three or four times a month Peter and I meet at one of our favorite restaurants in downtown Burbank, just a stone's throw from the Warner Brothers and Disney lots. The food is great, the service is great, and they never seem to mind that we will stay at the table long past the pasta, mostly talking about electronics. We usually have a little show-and-tell for each other, sharing current projects, and exchanging ideas. The meetings are always educational and, for me, it's the best way to "do lunch" in Hollywood.

Peter has been incredibly generous with his knowledge, particularly on a subject that I've been slow to approach: microcontroller networking. Sure, I've done very simple stuff, but having spent that last two Halloweens at Peter's home watching (with hundreds of others) his incredible animatronics display, I am pushing myself to jump in and give "real" microcontroller networking a go. Lucky for me I have the benefit of Peter's experience on this topic, as he's spent the last several years developing and improving his networked animatronics control system.

Several years back Peter set out to design a very flexible, fully modular animatronics control system that he could manage from a simple PC. Well, having seen it in action, I can tell you that he succeeded, and you can see for yourself by visiting his web site at http://www.socalhalloween.com. His system runs on an RS-485 network with several types of network nodes, the most sophisticated being the animation controller that is able to receive an animation frame while playing another (the servo control output of the animation controller uses an SX28).

My goals are somewhat less sophisticated than Peter's, though I've had them for quite some time. While I was living in Texas I read about man who built an enormous custom home; its size was somewhere on the order of 20,000 square feet. When he consulted the utilities companies they estimated that his monthly heating and air conditioning expenses would be around $4000. He figured for that much money he could create a custom home management system and when he did, his energy bills were reduced to under $400 per month. Along the way he discovered that a lot of "energy efficient" appliances were not performing to their stated specifications and he forced some manufacturers to restate their specs or fix the products.

Today the concept of "going green" is very popular, and it should be – a penny saved is a penny earned, especially when it's precious energy. So my system is going to be very straightforward with the ultimate goal to monitor and control my home from a simple PC; making it "smart" and, if I do it well, energy efficient.

As this is the beginning of what I expect to be a long journey, I'm borrowing another one of Peter's good ideas: I'm creating a prototyping system for an SX-based network node. What this means is that my generic network node PCB will have a processor and the RS-485 interface, and a small breadboarding area to add custom circuitry as needed for a given node. Let's have a look at the hardware.

Figure 147.1 shows the SX28 processor and RS-485 interface (MAX489 or equivalent). As you can see, it is in fact very generic. The design uses port RA for communications and ports RB and RC for I/O that is specific to the node. For the purposes of the rest of this article my node is going to be a 10-segment LED display. I'm starting with simple hardware so that I can get my head around the requirements of managing network messages. The node has dedicated RX and TX pins for the RS-485 link, so we can tie the MAX489 Receive Enable (/RE) pin to ground; that way the SX will always be "listening." On the other side, however, we will want to selectively activate the Data Enable (DE) pin so that RS-485 output from the node is active only when transmitting.

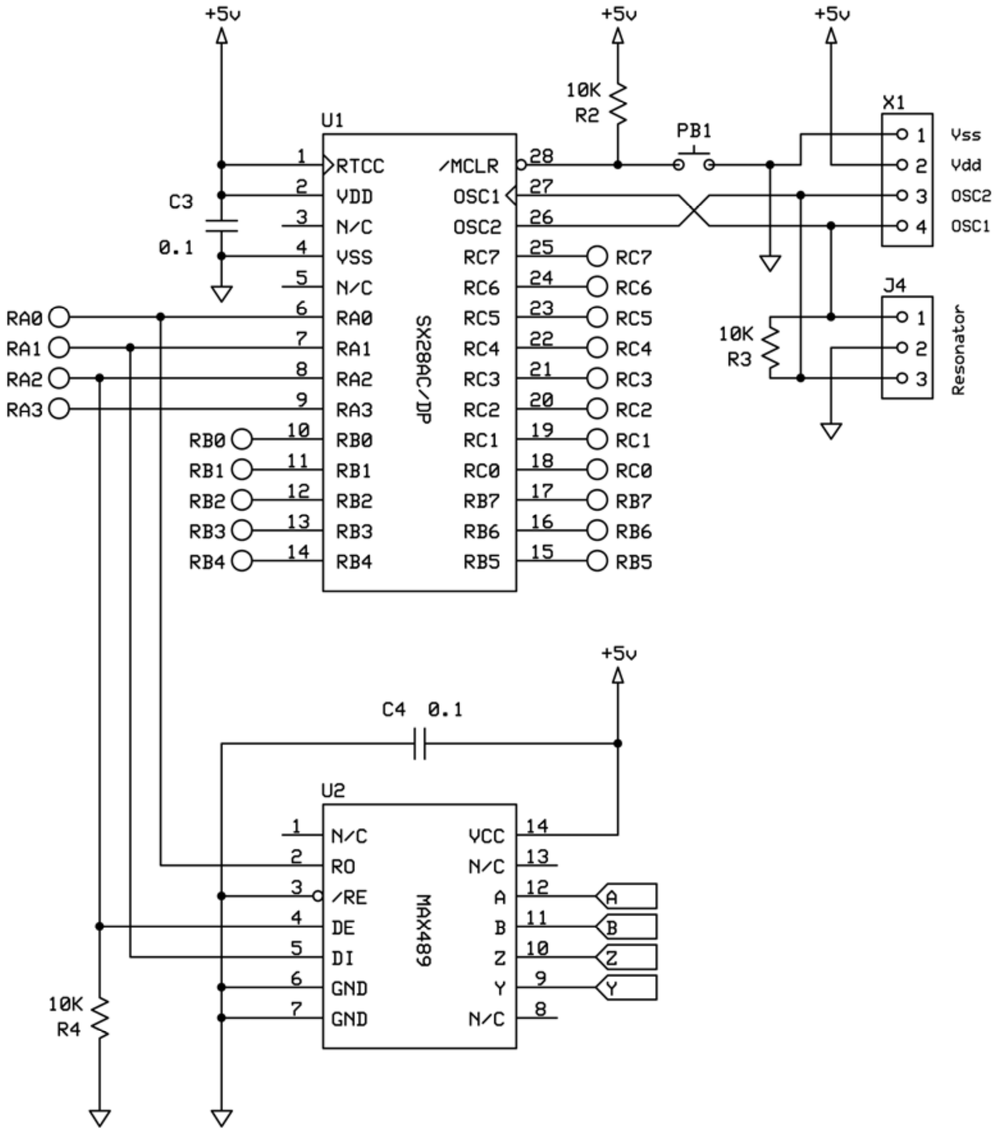


Figure 147.1: The SX28 processor and RS-485 Interface

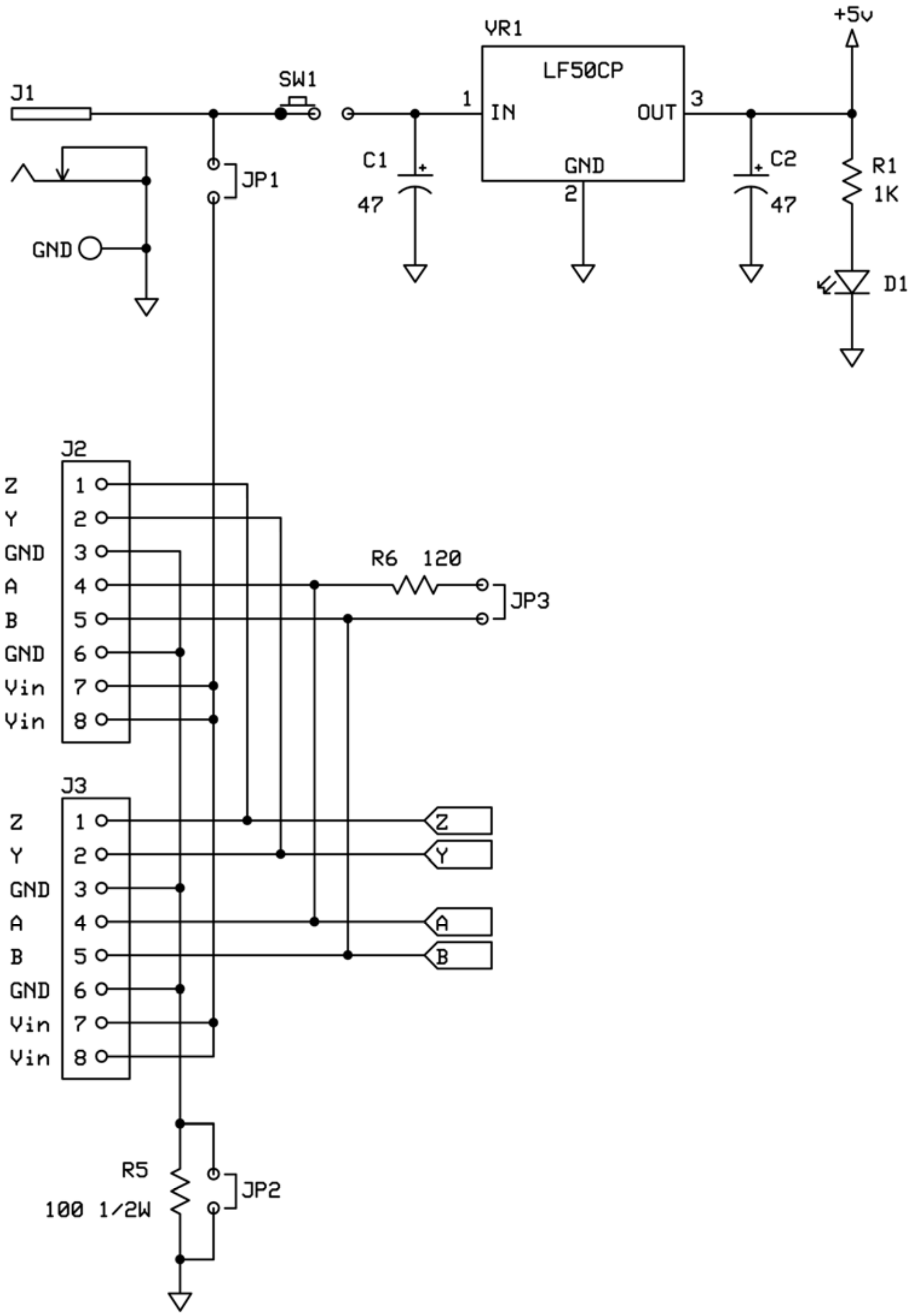


Figure 147.2: The Power Supply and RS-485 Connections

Figure 147.2 shows the power supply and RS-485 connections. Using RJ-45 jacks allows us to transmit full-duplex data on inexpensive CAT-5 cables. We can even put power on the cable to handle low-current nodes. When taking power from the CAT-5 cable jumpers JP1 and JP2 should be installed, otherwise they should be removed. If using local power you must remove JP1 – please be careful with this.

Jumper JP3 enables the receive line terminator. If a node is the last on the "receive" end then JP3 should be installed, otherwise is should be removed. I'm using a PC as my master node but there's no reason we can't have a network of SX-only nodes, with one being assigned as the master controller. If you're using an SX master then its JP3 should be installed.

Okay, the hardware is very simple, and that's by design as this is a prototyping system. One problem I did run into is the hole-count limitation when using ExpressPCB's mini board service. After getting my components laid out I just filled as much [logical] space as possible with standard pads. As I went to order boards I got a dialog that informed me I had too many holes for a mini board – so keep this in mind when you're prototyping with ExpressPCB.

Since most of my projects involve the SX I've become very comfortable with "virtual peripherals" and have created several code modules that I plug in as needed. A couple modules that get a lot of use in what I do are the buffered receive and transmit UARTs; these modules allow us to receive and transmit serial data in "the background" while our foreground code is doing other things. The receive UART for this project is a buffered version of what we used in the lighting controller we did last November. This project uses the complementary transmit UART that has a little addition to manage the MAX489 DE pin. Let's look at the modifications for controlling the MAX489.

```
Transmit:
  ASM
    BANK  txSerial
    CLRB  txDivide.BaudBit
    INC   txDivide
    JNB   txDivide.BaudBit, TX_Exit
    TEST  txCount
    JZ    TX_Buffer
    STC
    RR    txHi
    RR    txLo
    MOVB  TX, txLo.6
    DEC   txCount
    JMP   TX_Done

TX_Buffer:
    TEST  txBufCnt
    JZ    TX_Exit
    SETB  TxEnable
    MOV   W, #txBuf
    ADD   W, txTail
    MOV   FSR, W
    MOV   txHi, IND
    CLR   txLo
    MOV   txCount, #11
    INC   txTail
    CLRB  txTail.3
    DEC   txBufCnt
    JMP   TX_Done

TX_Exit:
    JNB   TxEnable, TX_Done
    CJA   txBufCnt, #0, TX_Done
    CJA   txCount, #0, TX_Done
    CLRB  TxEnable

TX_Done:
    BANK  0  ENDASM
```

One of the great aspects of SX/B is the ability to easily fold Assembly code segments into a BASIC program – that's what I did here; the UART code is really a modification of that found in Günther Daubachs' excellent book, *Programming the SX Microcontroller*. I modified the buffering to work within the same RAM bank as the other transmit variables and, for this project, included control of the MAX489.

In the section at **TX_Buffer** the DE pin (called TxEnable in the program) is taken high with **SETB** when a byte is about to be moved from the transmit ring buffer into the transmitter output (txHi). Since this byte won't start going out until the next interrupt, there is plenty of time for the DE pin to stabilize. The DE pin will stay high until the transmit buffer is empty (txBufCnt is 0) and there are no more bits to be transmitted (txCount is 0).

Okay, now that we can receive and transmit bytes in the background it's time to talk protocol. The neat part about this is we get to make it up, which in fact turns out to be the tough part too; sometimes it's just easier to work from an established specification. In my case I borrowed quite a lot from Peter's protocol, making a few changes that simplify the system and tie into my long-term goals.

The protocol is, essentially, peer-to-peer, so any node can talk to any other node. This opens the door to all kinds of interesting possibilities. The "sender" node will transmit a four-byte header that is followed by a data packet if required for the specific message.

The entire transmission is configured as follows:

Receiver	Receiver node (1 to 127) + $80– to designate start of header
Sender	Node sending the packet (1 to 127)
Message	Request or Command Message
Packet Size	Number of bytes in data packet (0 to n)
Data bytes	Data used by Message (optional)

The minimum transmission size will be four bytes (the header): the receiver, the sender, the message, and a zero when there are no data bytes. The receiver address will have BIT7 set to designate the start of a new header – the MIDI protocol uses this strategy and we're going to borrow from it.

The node we're going to create will be a simple I/O slave that will respond to [valid] commands and requests from another node. We'll use a VB program to send the messages from a PC. Since the node is a slave, it waits for bytes to show up in the receive buffer and then process them accordingly. The first part handles the basic message header.

```
Main:
  rxNode = RX_BYTE
  IF rxNode.7 = 0 THEN Main

Validate_Start:
  rxNode.7 = 0
  IF rxNode = GLOBAL_NODE THEN Get_Sender_Node
  IF rxNode <> MY_NODE THEN Main

Get_Sender_Node:
  txNode = RX_BYTE
  IF txNode.7 = 1 THEN
    rxNode = txNode
    GOTO Validate_Start
  ENDIF

Validate_Global_Sender:
  IF rxNode = GLOBAL_NODE THEN
    IF txNode <> MASTER_NODE THEN Main
  ENDIF

Get_Message:
  msgNum = RX_BYTE
  IF msgNum.7 = 1 THEN
    rxNode = msgNum
    GOTO Validate_Start
  ENDIF

Get_Packet_Length:
  packLen = RX_BYTE
  IF packLen.7 = 1 THEN
    rxNode = packLen
    GOTO Validate_Start
  ENDIF
```

When a byte comes in we need to check to see if BIT7 is set as this indicates the start of the header. When we get such a byte, BIT7 is cleared and we pull the next byte from the input buffer – this is the sender node. If the receive node was designated as global (address 0) the program ensures that the sender was the master; in my system only the master node is allowed to send global commands. Finally, the message number and packet length bytes are pulled from the stream.

Since it is possible for a transmission to be interrupted and then restarted, we must test every byte that comes in for BIT7 being set. By doing this check we can always re-sync the node with the start of a new header.

If the command or request includes data the packet length byte will be one or greater. I don't expect to have long packets in my home control system so the buffers are fairly small. Here's how we receive any data bytes:

```
RX_Raw_Packet:
  idx = 0
  DO WHILE idx < packLen
    tmpB1 = RX_BYTE
    IF tmpB1.7 = 1 THEN
      rxNode = tmpB1
      GOTO Validate_Start
    ELSE
      fifo(idx) = tmpB1
      INC idx
    ENDIF
  LOOP
```

As above each new byte is checked to ensure it's not a header start byte; if not it gets moved into a temporary array called fifo().

I know what you're thinking: "If we can't use BIT7, how do we transmit values greater than 127?" We're going to borrow another strategy from MIDI and use two bytes: the first byte will contain the lower seven bits of the eight-byte value and the second byte will hold BIT7. Remember, we don't always need all eight bits for a given command, so we only use this scheme when an eight-bit value is required.

After receiving the packet and any data bytes we will use a simple routing section to process the incoming transmission. By doing this we end up simplifying the message handlers.

```
Route_Message:
  IF msgNum = QRY_REQ THEN Unit_Acknowledge
  IF msgNum = DEV_RST THEN Device_Reset
  IF msgNum = SET_BIT THEN Set_One_Bit
  IF msgNum = GET_BIT THEN Get_One_Bit
  IF msgNum = WR_PORT THEN Write_Port
  IF msgNum = RD_PORT THEN Read_Port

  ' if we get here, message is not used by this node

Bad_Message:
  msgNum = MSG_NAK
  packLen = 0
  GOTO Unit_Reply
```

No mystery here: if the message is known used by this node then the program is routed to the appropriate handler, otherwise the response MSG_NAK is returned to the sender. Since we're now dealing with messages let's have a look at what's defined and explain the logic behind them.

```
QRY_REQ         CON     $01
QRY_ACK         CON     $02
MSG_ACK         CON     $03
MSG_NAK         CON     $04
MSG_FAIL        CON     $05
DEV_RST         CON     $0F

SET_BIT         CON     $10
GET_BIT         CON     $11
GET_BIT_ACK     CON     $12

WR_PORT         CON     $20
```

```
RD_PORT         CON     $21
RD_PORT_ACK     CON     $22

WR_CHAN         CON     $30
RD_CHAN         CON     $31
RD_CHAN_ACK     CON     $32
```

The first message, QRY_REQ, is used by the sender to “ping” the receiver; if the receiver is present then it responds with QRY_ACK. The next four messages are used to respond to commands or requests for data from the receiver. If node is able to complete a request and there is no data to be returned then it will respond with MSG_ACK. If the message sent isn’t used by the node then the response is MSG_NAK. If a valid message is sent with bad data (e.g., a bad port number) then the response will be MSG_FAIL. The final message in this lower group, DEV_RST, will usually be issued by the master to tell a node to reset itself.

For a simple I/O node I’ve defined three sets of commands: one for bit-level control, one for port-level control, and a third for setting values (called channels) within the program space. The set and write commands will respond with MSG_ACK, MSG_NAK, or MSG_FAIL as appropriate. The get and read commands have dedicated responses for the return data; the logic being this aids the “sender” side of the exchange when a lot of packets are flying around.

Let’s have a look at a few of the handlers.

```
Unit_Acknowledge:
  msgNum = QRY_ACK
  packLen = 0
  GOTO Unit_Reply
```

The **Unit_Acknowledge** handler is the simplest: is sets the message the QRY_ACK, the return packet length to zero, and then sends the reply. The reason for this process is to allow a “master” to poll all the expected “slave” devices to ensure that they’re actually online; there is no reason for sending command messages to a node that is not connected.

Now for something a little more interesting: we’ll accept a level for one of the I/O pins on the node. I happened to find a 10-segment bar-graph LED in my junk drawer so I soldered that onto the PCB. With just ten LEDs on the node the handler will only accept bit numbers between 0 (on RB.0) and 9 (on RC.1) – if you use more outputs be sure to adjust the code accordingly.

```
Set_One_Bit:
  tmpB1 = fifo(0)
  tmpB2 = fifo(1)

  IF tmpB1 < 10 THEN
    IF tmpB1 < 8 THEN
      tmpB1 = 1 << tmpB1
      IF tmpB2.0 = 1 THEN
        RB = RB | tmpB1
      ELSE
        tmpB1 = ~tmpB1
        RB = RB & tmpB1
      ENDIF
    ELSE
      tmpB1 = tmpB1 - 8
      tmpB1 = 1 << tmpB1
      IF tmpB2.0 = 1 THEN
        RC = RC | tmpB1
      ELSE
        tmpB1 = ~tmpB1
        RC = RC & tmpB1
      ENDIF
    ENDIF
    msgNum = MSG_ACK
    packLen = 0
  ELSE
    msgNum = MSG_FAIL
    packLen = 0
```

```
  ENDIF
  GOTO Unit_Reply
```

The **Set_One_Bit** handler pulls the bit number and bit level from the fifo() array. This message doesn't use "stuffed" data bytes as the 127 limit exceeds the pin count on the SX48. Now, if you want to add shift-registers so that there are more than 128 discrete outputs on the node then you'll need to modify this handler to accommodate the expansion.

The first test is of the bit number. Assuming it's valid for the node the program determines which I/O port (RB or RC) holds that bit. A mask is created and if BIT0 of the specified level is 1 the mask is ORed with the control port which makes the I/O pin go high. If BIT0 of the specified level is zero the mask is inverted and then ANDed with the control port which makes the I/O pin go low. The node will return MSG_ACK after the bit is manipulated – unless the bit number was bad, then it will return MSG_FAIL.

The WR_PORT and RD_PORT message deal with eight-byte values, so let's see how we receive and return them using the 7-bit container bytes in the packet.

```
Write_Port:
  tmpB1 = fifo(0)

  IF tmpB1 < 2 THEN
    tmpB2 = PACKW_TO_VAL fifo(1), fifo(2)
    IF tmpB1 = 0 THEN
      RB = tmpB2
    ELSE
      RC = tmpB2 & %00000011
    ENDIF
    msgNum = MSG_ACK
    packLen = 0
  ELSE
    msgNum = MSG_FAIL
    packLen = 0
  ENDIF
  GOTO Unit_Reply
```

The WR_PORT message requires three data bytes: the port number and two (seven-bit) bytes that make up the eight-bit value for the specified port. The first check, of course, is the port number. On my little I/O node RB is defined as port 0 and RC as port 1. If the specified port number is greater than one then we will abort with a MSG_FAIL response.

When the port number is valid then we'll use fifo(1) and fifo(2) to reconstruct the eight-bit value with **PACKW_TO_VAL**. This function expects two seven-bit bytes passed LSB, then MSB, and will return a properly-reconstructed word. In our program we'll only be using the low byte of the returned word, but you can reuse this code in a MIDI application as it will properly handle 14-bit values.

```
FUNC PACKW_TO_VAL
  tmpW1 = __WPARAM12

  tmpW1_LSB = tmpW1_LSB << 1
  tmpW1 = tmpW1 >> 1
  RETURN tmpW1
  ENDFUNC
```

Reconstructing a clean, 14-bit value from two seven-bit bytes is pretty easy. We move the bytes into tmpW1 and then shift the lower byte left by one to close the gap at BIT7. Now we can shift the entire word right by one to re-align everything to BIT0. That's it; the 14-bit value is reconstructed and can be returned to the caller.

The **Write_Port** handler will route the reconstructed byte to the appropriate port based on the contents of fifo(0). Since I'm only using two bits on RC the value is masked before it's written to that port.

The **Read_Port** handler allows us to read the state of a port on the SX. The sender will pass the port number and expects to get three data bytes back: the port number and two seven-bit bytes that will be reconstructed into a single eight-bit port value.

```
Read_Port:
  tmpB1 = fifo(0)

  IF tmpB1 < 2 THEN
    IF tmpB1 = 0 THEN
      tmpB2 = RB
    ELSE
      tmpB2 = RC & %00000011
    ENDIF
    tmpW1 = VAL_TO_PACKW tmpB2
    msgNum = RD_PORT_ACK
    packLen = 3
    fifo(1) = tmpW1_LSB
    fifo(2) = tmpW1_MSB
  ELSE
    msgNum = MSG_FAIL
    packLen = 3
    fifo(1) = 0
    fifo(2) = 0
  ENDIF
  GOTO Unit_Reply
```

This routine uses the **VAL_TO_PACKW** function to split the byte value into two seven-bit containers. To keep things simple we'll use a word variable to receive the return value from **VAL_TO_PACKW**.

```
FUNC VAL_TO_PACKW
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1
  ELSE
    tmpW1 = __WPARAM12
  ENDIF

  tmpW1 = tmpW1 << 1
  tmpW1_LSB = tmpW1_LSB >> 1
  tmpW1_MSB = tmpW1_MSB & $7F
  RETURN tmpW1
  ENDFUNC
```

This function is setup to accommodate bytes or words so that we can also use it in future MIDI applications. The value to split is moved into tmpW1 and then shifted left. This moves BIT7 of the lower byte into BIT0 of the upper. The next step is to shift the lower byte right by one to re-align its BIT0; BIT7 of the low byte will now be 0 as required by the protocol. The final step is to ensure that BIT7 of the high byte is clear before returning the new value.

We will move the low byte of the return value to fifo(1) and the high byte to fifo(2) – fifo(0) already holds the port number so we don't have to change that. The message is set to RD_PORT_ACK, the packet length to three, and then we send the response.

While chatting with Peter about networking he told me – and he was right – that designing these kinds of projects can turn into a bit of a chicken-and-egg dilemma. Testing a node requires another node, and writing the code for that requires specifications on both ends. Case in point is when I was sending a bad port number from my PC node; the slave node originally sent a MSG_FAIL packet (just four bytes) but my PC node was expecting success and waiting for seven. To keep things easy, and easy is usually best, the slave node will always have a three-byte packet for RD_PORT, even if the return message is MSG_FAIL. The port number is maintained so the master node can deal with it, and the incoming transmission processing is simplified by assigning an expected return message length to each command.

You've probably noticed that all message handlers jump to a routine called **Unit_Reply**. Here it is:

```
Unit_Reply:
  IF rxNode = MY_NODE THEN
    rxNode = txNode | $80
    TX_BYTE rxNode
    TX_BYTE MY_NODE
    TX_BYTE msgNum
    TX_BYTE packLen
    idx = 0
    DO WHILE idx < packLen
      TX_BYTE fifo(idx)
      INC idx
    LOOP
  ENDIF
  GOTO Main
```

The only time a node will send a response is when the node is individually addressed. A node could, for example, send a message to the global address of 0 that all nodes react to; in this case there will be no responses from the nodes as they would likely end up stomping on each other and the messages would be trashed. You can see that **Unit_Reply** takes the sender node address and turns it into the header start byte by setting BIT7.

Okay, now we have the makings of a reasonably sophisticated control network using the SX, and can do all kinds of cool things with it. Figure 147.3 shows my first prototype node along with a port-powered RS-485 interface, and Figure 147.4 shows the VB test node for experimenting with messages (the compiled program and source code is included in the download files for the article).

In the Query frame you can see four primary values; these comprise the message header. The middle row of inputs allows for uncompressed data. The lower row of [red] boxes show the actual packet bytes transmitted to the receiver node. The Response frame is similarly constructed, except that the middle line holds seven-bit "packed" values and the lower [green] boxes hold the reconstructed bytes.

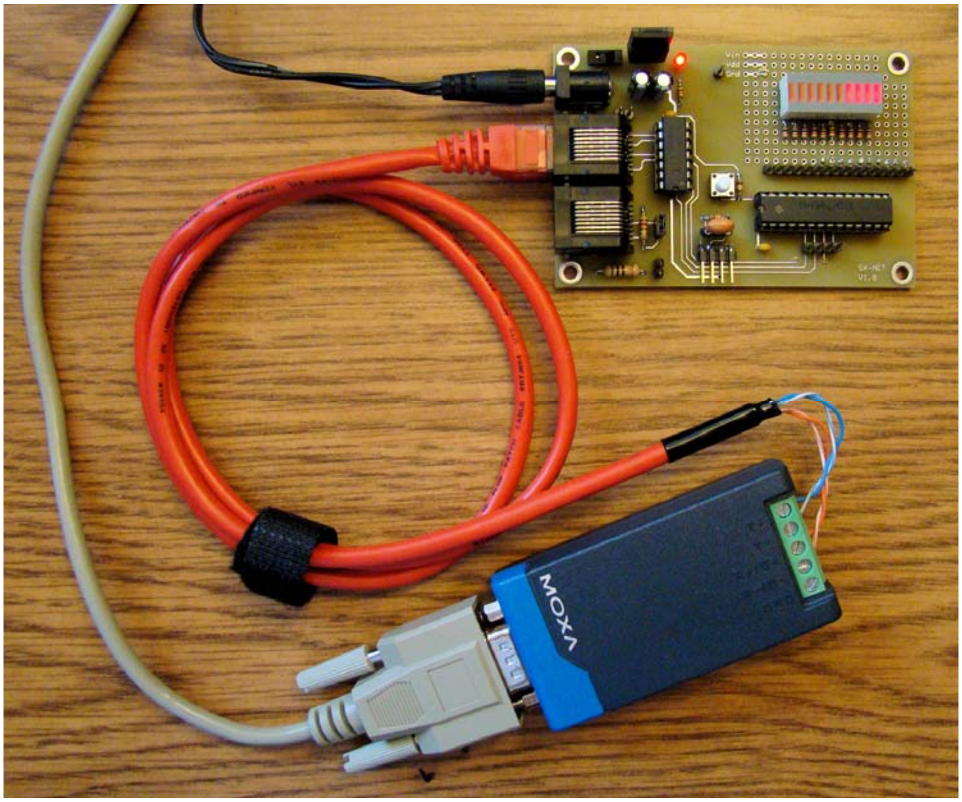
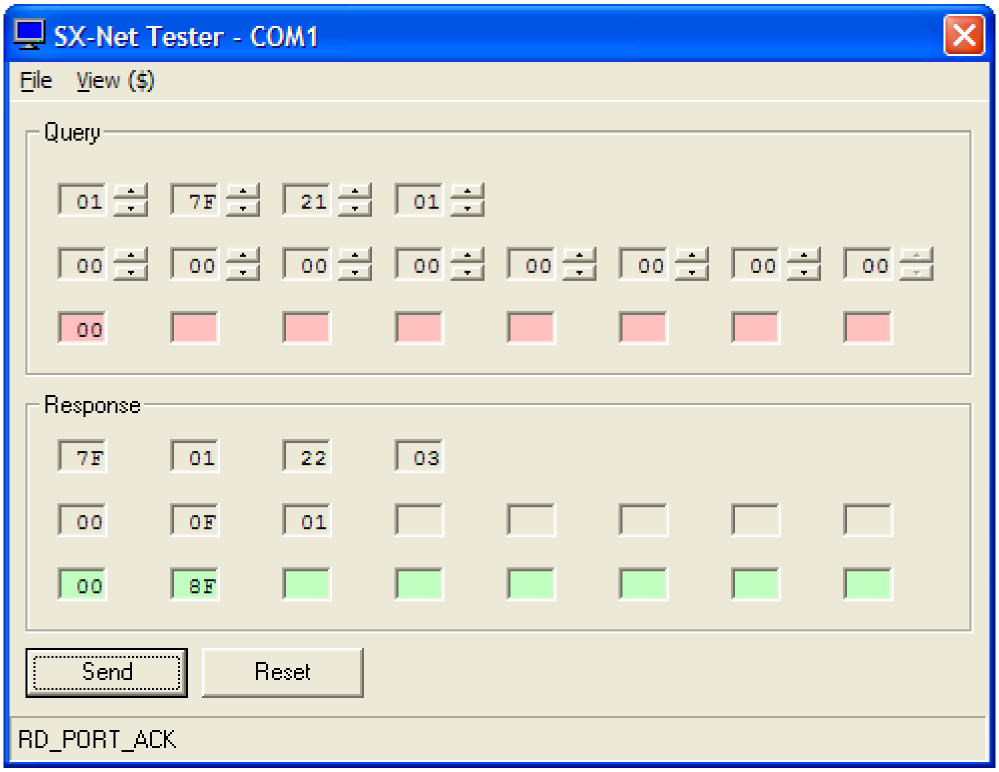


Figure 147.3: Prototype Node and Port-Powered RS-485 Interface



Query

01 7F 21 01

00 00 00 00 00 00 00 00

00

Response

7F 01 22 03

00 0F 01

00 8F

RD_PORT_ACK

Figure 147.4: The VB Test Node for Experimenting with Messages

The Big Squeeze

At some point you will probably want to create a node that requires more than one eight-bit value for a message and you don't want to use two bytes for each. Peter came up with a neat compression solution for his network and I've created so I can use it. Have a look at Figure 5 to see how Peter compresses several eight-bit values into seven-bit containers.

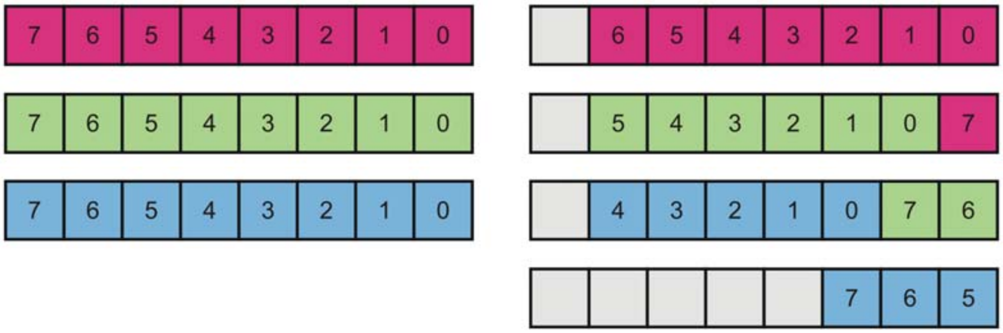


Figure 147.5

Since we will typically manipulate blocks of values I created a subroutine called **UNSQUEEZE** that will take two bytes from an input buffer and move them into a single byte of an output buffer. To use this routine we will pass a pointer to the start of the input buffer, an offset for the desired value, and a pointer to the start of the output buffer.

```
SUB UNSQUEEZE
  src = __PARAM1
  offset = __PARAM2
  dest = __PARAM3
```

```
  src = src + offset
  dest = dest + offset
  dByte = __RAM(src)
  dByte = dByte >> offset
  INC src
  dbMSB = __RAM(src)
  offset = 7 - offset
  dbMSB = dbMSB << offset
  dByte = dByte | dbMSB
  __RAM(dest) = dByte
  ENDSUB
```

The actual source and destination addresses are incremented by the offset to get to the LSB of the target byte. By doing this math in the subroutine we simplify the interface to it – we don't have to remember the @ (address of) operator with the array name, we just use the name on its own. The low bits of the output byte are retrieved using the __RAM() array and shifted right by the offset to re-align BIT0. The source address is then incremented to get to the upper bits. This value is shifted left to move the bits to the correct position and then the two bytes are ORed together to reconstruct the eight-bit value. Finally, the __RAM() array is used to move the reconstituted value to the desired output address.

The complement of **UNSQUEEZE** is – no big surprise – **SQUEEZE**; we can use this to create a compressed packet to send to another node.

```
SUB SQUEEZE
  src = __PARAM1
  offset = __PARAM2
  dest = __PARAM3

  src = src + offset
  dest = dest + offset
  dByte = __RAM(src)
  dbCopy = dByte
  destVal = __RAM(dest)
  dByte = dByte << offset
  dByte = dByte & $7F
  destVal = destVal | dByte
  __RAM(dest) = destVal
  INC dest
  offset = 7 - offset
  destVal = dbCopy >> offset
  __RAM(dest) = destVal
  ENDSUB
```

With **SQUEEZE** the eight-bit value will be split based on its position in the output array, with the lower half ORed into the output array so that any previous values there are not disturbed. Let me suggest that the **FILL** subroutine be used to clear the output array before looping through the input array – in order – to create the compressed packet. It's a little bit of code but now we can send seven full bytes using eight instead of 14, and this can be important if we have a lot of network traffic.

Okay, I think we should probably wrap it up right here. Order your boards, build a simple node, and start experimenting. I'd love to hear your ideas on home control, especially those ideas that allow us to conserve energy.

Until next time, Happy Networking with the SX!

SX-Net Prototyping Node Bill of Materials		
Designator	**Value**	**Source**
C1, C2	47µF	Mouser 647-UVR1V470MDD
C3, C4	0.1 µF	Mouser 80-C315C104M5U
D1	LED	Mouser 859-LTL-4222N
J1	2.1mm barrel	Mouser 806-KLDX-0202-A
J2-J3	RJ-45 R/A	Mouser 571-5202514
J4	Machine pin	Mouser 506-510-AG91D
JP1-JP4	Pin Strip Header	Mouser 517-6111TG
Jumpers	0.1" shunt	Mouser 538-15-29-1024
PB1	N.O. button	Mouser 612-TL59F160Q
PCB		ExpressPCB.com
R1	1 K	Mouser 299-1K-RC
R2-R4	10 K	Mouser 299-10K-RC
R5	100 1/2 W	Mouser 293-100-RC
R6-R7	120	Mouser 291-120-RC
RES1	20 MHz	Parallax 250-02060
S1	28-pin DIP	Mouser 571-1-390261-9
S2	14-pin DIP	Mouser 571-1-390261-3
SW1	slide switch	Mouser 506-SSA12
U1	SX28AC/DP	Parallax SX28AC/DP
U2	MAX489	Mouser 837-ISL8489IP
VR1	LF50CP	Mouser 511-LF50CP
X1	R/A Header	Mouser 517-5111TG

```
' =========================================================================
'
'   File...... SX-NET.SXB
'   Purpose...
'   Author.... Jon Williams, EFX-TEK
'              Copyright (c) 2007 EFX-TEK
'              Some Rights Reserved
'              -- see http://creativecommons.org/licenses/by/3.0/
'   E-mail.... jwilliams@efx-tek.com
'   Started...
'   Updated... 18 NOV 2007
'
' =========================================================================


' -------------------------------------------------------------------------
' Program Description
' -------------------------------------------------------------------------



' -------------------------------------------------------------------------
' Conditional Compilation Symbols
' -------------------------------------------------------------------------

'{$DEFINE TESTING_OFF}


' -------------------------------------------------------------------------
' Device Settings
' -------------------------------------------------------------------------
```

```
DEVICE          SX28, OSCXT2, TURBO, STACKX, OPTIONX, BOR42
FREQ            20_000_000
ID              "SX-Net"


' -------------------------------------------------------------------------
' I/O Pins
' -------------------------------------------------------------------------

RX             PIN     RA.0 INPUT  PULLUP
TX             PIN     RA.1 OUTPUT
TxEnable       PIN     RA.2 OUTPUT
UnusedRA3      PIN     RA.3 INPUT  PULLUP

Port0          PIN     RB
 Led0          PIN     Port0.0 OUTPUT
 Led1          PIN     Port0.1 OUTPUT
 Led2          PIN     Port0.2 OUTPUT
 Led3          PIN     Port0.3 OUTPUT
 Led4          PIN     Port0.4 OUTPUT
 Led5          PIN     Port0.5 OUTPUT
 Led6          PIN     Port0.6 OUTPUT
 Led7          PIN     Port0.7 OUTPUT

Port1          PIN     RC
 Led8          PIN     Port1.0 OUTPUT
 Led9          PIN     Port1.1 OUTPUT
 UnusedP1_2    PIN     Port1.2 INPUT  PULLUP
 UnusedP1_3    PIN     Port1.3 INPUT  PULLUP
 UnusedP1_4    PIN     Port1.4 INPUT  PULLUP
 UnusedP1_5    PIN     Port1.5 INPUT  PULLUP
 UnusedP1_6    PIN     Port1.6 INPUT  PULLUP
 UnusedP1_7    PIN     Port1.7 INPUT  PULLUP


' -------------------------------------------------------------------------
' Constants
' -------------------------------------------------------------------------

GLOBAL_NODE    CON     $00                     ' all units listen
MY_NODE        CON     $01                     ' slave addres (1 - 126)
MASTER_NODE    CON     $7F                     ' master is #127

' network messages
' -- those marked with * use compressed data

QRY_REQ        CON     $01                     ' unit query
QRY_ACK        CON     $02                     ' query ackowledge
MSG_ACK        CON     $03                     ' msg okay
MSG_NAK        CON     $04                     ' msg not used by node
MSG_FAIL       CON     $05                     ' msg had bad data
DEV_RST        CON     $0F                     ' unit reset

SET_BIT        CON     $10                     ' pass bit# + level
GET_BIT        CON     $11                     ' pass bit#
GET_BIT_ACK    CON     $12

WR_PORT        CON     $20                     ' pass port# + value  (*)
RD_PORT        CON     $21                     ' pass port#
RD_PORT_ACK    CON     $22

WR_CHAN        CON     $30                     ' pass chan# + value  (*)
RD_CHAN        CON     $31                     ' pass chan#
RD_CHAN_ACK    CON     $32


' Bit dividers for 6.51 uS interrupt

Baud2400       CON     6                       ' for ISR bit divisor
Baud4800       CON     5
Baud9600       CON     4
```

```
Baud19K2       CON    3
Baud38K4       CON    2

BaudBit        CON    Baud38K4               ' set baud rate
Baud1x0        CON    1 << BaudBit           ' calculate # ISR cycles
Baud1x5        CON    Baud1x0 * 3 / 2        ' start bit cycles


IsOff          CON    0
IsOn           CON    1


' -------------------------------------------------------------------------
' Variables
' -------------------------------------------------------------------------

flags          VAR    Byte
 isrFlag       VAR    flags.0                ' marks start of ISR
 rxReady       VAR    flags.1                ' indicates rx byte ready

rxNode         VAR    Byte                   ' intended receiver
txNode         VAR    Byte                   ' source node (1 - 127)
msgNum         VAR    Byte                   ' msg number (1 - 127)
packLen        VAR    Byte                   ' bytes in packet

idx            VAR    Byte

tmpW1          VAR    Word                   ' for subs/funcs
tmpW2          VAR    Word
tmpB1          VAR    Byte
tmpB2          VAR    Byte
tmpB3          VAR    Byte
tmpB4          VAR    Byte
tmpB5          VAR    tmpW1_LSB
tmpB6          VAR    tmpW1_MSB
tmpB7          VAR    tmpW2_LSB
tmpB8          VAR    tmpW2_MSB

' aliases for SQUEEZE / UNSQUEEZE

src            VAR    tmpB1
offset         VAR    tmpB2
dest           VAR    tmpB3
dByte          VAR    tmpB4                  ' 8-bit data byte
dbCopy         VAR    tmpB5                  ' copy of data byte
dbMSB          VAR    tmpB6                  ' for (unpack)
destVal        VAR    tmpB7                  ' value in dest(offset)


rxSerial       VAR    Byte (16)
 rxBuf         VAR    rxSerial(0)            ' 8-byte buffer
 rxCount       VAR    rxSerial(8)            ' rx bit count
 rxDivide      VAR    rxSerial(9)            ' bit divisor timer
 rxByte        VAR    rxSerial(10)           ' recevied byte
 rxHead        VAR     rxSerial(11)          ' buffer head (write to)
 rxTail        VAR    rxSerial(12)           ' buffer tail (read from)
 rxBufCnt      VAR    rxSerial(13)           ' # bytes in buffer

txSerial       VAR    Byte (16)              ' tx serial data
 txBuf         VAR     txSerial(0)           ' eight-byte buffer
 txCount       VAR     txSerial(8)           ' tx bit count
 txDivide      VAR     txSerial(9)           ' bit divisor timer
 txLo          VAR     txSerial(10)          ' holds start bit
 txHi          VAR     txSerial(11)          ' tx output reg
 txHead        VAR     txSerial(12)          ' buffer head (write to)
 txTail        VAR    txSerial(13)           ' buffer tail (read from)
 txBufCnt      VAR    txSerial(14)           ' # bytes in buffer

fifo           VAR    Byte (8)
packet         VAR    Byte (8)
```

```
' =========================================================================
  INTERRUPT NOPRESERVE 153_600               ' run every 6.51 uS
' =========================================================================

' --------------------------------
' Mark ISR - use for timing events
' --------------------------------
'
Marker:
  ASM
    SETB  isrFlag                            ' (1)
  ENDASM


' -------
' RX UART
' -------

Receive:
  ASM
    BANK  rxSerial                           ' (1)
    JB    rxBufCnt.4, RX_Done                ' (2/4) skip if buffer is full
    MOVB  C, RX                              ' (4)   sample serial input
    TEST  rxCount                            ' (1)   receiving now?
    JNZ   RX_Bit                             ' (2/4) yes, get next bit
    MOV   W, #9                              ' (1)   no, prep for next byte
    SC                                       ' (1/2)
    MOV   rxCount, W                         ' (1)   if start, load  bit count
    MOV   rxDivide, #Baud1x5                 ' (2)   prep for 1.5 bit periods

RX_Bit:
    DJNZ  rxDivide, RX_Done                  ' (2/4) complete bit cycle?
    MOV   rxDivide, #Baud1x0                 ' (2)   yes, reload bit timer
    DEC   rxCount                            ' (1)   update bit count
    SZ                                       ' (1/2)
    RR    rxByte                             ' (1)   position for next bit
    SZ                                       ' (1/2)
    JMP   RX_Done                            ' (3)

RX_Buffer:
    MOV   W, #rxBuf                          ' (1)   point to buffer head
    ADD   W, rxHead                          ' (1)
    MOV   FSR, W                             ' (1)
    MOV   IND, rxByte                        ' (2)   move rxByte to head
    INC   rxHead                             ' (1)   update head
    CLRB  rxHead.3                           ' (1)   keep 0..7
    INC   rxBufCnt                           ' (1)   update buffer count
    SETB  rxReady                            ' (1)   set ready flag

RX_Done:
    BANK  0                                  ' (1)
  ENDASM


' -------
' TX UART
' -------
'
Transmit:
  ASM
    BANK  txSerial                           ' (1)
    CLRB  txDivide.BaudBit                   ' (1)   clear tx bit flag
    INC   txDivide                           ' (1)   update tx bit timer
    JNB   txDivide.BaudBit, TX_Exit          ' (2/4)
    TEST  txCount                            ' (1)   transmitting now?
    JZ    TX_Buffer                          ' (2/4) if txCount = 0, no
    STC                                      ' (1)   set for stop bit
    RR    txHi                               ' (1)   rotate TX buf
    RR    txLo                               ' (1)
    MOVB  TX, txLo.6                         ' (4)   output the bit
```

```
    DEC   txCount                          ' (1)   update the bit count
    JMP   TX_Done                          ' (3)

TX_Buffer:
    TEST  txBufCnt                         ' (1)   anything in buffer?
    JZ    TX_Exit                          ' (2/4) exit if empty
    SETB  TxEnable                         ' (1)   enable transmitter
    MOV   W, #txBuf                        ' (2)   point to buffer tail
    ADD   W, txTail                        ' (1)
    MOV   FSR, W                           ' (1)
    MOV   txHi, IND                        ' (2)   move byte to TX reg
    CLR   txLo                             ' (1)   clear for start bit
    MOV   txCount, #11                     ' (2)   start + 8 + 2 stop
    INC   txTail                           ' (1)   update tail pointer
    CLRB  txTail.3                         ' (1)   keep 0..7
    DEC   txBufCnt                         ' (1)   update buffer count
    JMP   TX_Done                          ' (3)

TX_Exit:
    JNB   TxEnable, TX_Done                ' (2/4) skip if enable clear
    CJA   txBufCnt, #0, TX_Done            ' (4/6) skip if buffer not empty
    CJA   txCount, #0, TX_Done             ' (4/6) skip if still transmitting
    CLRB  TxEnable                         ' (1)   disable TX

TX_Done:
    BANK  0                                ' (1)
  ENDASM

  RETURNINT


' =========================================================================
  PROGRAM Start
' =========================================================================


' -------------------------------------------------------------------------
' Subroutine / Function Declarations
' -------------------------------------------------------------------------

DELAY_MS       SUB    1, 2                 ' delay in milliseconds
DELAY_TIX      SUB    1, 2                 ' delay in 6.51 uS units

RX_BYTE        FUNC   1, 0                 ' receive a byte
TX_BYTE        SUB    1                    ' transmit a byte

VAL_TO_PACKW   FUNC   2, 1, 2              ' value to packed word
PACKW_TO_VAL   FUNC   2, 2                 ' packed word to byte

FILL           SUB    3                    ' fill RAM with value
SQUEEZE        SUB    3                    ' compress bytes (8 --> 7)
UNSQUEEZE      SUB    3                    ' decompress bytes (7 --> 8)


' -------------------------------------------------------------------------
' Program Code
' -------------------------------------------------------------------------

Start:
  TX = 1                                   ' set TX to idle state

Main:
'{$IFDEF TESTING_ON}
  GOTO Testing_123
'{$ENDIF}

  rxNode = RX_BYTE                         ' get receiver node
  IF rxNode.7 = 0 THEN Main                ' try again if no start

Validate_Start:
  rxNode.7 = 0                             ' strip start flag
```

```
  IF rxNode = GLOBAL_NODE THEN Get_Sender_Node       ' respond to global node
  IF rxNode <> MY_NODE THEN Main             ' validate node #

Get_Sender_Node:
  txNode = RX_BYTE                           ' rx sender node #
  IF txNode.7 = 1 THEN                       ' restart of packet?
    rxNode = txNode                          '  yes, send back to top
    GOTO Validate_Start
  ENDIF

Validate_Global_Sender:
  IF rxNode = GLOBAL_NODE THEN               ' if receiver is all
    IF txNode <> MASTER_NODE THEN Main       ' validate source is master
  ENDIF

Get_Message:
  msgNum = RX_BYTE                           ' rx message #
  IF msgNum.7 = 1 THEN
    rxNode = msgNum
    GOTO Validate_Start
  ENDIF

Get_Packet_Length:
  packLen = RX_BYTE                          ' rx packet length
  IF packLen.7 = 1 THEN
    rxNode = packLen
    GOTO Validate_Start
  ENDIF

RX_Raw_Packet:
  idx = 0
  DO WHILE idx < packLen
    tmpB1 = RX_BYTE                          ' get packet byte
    IF tmpB1.7 = 1 THEN                      ' check for restart
      rxNode = tmpB1
      GOTO Validate_Start
    ELSE
      fifo(idx) = tmpB1                      ' move to buffer
      INC idx
    ENDIF
  LOOP


' ----------------
' Manual Test Data
' ----------------
'
Testing_123:
'{$IFDEF TESTING_ON}
  rxNode = MY_NODE
  txNode = MASTER_NODE
  msgNum = WR_PORT
  packLen = 3
  fifo(0) = 0
  fifo(1) = $70
  fifo(2) = $01
'{$ENDIF}


' --------------
' Message Router
' --------------

Route_Message:
  IF msgNum = QRY_REQ THEN Unit_Acknowledge
  IF msgNum = DEV_RST THEN Device_Reset
  IF msgNum = SET_BIT THEN Set_One_Bit
  IF msgNum = GET_BIT THEN Get_One_Bit
  IF msgNum = WR_PORT THEN Write_Port
  IF msgNum = RD_PORT THEN Read_Port
```

```
  ' if we get here, message is not used by this node

Bad_Message:
  msgNum = MSG_NAK
  packLen = 0
  GOTO Unit_Reply


' ****************
' QRY_REQ
' ****************
'
Unit_Acknowledge:
  msgNum = QRY_ACK                            ' send a response
  packLen = 0
  GOTO Unit_Reply


' ****************
' DEV_RESET
' ****************
'
Device_Reset:
  IF txNode = MASTER_NODE THEN                ' only the master can reset me
    Port0 = IsOff                             ' clear the leds
    Port1 = IsOff
    msgNum = MSG_ACK
    packLen = 0
  ELSE
    msgNum = MSG_NAK
    packLen = 0
  ENDIF
  GOTO Unit_Reply


' ****************
' SET_BIT
' ****************
' -- bit # in fifo(0)
' -- value in fifo(1).0
'
Set_One_Bit:
  tmpB1 = fifo(0)                             ' get bit #
  tmpB2 = fifo(1)                             ' level (in bit 0)

  IF tmpB1 < 10 THEN                          ' valid?
    IF tmpB1 < 8 THEN                         ' on RB
      tmpB1 = 1 << tmpB1                      ' create bit mask
      IF tmpB2.0 = 1 THEN                     ' if set
        RB = RB | tmpB1                       '  do it
      ELSE
        tmpB1 = ~tmpB1                        ' otherwise invert mask
        RB = RB & tmpB1                       '  clear selected bit
      ENDIF
    ELSE
      tmpB1 = tmpB1 - 8                       ' adjust for RC
      tmpB1 = 1 << tmpB1
      IF tmpB2.0 = 1 THEN
        RC = RC | tmpB1
      ELSE
        tmpB1 = ~tmpB1
        RC = RC & tmpB1
      ENDIF
    ENDIF
    msgNum = MSG_ACK                          ' bit # okay, output updated
    packLen = 0
  ELSE
    msgNum = MSG_FAIL                         ' invalid bit # sent
    packLen = 0
  ENDIF
  GOTO Unit_Reply
```

```
' ***************
' GET_BIT
' ***************
' -- bit # in fifo(0)
' -- returns bit # in fifo(0) and value (0 or 1) in fifo(1)
'
Get_One_Bit:
  tmpB1 = fifo(0)                             ' bet bit #

  IF tmpB1 < 10 THEN                          ' valid?
    IF tmpB1 < 8 THEN                         ' RB?
      tmpB1 = 1 << tmpB1                      ' create mask
      tmpB1 = RB & tmpB1                      ' read the bit
    ELSE
      tmpB1 = tmpB1 - 8                       ' adjust for RC
      tmpB1 = 1 << tmpB1
      tmpB1 = RC & tmpB1
    ENDIF
    msgNum = GET_BIT_ACK
    packLen = 2                               ' bit and value
    IF tmpB1 > 0 THEN
      fifo(1) = 1                             ' put into buffer
    ELSE
      fifo(0) = 0
    ENDIF
  ELSE
    msgNum = MSG_FAIL                         ' bad bit #
    packLen = 2
    FILL fifo, 0, 2                           ' clear unused return bytes
  ENDIF
  GOTO Unit_Reply


' ***************
' WR_PORT
' ***************
' -- port # in fifo(0), compressed value in fifo(1),fifo(2)
'
Write_Port:
  tmpB1 = fifo(0)                             ' get port #

  IF tmpB1 < 2 THEN                           ' valid?
    tmpB2 = PACKW_TO_VAL fifo(1), fifo(2)     ' convert to byte
    IF tmpB1 = 0 THEN
      RB = tmpB2
    ELSE
      RC = tmpB2 & %00000011                  ' update available bits
    ENDIF
    msgNum = MSG_ACK
    packLen = 0
  ELSE
    msgNum = MSG_FAIL
    packLen = 0
  ENDIF
  GOTO Unit_Reply


' ***************
' RD_PORT
' ***************
' -- port # in fifo(0)
' -- returns port# in fifo(0), compressed value in fifo(1),fifo(2)
'
Read_Port:
  tmpB1 = fifo(0)                             ' get port #

  IF tmpB1 < 2 THEN                           ' valid?
    IF tmpB1 = 0 THEN
      tmpB2 = RB
```

```
    ELSE
      tmpB2 = RC & %00000011                ' mask unused bits
    ENDIF
    tmpW1 = VAL_TO_PACKW tmpB2              ' pack bits
    msgNum = RD_PORT_ACK
    packLen = 3
    fifo(1) = tmpW1_LSB
    fifo(2) = tmpW1_MSB
  ELSE
    msgNum = MSG_FAIL                       ' bad port #
    packLen = 3
    fifo(1) = 0
    fifo(2) = 0
  ENDIF
  GOTO Unit_Reply


' Send reply and any data to originator node
' -- only when message was for this node
'
Unit_Reply:
  IF rxNode = MY_NODE THEN
    rxNode = txNode | $80                   ' return to sender
    TX_BYTE rxNode
    TX_BYTE MY_NODE
    TX_BYTE msgNum
    TX_BYTE packLen
    idx = 0
    DO WHILE idx < packLen
      TX_BYTE fifo(idx)
      INC idx
    LOOP
  ENDIF
  GOTO Main


' -------------------------------------------------------------------------
' Subroutine / Function Code
' -------------------------------------------------------------------------

' Use: DELAY_MS duration
' -- delay in milliseconds
' -- ideal 1 ms timer reload value is 153.6; code attempts to compensate

SUB DELAY_MS
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                        ' save byte parameter
  ELSE
    tmpW1 = __WPARAM12                      ' save word parameter
  ENDIF
  DO WHILE tmpW1 > 0
    tmpB1 = 153 + tmpW1_LSB.0               ' load 1 ms timer
    DO WHILE tmpB1 > 0                      ' let timer expire
      \ CLRB isrFlag                        ' clear ISR flag
      \ JNB  isrFlag, @$                    ' wait for flag to be set
      DEC tmpB1                             ' update 1 ms timer
    LOOP
    DEC tmpW1                               ' update delay timer
  LOOP
  ENDSUB

' -------------------------------------------------------------------------

' Use: DELAY_TIX units
' -- delay 6.51 uS units

SUB DELAY_TIX
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                        ' save byte parameter
  ELSE
    tmpW1 = __WPARAM12                      ' save word parameter
```

```
  ENDIF
  DO WHILE tmpW1 > 0
    \ CLRB isrFlag                           ' clear ISR flag
    \ JNB  isrFlag, @$                       ' wait for flag to be set
    DEC tmpW1                                ' update delay timer
  LOOP
  ENDSUB

' -------------------------------------------------------------------------

' Use: aByte = RX_BYTE
' -- returns "aByte" from 8-byte circular buffer
' -- will wait if buffer is presently empty
' -- rxBufCnt holds byte count of receive buffer (0 to 8)

FUNC RX_BYTE
  ASM
    BANK  rxSerial
    TEST  rxBufCnt                           ' check buffer count
    JZ    @RX_BYTE                           ' wait if empty
    MOV   W, #rxBuf                          ' point to tail
    ADD   W, rxTail
    MOV   FSR, W
    MOV   __PARAM1, IND                      ' get byte at tail
    INC   rxTail                             ' update tail
    CLRB  rxTail.3                           ' keep 0..7
    DEC   rxBufCnt                           ' update buffer count
    TEST  rxBufCnt                           ' check the count
    SNZ                                      ' exit if not zero
    CLRB  rxReady                            ' else clear ready flag
    BANK  0
  ENDASM
  ENDFUNC

' -------------------------------------------------------------------------

' Use: TX_BYTE aByte
' -- moves "aByte" to 8-byte circular buffer (when space is available)
' -- will wait if buffer is presently full
' -- txBufCnt holds byte count of transmit buffer (0 to 8)

SUB TX_BYTE
  ASM
    BANK  txSerial                           ' point to tx vars
    JB    txBufCnt.3, @TX_BYTE               ' prevent buffer overrun
    MOV   W, #txBuf                          ' point to buffer head
    ADD   W, txHead
    MOV   FSR, W
    MOV   IND, __PARAM1                      ' move byte to tx buf
    INC   txHead                             ' update head pointer
    CLRB  txHead.3                           ' keep 0..7
    INC   txBufCnt                           ' update buffer count
    BANK  0
  ENDASM
  ENDSUB

' -------------------------------------------------------------------------

' Use: FILL *target, value, count
' -- fills RAM locations at "target" with "value"
' -- "target" is a RAM pointer

SUB FILL
  tmpB1 = __PARAM1                           ' pointer to target
  tmpB2 = __PARAM2                           ' value to write
  tmpB3 = __PARAM3                           ' byte count

  DO WHILE tmpB3 > 0
    __RAM(tmpB1) = tmpB2
    INC tmpB1
    DEC tmpB3
```

```
  LOOP
  ENDSUB

' --------------------------------------------------------------------------

' Use: wResult = VAL_TO_PACKW value
' -- MIDI style byte packing (puts 14-bit value into two 7-bit bytes)

FUNC VAL_TO_PACKW
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                         ' byte value
  ELSE
    tmpW1 = __WPARAM12                       ' word value
  ENDIF

  tmpW1 = tmpW1 << 1                         ' shift upper bits
  tmpW1_LSB = tmpW1_LSB >> 1                 ' correct lower bits
  tmpW1_MSB = tmpW1_MSB & $7F                ' mask MSB of uppter bits
  RETURN tmpW1
  ENDFUNC

' --------------------------------------------------------------------------

' Use: value = PACKW_TO_VAL packed
' -- MIDI-style unpacking

FUNC PACKW_TO_VAL
  tmpW1 = __WPARAM12

  tmpW1_LSB = tmpW1_LSB << 1                 ' close "gap"
  tmpW1 = tmpW1 >> 1                         ' re-align value
  RETURN tmpW1
  ENDFUNC

' --------------------------------------------------------------------------

' Use: SQUEEZE *source, offset, *destination
' -- source(offset) is 8-bit data byte
' -- destination(offset), destination(offset+1) holds 7-bit packed value

SUB SQUEEZE
  src = __PARAM1                             ' pointer to src(0)
  offset = __PARAM2                          ' offset into src
  dest = __PARAM3                            ' poiner to dest(0)

  src = src + offset                         ' point to dByte
  dest = dest + offset                       ' point to dest (LSB)
  dByte = __RAM(src)                         ' get 8-bit value
  dbCopy = dByte                             ' make a copy (for high bits)
  destVal = __RAM(dest)                      ' get current dest value
  dByte = dByte << offset                    ' adjust low bits
  dByte = dByte & $7F                        ' strip MSB
  destVal = destVal | dByte                  ' overlay new bits
  __RAM(dest) = destVal                      ' write updated des (low bits)
  INC dest                                   ' point to dest + 1
  offset = 7 - offset                        ' fix offset
  destVal = dbCopy >> offset                 ' adjust high bits
  __RAM(dest) = destVal                      ' write high bits
  ENDSUB

' --------------------------------------------------------------------------

' Use: UNSQUEEZE *source, offset, *destination
' -- source(offset) is LSB of 7-bit (packed) network byte
' -- destination will hold 8-bit data bit

SUB UNSQUEEZE
  src = __PARAM1                             ' pointer to src(0)
  offset = __PARAM2                          ' offset into src
  dest = __PARAM3                            ' poiner to dest(0)
```

```
  src = src + offset                          ' point to dByte (LSB)
  dest = dest + offset                        ' point to output
  dByte = __RAM(src)                          ' get packed LSB
  dByte = dByte >> offset                     ' unpack LSB
  INC src                                     ' point to MSB
  dbMSB = __RAM(src)                          ' get it
  offset = 7 - offset
  dbMSB = dbMSB << offset                     ' unpack
  dByte = dByte | dbMSB                       ' reconstitute dByte
  __RAM(dest) = dByte
  ENDSUB


' -------------------------------------------------------------------------
' User Data
' -------------------------------------------------------------------------
```