Column #122 June 2005 by Jon Williams:

Even Mo' MIDI

About two years ago I did a couple columns on MIDI (Musical Instrument Digital Interface) using a BASIC Stamp 2 microcontroller. Our experiments at the time were limited to sending MIDI data. These articles generated a lot of interest, and the most pressing question has been, "How can I receive and process MIDI data in my project?" The fact is that it's very tough to do that effectively with a BASIC Stamp, but now that we're equipped with SX/B, we're ready to rock ... and roll ... and do anything else we chose to do with a MIDI data stream.

In the event you haven't actually heard of it yet, MIDI is a serial communications scheme developed to link instruments like synthesizers together. It's not particularly difficult; data is sent at 31.5 kBaud in small packets. The tough part for BASIC Stamp users is that the packets can vary in length; this creates a real challenge for SERIN. And the reality is that the processing time of bytes already captured would cause us to miss other data in the stream. What we need is a serial buffer to capture everything and allow us to process data on-the-fly.

Since there's a good chance that this project could become a product for the new Parallax EFX group, I wanted to keep the cost and parts count low – this kind of precludes the use of a BASIC Stamp and an external UART to capture the data. What we need is a chip we can program in BASIC that will buffer the MIDI data while we're doing other things. Oh, hey, we already have one: the SX micro – when we program it with the SX/B compiler.

SX/B 1.2

It's no big secret that I work for Parallax, and that I'm part of the SX/B team – still, I'm very proud that the company has provided this product at no charge, and continues to make improvements to it. Honestly, SX/B has opened a whole new world for me personally as I just don't have the patience to program in 100% assembly. A routine here and there, no problem – the whole doggone program, no way.

Version 1.2 of the SX/B compiler adds support for the SX48 and SX52, and – what I think is best – is that it simplifies the use of code pages in the SX. You'll remember in our December and January projects that we created a "jump table" to get to subroutines located on another code page within the SX. With SX/B 1.2, all we have to do is declare our subroutines (with SUB) and the compiler handles the rest. Oh, another time saver is IF-THEN-ELSE. Yes, this makes decision-making a lot easier and saves us from having to insert our own labels to handle the IF-THEN branching. Of course there are other improvements in the compiler, but these are the features that I think programmers will find most useful.

MIDI Controller

My colleague, John Barrowman, and I started a group called Parallax EFX to build products for the props and FX industry (movies, TV, holiday displays, etc.). As we spend more time with folks who build props, we're finding an increasing interest in MIDI control – this is especially true with those folks building Halloween displays. The idea is to use a computer-based sequencer to play audio tracks and send MIDI messages to electronic circuits that control prop actuators and effects devices. This month's project is the electronic control end of things.

Before we get to the details of the circuit and code, let's chat a little about the MIDI messages and the challenges we face. In a simple world, a MIDI device would use fixed length packets – perhaps doing something like this:

```
90 3C 64   ' note on for middle C
80 3C 00   ' note off for middle C
```

Well, we can dream, but MIDI is not so simple in application. What we'll actually see – from most instruments and sequencers – is this:

```
90 3C 64   ' note on for middle C
3C 00      ' note off for middle C
```

Huh? What happened to the status byte for the Note Off command (should have been $80)? The truth is that it's not really used much. What happens is:

```
90 3C 64   ' note on for middle C
90 3C 00   ' note off for middle C
```

The logic here is that turning a note on with a zero velocity (initial volume) is the same as turning it off. So why didn't we see the second $90? Running Status. You see, MIDI data isn't flying around particularly fast, so anything that can be done to reduce the number of bytes in the stream is helpful to system performance. The MIDI protocol employs a strategy called running status to do this. What this means is that the MIDI receiver is expected to keep track of the last valid status byte and use it when a data byte shows up when a new status byte is expected. In the example above, the receiver would know to use the last valid status byte ($90) when a data byte arrives unexpectedly.

Alright, I've been a little loose with terms … how do we know the difference between a status byte and a data byte? Luckily, the folks who created the MIDI standard made it pretty easy: status bytes are $80 and higher; data bytes are $7F and lower. While this helps us determine what's what, it actually creates a bit of work in some circumstances. A pitch wheel change, for example, sends $Ex (x is the MIDI channel) followed by two bytes that represent the 14-bit pitch wheel value – but each byte only has seven bits so the receiver has to reassemble them to get a usable value.

MIDI In – Control Out

Okay, let's jump into the project. Our goal is to "listen" to a MIDI stream and respond to Note On and Note Off messages that match our channel and octave settings. Now, there are more legal notes than what we want to deal with in a small controller, so our design will handle one: control output 1 will be assigned to C in the selected octave; control output 12 will be assigned to B in that same octave.

This choice works very nicely with the number of IO pins available on an SX28. Have a look at Figure 122.1, the schematic for the project. MIDI data is actually transmitted through a current loop, and the specification says that one MIDI output will feed just one MIDI input. The 6N138 opto-isolator converts the incoming current loop to a TTL level serial signal that gets fed to the SX28. It also gets routed, through two inverters, to a MIDI THRU port. This lets us insert our controller in a chain of MIDI-compatible devices.

The serial data coming into RA.0 is 31.25 kBaud, N81, true (idle state is high) mode. A start bit, then, will be when the serial line goes from Vdd to Vss. Back in January we did a project that captured serial data in an ISR (interrupt service routine) and that same code is used here. The only thing we have to change is the Interrupt period for the MIDI baud rate.

At 31.25 kBaud, each MIDI bit is 32 microseconds wide. Sticking with the idea that we should sample the serial line at least four times per bit period, we need to setup the ISR to trip every eight microseconds. This isn't very hard to do with SX/B; we simply put the number of cycles for our interrupt period after the RETURNINT instruction. So what is that number?

We can calculate it like this:

Cycles = Freq / Prescaler * Int_Period

With a clock frequency of 20 MHz, and a prescaler setting of 1:1, we get 160.

20,000,000 / 1 * 0.000008 = 160

What would happen if we decided to bump our clock frequency up to 50 MHz? We'd get:

50,000,000 / 1 * 0.000008 = 400

Houston, we have a problem – the value following the RETURNINT must fit into a byte. What we would have to do is bump the RTCC prescaler to 1:2, and then things work out:

50,000,000 / 2 * 0.000008 = 200

This is about the only tricky aspect of dealing with the ISR serial code; we really have to keep on top of things when it comes to the numbers. If we do, we'll be we rewarded because the serial code happily receives and buffers incoming data while we blissfully run our foreground code.

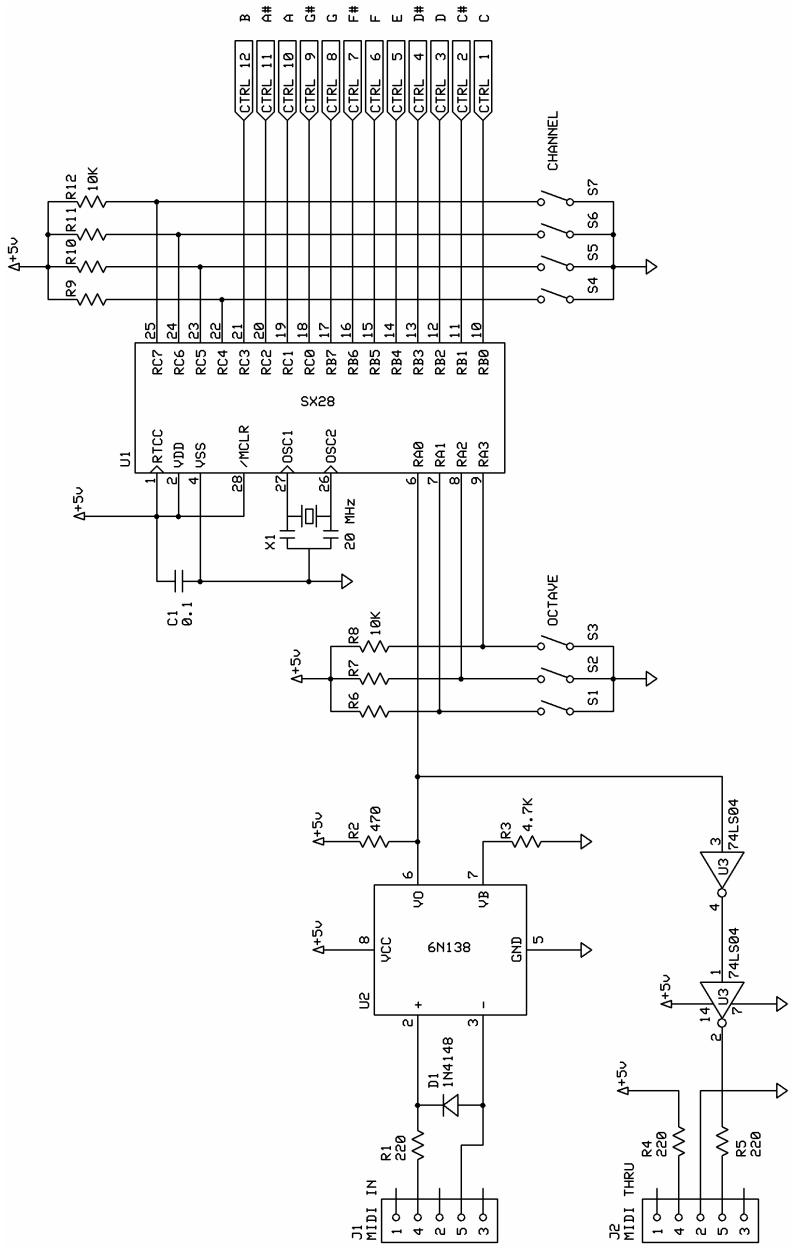

Figure 122.1: MIDI Schematic with SX28AC/DP

```
ISR_Start:
  ASM
    MOVB C, MidiIn
    TEST rxCount
    JNZ RX_Bit
    MOV W, #9
    SC
    MOV rxCount, W
    MOV rxTimer, #6

RX_Bit:
    DJNZ rxTimer, ISR_Exit
    MOV rxTimer, #4
    DEC rxCount
    SZ
    RR rxByte
    SZ
    JMP ISR_Exit

RX_Buffer:
    MOV FSR, #rxBuf
    ADD FSR, rxHead
    MOV IND, rxByte
    INC rxHead
    CLRB rxHead.4
  ENDASM

ISR_Exit:
  BANK $00
  RETURNINT 160
```

Other than the timing change for MIDI, this is in fact the same code we used back in January for the LED multiplexer (see that article for a description of the serial receive and buffer routine). Now that we have MIDI data collecting in a 16-byte circular buffer, we can start pulling it out and comparing to the commands that our device will respond to (specifically Note On and Note Off).

As we move down the listing into the heart of the program we encounter something new: subroutine declarations. This is my favorite aspect of SX/B version 1.2.

```
GETBYTE         SUB     1
GETCHANNEL      SUB
GETPORT         SUB     1
SETCTRL         SUB     2
```

The declaration of subroutines does two things for us: 1) It causes the compiler to create a jump table to the actual code, saving us the trouble of doing that, and 2) It causes the compiler to validate the number of parameters passed to a subroutine – this is a very big help and prevents a lot of program bugs from causing problems.

The number following the SUB declaration tells the compiler how many parameters, if any, are required by the subroutine. Better yet, the compiler can even check for a variable number of parameters. For example:

```
DELAYUS         SUB     1, 2
```

In the declaration for DELAYUS (not used in the MIDI program), we're telling the compiler that we must pass at least one parameter, and that we could pass two. The help file has several examples that show how to use variable parameter declarations.

Finally, when using a declared subroutine, we don't have to use the keyword GOSUB anymore. As you look through the project listing you'll see lines like:

```
GETBYTE @midiStatus
```

This code calls the GETBYTE subroutine and passes the address (@) of the midiStatus variable as a parameter. As you can see, the SUB declaration is a really powerful feature, and for my money (yes, I know SX/B is free…), the best improvement to the SX/B compiler.

Okay, let's get to the actual MIDI decoding. The program starts by setting up the IOs, enabling the ISR, and then drops into the top where we look for a status byte.

```
Main:
  IF hasStatus = 0 THEN
    GETBYTE @midiStatus
    IF midiStatus.7 = 0 THEN Main

Check_Sys:
    IF midiStatus >= $F0 THEN Sys_Cmd
    runStatus = midiStatus
  ENDIF
```

In the beginning we don't have a status byte saved, so the program will in fact call GETBYTE and wait for a status byte to arrive. Since status bytes are $80 and higher, all we have to do is look at bit 7 (aliased as hasStatus) to tell if the byte received is status or not. If not, we try again.

There's a special case when the status byte is $F0 and higher – these bytes are system commands and need to be handled separately. When we get a normal ("voice") status byte ($80 - $EF) we will save that in our running status variable. All status bytes require at least one data byte, so that's what we collect next.

```
Get_DB1:
  midiStatus = runStatus
  GETBYTE @midiDB1
  IF midiDB1.7 = 1 THEN
    midiStatus = midiDB1
    GOTO Check_Sys
  ENDIF
```

You may be wondering why we need to copy the running status byte back to our midiStatus variable. Well, we'll ultimately end up here again, and if the last status byte that came in was a system byte we need to refresh the running status byte for the incoming data. If, for some reason, we get a status byte at this point, it gets moved into the midiStatus variable and we jump to Check_Sys to handle a system byte that might have shown up. If it isn't, the process continues as before with the new status byte.

Let's say that things went well and the status byte received was $90 (Note On, channel 1) and the first data byte was $3C (middle C). The next thing we have to do is look at the status byte and jump to a handler for it.

```
Do_Command:
  temp1 = midiStatus & $F0
  IF temp1 = $80 THEN Note_Off
  IF temp1 = $90 THEN Note_On
  IF temp1 = $A0 THEN Aftertouch
  IF temp1 = $B0 THEN Controller
  IF temp1 = $C0 THEN Pgm_Change
  IF temp1 = $D0 THEN Chan_Pressure
  IF temp1 = $E0 THEN Pitch_Wheel
  GOTO Main
```

The first line of this routine masks out the channel data (low nibble of status byte), then does a comparison of valid commands and jumps to the proper routine. Why jump to a routine if the channel doesn't match? Well, the status and data bytes are in the buffer anyway, and they have to be pulled out (I suppose we could come up with a clever routine to manipulate the buffer head pointer, but that could lead to more complications than its worth, especially since we need to check for system commands).

In our case we're going to jump to Note_On:

```
Note_On:
  GETBYTE @midiDB2
  IF midiDB2.7 = 1 THEN
    midiStatus = midiDB1
    GOTO Check_Sys
  ENDIF
  GETCHANNEL
  temp1 = midiStatus & $0F
  IF temp1 = channel THEN
    GETPORT @midiDB1
    IF midiDB1 < 12 THEN
      IF midiDB2 > 0 THEN                    ' velocity > 0?
        SETCTRL midiDB1, 1                   '   yes, turn port on
      ELSE
        SETCTRL midiDB1, 0                   '   no, turn port off
      ENDIF
    ENDIF
  ENDIF
  GOTO Main
```

Again, we'll call GETBYTE to get the second byte for the Note On command; this byte will hold the "velocity" (initial volume) for the note currently being held in midiDB1. And, again, we'll make sure that we didn't get a status byte when it's not expected. If we do, it's handled by moving the new status byte into midiStatus and jumping back to Check_Sys for appropriate processing. That will usually not be the case, however, what we'll end up with is a value between zero (off) and 127 (max volume).

With the entire packet removed from the buffer we can compare the channel information in the status byte with the channel setting of our controller.

```
GETCHANNEL:
  channel = ~CtrlHi
  SWAP channel
  channel = channel & $0F
  RETURN
```

As you can see, checking the channel setting of the controller is pretty easy: we read the channel switch settings and invert them since we're using active-low inputs. As the channel data is in the upper nibble, we can use SWAP to move it to the lower nibble. This is a new command in SX/B and does exactly the same thing as its assembly namesake. It's also much quicker than using a shift instruction to move the bits from one nibble to the other. Finally,

we mask out the unused bits (high nibble of channel) and return to the program. In case you're wondering why this isn't done at the beginning of the program, the reason is it lets us change the channel setting of the controller on-the-fly. This can be very useful when we're attempting to integrate it into a MIDI system.

Now that we have the channel number from the controller, we can grab the channel data from the status byte and compare them. Let's assume a match. The next thing we have to check for is a match for our range of outputs. Remember, there are 128 possible note values but we've only got 12 outputs. What we've done is divided the possible outputs into octaves – just like on a piano keyboard. The GETPORT subroutine does two things: it checks to see if the note value in the packet matches our setting, and it converts that value to a zero to 11, or to $FF (not valid for us) for later use.

```
GETPORT:
  temp1 = __PARAM1
  temp2 = __RAM(temp1)

  baseNote = ~Octave
  baseNote = baseNote >> 1
  baseNote = baseNote & 7
  baseNote = baseNote + Transpose
  baseNote = baseNote * 12
  IF temp2 < baseNote THEN
    temp2 = $FF
  ELSE
    temp2 = temp2 - baseNote
    IF temp2 > 11 THEN
      temp2 = $FF
    ENDIF
  ENDIF
  __RAM(temp1) = temp2
  RETURN
```

This subroutine expects an address to be passed to it, so the second line takes care of reading the value from that address. After that, the Octave switches are read and the base note value for the controller is calculated. One thing that may require a little extra explanation is the Transpose constant. I have two keyboards, and the lowest C on both of them was actually in octave three. By setting Transpose to 3, I was able to make the lowest key on my keyboards correspond to octave zero on the controller.

Now we can compare the note sent with our own range. When it's in range, a simple subtraction will reduce the value to between zero and 11 – this corresponds to our output

ports. If the note value is not in range, we'll reset it to $FF; this serves as a flag to the program that the note is of no use to us.

Let's continue assuming that the controller was set such that pressing Middle C on a keyboard would cause the note value to be found valid, hence set to zero. The last thing we need to do is check the velocity (initial volume) value. Remember, most MIDI devices use the Note On command with a velocity of zero to turn a note off. After a simple comparison we can call the SETCTRL subroutine with the second parameter as 1 for on, 0 for off.

```
SETCTRL:
  temp1 = __PARAM1
  temp2 = __PARAM2

  IF temp1 < 8 THEN
    IF temp2.0 = 1 THEN
      temp2 = 1 << temp1
      CtrlLo = CtrlLo | temp2
    ELSE
      temp2 = 1 << temp1
      temp2 = ~temp2
      CtrlLo = CtrlLo & temp2
    ENDIF
  ELSE
    IF temp2.0 = 1 THEN
      temp1.3 = 0
      temp2 = 1 << temp1
      CtrlHi = CtrlHi | temp2
    ELSE
      temp1.3 = 0
      temp2 = 1 << temp1
      temp2 = ~temp2
      CtrlHi = CtrlHi & temp2
    ENDIF
  ENDIF
  RETURN
```

One of the nice things the BASIC Stamp does for us is hide details about controlling IO pins. The fact of the matter is that the IO ports on the PIC and SX micros used to make BASIC Stamp modules are only eight bits wide, but the design of PBASIC lets us treat the two ports as one 16-bit entity.

That's what we're doing with SETCTRL – we're treating two ports (RB and RC) as one big group. The first thing we have to do is figure out which of the two ports is going to be

affected, RB or RC. If the control port is less than eight, it's RB, otherwise it's RC. Then we check to see if the port is going to be turned on or off.

Let's stick with our Note On. In that case we will create a bit mask for the proper pin using the shift left operator, then add (with OR) that mask to the current state of the outputs. Turning a port off requires one additional step: we have to invert the mask (putting a zero into the affected control port bit) then use AND to clear it while maintaining the current state of the other pins.

Whew … that was a bit of work but what we have at the moment is an output on. If we connect to an LED we'll see it lit. Now, let's release the key. Again, what we'll probably get is only two bytes:

3C 00

At Main we see that the byte is not a new status byte (bit 7 is clear) so we reload the running status (currently $90), jump to Get_DB1 where we grab the $3C note value (in midiDB1), then ultimately back to Note_On where we get $00 into midiDB2. Since the velocity value is zero, we'll call SETCTRL with 0 as the second parameter and the control port will be turned off.

Okay, time to take a breath.

This is one of those programs where the explanation is far more complicated than the process. That said, the process is not to be taken for granted and even after having code working for over a week now, I find myself making small improvements to it.

If you find yourself overwhelmed, but are interested in building a MIDI-compatible controller, don't fret – go do a Google search on MIDI and you'll find all kinds of useful information on the protocol and lots of projects people have done with small microcontrollers. After reading some of the protocol explanations, come back to the program listing and have a look. After a couple reads it will start to make sense; it did for me, anyway. When I went into this project I expected it to be a little less involved – that was a silly assumption on my part. All's well now, though, and we have a base for all kinds of MIDI projects.

Have fun, do neat things with your MIDI controller and, as always, Happy Stamping! – SX/B style….

```
' =============================================================================
'
'   File...... MIDI_Control.SXB
'   Purpose... MIDI Digital Controller
'   Author.... Jon Williams, Parallax EFX
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 18 APR 2005
'
' =============================================================================


' -----------------------------------------------------------------------------
' Program Description
' -----------------------------------------------------------------------------'
'
' "Listens" to a MIDI stream and will control outputs if Note value
' received is within the range of the device settings.
'
' Commands:
' -- Note Off ($80)
' -- Note On  ($90)
' -- Controller message ($B0 $7B $00) = All off
' -- System reset ($FF) = All off


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

DEVICE          SX28, OSCXT2, TURBO, STACKX, OPTIONX
FREQ            20_000_000


' -----------------------------------------------------------------------------
' IO Pins
' -----------------------------------------------------------------------------

MidiIn          VAR     RA.0                ' from 6N138 interface
Octave          VAR     RA                  ' RA.1 - RA.3
CtrlLo          VAR     RB                  ' outs 0 - 7
CtrlHi          VAR     RC                  ' outs 8 - 11, chan select


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

Transpose       CON     3                   ' octave offset
                                            ' -- for user keyboard
```

```
' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------

rxCount        VAR     Byte                    ' bits to receive
rxTimer        VAR     Byte                    ' bit timer for ISR
rxByte         VAR     Byte                    ' serial byte
rxHead         VAR     Byte                    ' available slot
rxTail         VAR     Byte                    ' next byte to read
rxBuf          VAR     Byte(16)                ' circular buffer

midiStatus     VAR     Byte                    ' midi packet
midiDB1        VAR     Byte
midiDB2        VAR     Byte
runStatus      VAR     Byte                    ' running status byte
hasStatus      VAR     runStatus.7             ' we have a status byte
channel        VAR     Byte                    ' channel assignment
baseNote       VAR     Byte                    ' base note (C) of port 0

regAddr        VAR      Byte                   ' register address
temp1          VAR      Byte                   ' parameters
temp2          VAR     Byte


' -----------------------------------------------------------------------------
  INTERRUPT NOPRESERVE
' -----------------------------------------------------------------------------

' ISR is setup to receive N81, true mode.
'
' Notes:
' -- MIDI baud is 31.25 kB, 32 uS per bit
' -- Interrupt setup for 4x bit period for adequate sampling

ISR_Start:
  ASM
    MOVB C, MidiIn                          ' sample serial input
    TEST rxCount                            ' receiving now?
    JNZ RX_Bit                              ' yes if rxCount > 0
    MOV W, #9                               ' start + 8 bits
    SC                                      ' skip if no start bit
    MOV rxCount, W                          ' got start, load bit count
    MOV rxTimer, #6                         ' delay 1.5 bits

RX_Bit:
    DJNZ rxTimer, ISR_Exit                  ' update bit timer
    MOV rxTimer, #4                         ' reload bit timer
    DEC rxCount                             ' mark bit done
    SZ                                      ' if last bit, we're done
    RR rxByte                               ' move bit into rxByte
    SZ                                      ' if not 0, get more bits
```

```
    JMP ISR_Exit

RX_Buffer:
    MOV FSR, #rxBuf                         ' get buffer address
    ADD FSR, rxHead                         ' point to head
    MOV IND, rxByte                         ' move rxByte to head
    INC rxHead                              ' update head
    CLRB rxHead.4                           ' keep 0 - 15
  ENDASM

ISR_Exit:
  BANK $00
  RETURNINT 160                             ' 8 uS @ 20 MHz


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


' ----------------------------------------------------------------------------
' Subroutine Declarations
' ----------------------------------------------------------------------------

GETBYTE        SUB     1                    ' pass addr of byte
GETCHANNEL     SUB                          ' read channel from sw
GETPORT        SUB     1                    ' get output port
SETCTRL        SUB     2                    ' set/clear output port


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

Start:
  TRIS_A = %1111                            ' Serial + Base select
  TRIS_B = %00000000                        ' lo outs
  TRIS_C = %11110000                        ' hi outs, channel in
  OPTION = $88                              ' Interrupt on, 1:1

Main:
  IF hasStatus = 0 THEN                     ' running status?
    GETBYTE @midiStatus                     '   no, get status
    IF midiStatus.7 = 0 THEN Main           ' skip orphan data

Check_Sys:
    IF midiStatus >= $F0 THEN Sys_Cmd       ' handle system command
    runStatus = midiStatus                  ' save regular status
  ENDIF

Get_DB1:
```

```
  midiStatus = runStatus                    ' refresh status byte
  GETBYTE @midiDB1                          ' get first data byte
  IF midiDB1.7 = 1 THEN                     ' new status byte?
    midiStatus = midiDB1                    '   yes, reposition
    GOTO Check_Sys
  ENDIF

Do_Command:
  temp1 = midiStatus & $F0                  ' isolate command
  IF temp1 = $80 THEN Note_Off              ' jump to cmd handler
  IF temp1 = $90 THEN Note_On
  IF temp1 = $A0 THEN Aftertouch
  IF temp1 = $B0 THEN Controller
  IF temp1 = $C0 THEN Pgm_Change
  IF temp1 = $D0 THEN Chan_Pressure
  IF temp1 = $E0 THEN Pitch_Wheel
  GOTO Main


' *** MIDI Command Processing ***

Note_Off:
  GETBYTE @midiDB2                          ' get velocity
  IF midiDB2.7 = 1 THEN                     ' check for status byte
    midiStatus = midiDB1
    GOTO Check_Sys
  ENDIF
  GETCHANNEL
  temp1 = midiStatus & $0F                  ' isolate channel
  IF temp1 = channel THEN
    GETPORT @midiDB1                        ' check note for range
    IF midiDB1 < 12 THEN                    ' if in range...
      SETCTRL midiDB1, 0                    '   yes, turn port off
    ENDIF
  ENDIF
  GOTO Main


Note_On:
  GETBYTE @midiDB2                          ' get velocity
  IF midiDB2.7 = 1 THEN                     ' check for status byte
    midiStatus = midiDB1
    GOTO Check_Sys
  ENDIF
  GETCHANNEL
  temp1 = midiStatus & $0F                  ' isolate channel
  IF temp1 = channel THEN
    GETPORT @midiDB1                        ' check note for range
    IF midiDB1 < 12 THEN                    ' if in range...
      IF midiDB2 > 0 THEN                   ' velocity > 0?
        SETCTRL midiDB1, 1                  '   yes, turn port on
```

```
      ELSE
        SETCTRL midiDB1, 0                  '   no, turn port off
      ENDIF
    ENDIF
  ENDIF
  GOTO Main


Aftertouch:
  GETBYTE @midiDB2                          ' get pressure byte
  IF midiDB2.7 = 1 THEN                     ' check for status byte
    midiStatus = midiDB1
    GOTO Check_Sys
  ENDIF

  ' aftertouch processing here

  GOTO Main


Controller:
  GETBYTE @midiDB2                          ' get second data
  IF midiDB2.7 = 1 THEN                     ' check for status byte
    midiStatus = midiDB1
    GOTO Check_Sys
  ENDIF

  IF midiDB1 = $7B THEN                     ' all off?
    CtrlLo = %00000000
    CtrlHi = %0000
    runStatus = $00
  ENDIF

  GOTO Main


Pgm_Change:
  ' patch already in midiDB2
  GOTO Main


Chan_Pressure:
  ' pressure already in midiDB2
  GOTO Main


Pitch_Wheel:
  GETBYTE @midiDB2                          ' get MSB of pitch
  IF midiDB2.7 = 1 THEN                     ' check for status byte
    midiStatus = midiDB1
    GOTO Check_Sys
```

```
  ENDIF

  ' convert pitch bytes to 16-bit var
  '
  midiDB1.7 = midiDB2.0
  midiDB2 = midiDB2 >> 1

  GOTO Main


Sys_Cmd:
  IF midiStatus = $F0 THEN                    ' handle SysEx bytes
    GETBYTE @midiDB1
    IF midiDB1 = $F7 THEN Main                ' wait for end ($F7)
    runStatus = $00
    GOTO Sys_Cmd
  ENDIF

  IF midiStatus = $FF THEN                    ' reset?
    CtrlLo = %00000000
    CtrlHi = %0000
    runStatus = $00
  ENDIF
  GOTO Main


' -----------------------------------------------------------------------------
' Subroutine Code
' -----------------------------------------------------------------------------

' Use: GETBYTE @aVar
' -- if data is in buffer, the next byte is move to 'aVar'

GETBYTE:
  regAddr = __PARAM1                          ' save return address

Buf_Wait:
  IF rxHead = rxTail THEN Buf_Wait            ' anything in buffer?

  temp1 = rxBuf(rxTail)                       ' get byte at tail
  INC rxTail                                  ' update tail position
  rxTail.4 = 0                                ' keep 0 - 15
  __RAM(regAddr) = temp1                      ' move byte to target
  RETURN


' Use: GETCHANNEL
' -- reads channel inputs and refreshes 'channel'

GETCHANNEL:
  channel = ~CtrlHi                           ' get channel number
```

```
  SWAP channel
  channel = channel & $0F
  RETURN


' Use: GETPORT @note
' -- validates note in range of device setting
' -- returns 0 - 11 if in range
' -- returns $FF if out of range

GETPORT:
  temp1 = __PARAM1                          ' save address
  temp2 = __RAM(temp1)                      ' get current value

  baseNote = ~Octave                        ' get base octave value
  baseNote = baseNote >> 1
  baseNote = baseNote & 7                   ' mask out unused bits
  baseNote = baseNote + Transpose           ' adjust for instrument
  baseNote = baseNote * 12                  ' calculate base (C) note
  IF temp2 < baseNote THEN                  ' below range?
    temp2 = $FF                             '   yes - mark OOR
  ELSE
    temp2 = temp2 - baseNote                ' adjust for upper end
    IF temp2 > 11 THEN                      ' above range?

      temp2 = $FF                           '   yes - mark OOR
    ENDIF
  ENDIF
  __RAM(temp1) = temp2                      ' move port or OOR to RAM
  RETURN


' Use: SETCTRL ctrlPort, portVal
' -- sets 'ctrlPort' (0 - 11) to 'portVal.0'

SETCTRL:
  temp1 = __PARAM1                          ' save port
  temp2 = __PARAM2                          ' save status

  IF temp1 < 8 THEN                         ' lower port?
    IF temp2.0 = 1 THEN
      temp2 = 1 << temp1                    ' create bit mask
      CtrlLo = CtrlLo | temp2               ' activate port
    ELSE
      temp2 = 1 << temp1                    ' create bit mask
      temp2 = ~temp2                        ' invert mask
      CtrlLo = CtrlLo & temp2               ' clear port
    ENDIF
  ELSE
    IF temp2.0 = 1 THEN
      temp1.3 = 0                           ' subtract 8 from port
```

```
    temp2 = 1 << temp1                    ' create bit mask
    CtrlHi = CtrlHi | temp2               ' activate port
  ELSE
    temp1.3 = 0                           ' subtract 8 from port
    temp2 = 1 << temp1                    ' create bit mask
    temp2 = ~temp2                        ' invert mask
    CtrlHi = CtrlHi & temp2               ' clear port
  ENDIF
ENDIF
RETURN
```