Column #95 March 2003 by Jon Williams:

Mo' MIDI

When it comes right down to it, I'm a simple, fun-loving, goofy guy. I like movies, good coffee, pretty ladies (who doesn't?) and fun BASIC Stamp projects. I seem to get the most enjoyment out of the simplest of my projects. This month fits into that category: easy projects with simple hardware and code and I have had more fun playing with them than most projects I've created with BASIC Stamps in the last eight years.

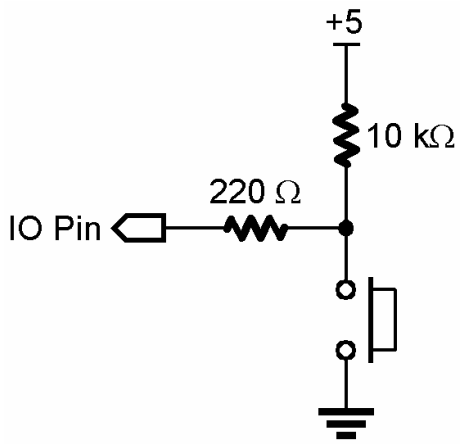
Last month we saw that it's fairly easy to send MIDI commands and play a stored song. What we did was easy, but not particularly interactive. This time we'll use the BASIC Stamp as a sensor interface and send note-on and -off commands based on the sensors. Now, I know what you're thinking: "Why do that when I have a perfectly good keyboard sitting right in front of me?" Well, you might want to do something more fun. Remember the on-the-floor keyboard that Tom Hanks and Robert Logia played in the movie "Big"? You could do that. Or create a neat, interactive sculpture as a museum piece. Kids especially like those kinds of things. And then, this forty-year-old kid likes them too!

A Custom Keyboard

A common topic on the BASIC Stamps [Yahoo! Groups] list is dealing with multiple simultaneous inputs and their change of states. Without fail, my good buddy Tracy Allen comes to the rescue with great information on finite state machines (www.emesystems.com/BS2fsm.htm) and how to take advantage of this programming strategy with BASIC Stamps.

We're going to put those techniques to use here and create a custom keyboard. For testing, we can use pushbuttons as in Figure 95.1 – use one circuit for each key. In our program, we'll keep things simple by using eight keys so that we can read them all at once by grabbing the status of InL (pins 0 through 7), which our program aliases as Keys.

Figure 95.1: MIDI Pushbutton Test Circuit



Let's look at the scanning subroutine then discuss how it works.

```
Get Keys:
  scan = Keys
  changes = scan ^ last
  last = scan
  RETURN
```

This is actually quite simple, but what trips up many beginning BASIC Stamp programmers is the use of the Exclusive OR operator (^). The rule for XOR is "If one bit OR the other is set, but NOT both, the output will be True." Here's what it looks like as a truth-table:

A	B	XOR
0	0	0
1	0	1
0	1	1
1	1	0

Another way to remember the behavior of XOR is that if the bits are different, the output will be True (1); if they're the same, the output will be False (0).

We can take advantage of this behavior by comparing the last key scan with the current scan. Any changes between them will appear as 1 in the changes variable. Let's say, for example, we start the program then press the keys connected to pins 7 and 5. The next time through the Get_Keys subroutine we would see these values after the line that evaluates changes:

scan	%01011111
last	%11111111
changes	%10100000

Of course, after the changes have been evaluated, we store the current scan in the variable called last so that it's updated for the next time we call Get_Keys.

Now that we can tell what key (or keys) has changed, if any, sending MIDI commands is as easy as looping through the changes and acting accordingly.

```
Play Notes:
  IF changes THEN
    FOR idx = 7 TO 0
      IF (changes.LowBit(idx)) THEN
        cmd = NoteOn - ($10 * scan.LowBit(idx))
        note = MiddleC + (7 - idx)
        SEROUT MidiOut, MidiBaud, [cmd, note, velocity]
      ENDIF
    NEXT
  ENDIF
  RETURN
```

The first thing we'll check for is changes. We just saw that bits will only be set in changes if a key was pressed or released between scans. If there have been no changes, all bits will be zero and we can skip right past the rest of the subroutine code.

But let's say a key was pressed or released. We'll use a FOR-NEXT loop to scan the bits in changes to see what happened. When we find a 1 bit in changes, the command is calculated based on the corresponding bit value in scan. We can do this mathematically with just one line of code.

```
  cmd = NoteOn - ($10 * scan.LowBit(idx))
```

The variable idx holds the bit (key position) we're evaluating. If the value of that bit in scan is zero, the key was pressed and the line above evaluates as:

```
  cmd = $90 - ($10 * 0) = $90
```

If the key had been released, we'd get:

```
  cmd = $90 - ($10 * 1) = $80
```

The note value is also calculated. Our keyboard works like a piano keyboard, with notes getting higher as we move left to right. The base note, Middle C, is assigned to bit 7, so our eight-bit keyboard will play these notes:

C-C#-D-Eb-E-F-F#-G

If we wanted to eliminate the sharps and flats, we could insert a LOOKUP table to assign the note value:

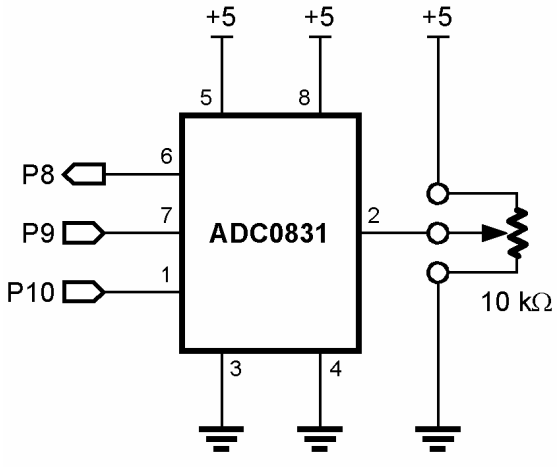
```
  LOOKUP (idx - 7), [60, 62, 64, 65, 67, 69, 71, 72], note
```

The reason we subtract seven from idx is to align the table values with the keyboard; bit 7 corresponding to the first note value in the table.

Finally, we can add some life to our keyboard by allowing a volume (velocity) change for the notes we play. One way of doing this is to read the position of a potentiometer with an analog-to-digital converter. Figure 95.2 shows a simple set-up with ADC0831 – a part we've used many times in the past. Reading the position of the potentiometer is a no-brainer:

```
Get Velocity:
  LOW A2Dcs
  SHIFTIN A2Ddata, A2Dclock, MSBPost, [velocity\9]
  HIGH A2Dcs
  velocity = velocity / 2
  RETURN
```

Figure 95.2: ADC0831 Circuit for MIDI Volume Change



Since the ADC0831 returns a value of 0 to 255, we can scale it to a legal MIDI value (0 to 127) by dividing by two. Easy. With everything in place, SEROUT sends the MIDI command to our instrument.

You may wonder why we don't just use RCTIME to read the potentiometer. The reason is that RCTIME is a time-based function, and can get quite long at large pot values. By using the ADC0831, the time to read the position of the pot is the same, regardless of its position. This strategy also lets us use other control [voltage] sources for the velocity level.

The advantage of keeping everything modular is that we can make sectional changes without upsetting the overall program design. It keeps the main loop organized and, with appropriate subroutine names, self documenting of its behavior. Take a look:

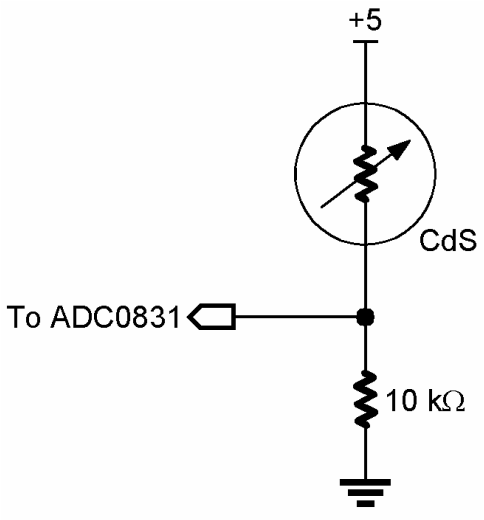
```
Main:
  DO
    GOSUB Get_Keys
    GOSUB Get_Velocity
    GOSUB Play Notes
  LOOP
  END
```

Very clean, isn't it? Technically, the END command is not needed in this program but it's my programming habit to put it in between the main program loop and subroutines section. That way, if I add an EXIT command to this loop, I won't run into any unusual program behavior by having the code crash into the first subroutine on a loop exit.

Music By Light

We just saw how we could use the ADC0831 to set our volume by reading a pot. Let's create a new type of instrument by letting the light falling on a CdS photocell control our note. Replace the pot with the circuit shown in Figure 95.3 for light control.

Figure 95.3: Replace the Pot with this Circuit for Music By Light



What we're going to do with this program, is use the light to control note value. While simple in concept, there is a bit of trickiness here, because we don't know how much light is going to be falling on the photocell when we start. What we need to do then, is scale the program on start-up to the ambient light.

When the program starts, we'll read the ambient light with our A2D subroutine:

```
Get Note:
  LOW A2Dcs
  SHIFTIN A2Ddata, A2Dclock, MSBPost, [note\9]
  HIGH A2Dcs
  note = note */ scale MIN ScaleMin
  RETURN
```

The code should look familiar, we just used most of it to read our volume pot. The difference now is that we're using it to read our note. After reading the raw note value, we want to scale it for the ambient light and for our instrument. Here's where things a little tricky.

We're actually going to call this routine from our initialization section after setting the value of scale to $0100 – or 1.00 for use with the */ (star-slash) operator. What we get, then, on this first read is the raw value of the ambient light. Let's go ahead and look at the initialization section.

```
Setup:
  HIGH A2Dcs
  scale = $0100
  GOSUB Get_Note
  scale = ScaleMax * $0100 / note
  GOSUB Get Note
  last = note
  velocity = 96
```

What I want to focus on is this line:

```
scale = ScaleMax * $0100 / note
```

Like many, I monitored the post-holiday sales and found a MIDI keyboard at Radio Shack for less than a hundred dollars. It's not a full-sized keyboard, though, and the highest note value it can play is 96. That's the value that ScaleMax is set to.

By shifting the ScaleMax value into the upper byte of scale, then dividing by the ambient light reading, we end up with a [fractional] value in scale that sets the ambient light level to the highest note on the keyboard. Let's step through the process.

If the raw value of note is 153, we get:

```
scale = 96 * 256 / 153 = 24576 / 153 = 160
```

Remember that we're using integer math in the BASIC Stamp, so may get a slight rounding error. Also keep in mind that the value of scale is being used with */, so it actually represents units of 1/256. In our example, the equivalent fractional value is 160 / 256 = 0.625.

With the scale value set, we read the sensor again (which should give us the same raw value), but this time the scaling returns a note value of 95 – pretty close to our scale maximum.

153 */ 160 □ 153 * 0.625 = 95

I'm not worried about the scale value not reaching the end of the keyboard because I actually want the keyboard not to play when the ambient light is falling on it. Here's the rest of our simple light controlled MIDI instrument program:

```
Main:
  GOSUB Get_Note
  note = (note */ NMix) + (last */ LMix)
  IF (note = last) THEN Main
  IF (note > HighNote) THEN Last Off

New On:
  SEROUT MidiOut, MidiBaud, [NoteOn, note, velocity]
  PAUSE 5

Last Off:
  SEROUT MidiOut, MidiBaud, [NoteOff, last, 0]
  last = note
  GOTO Main
  END
```

For a moment, let's skip past the line that follows the call to Get_Note and see how the rest of the program works. The first thing we're going to do is look at the current note and see if it's different from the last. If not, we'll go back and look again, waiting until we get a note change.

Once the note changes, we'll make sure it's in range. If not, we'll silence the instrument by jumping to Last_Off which will kill the last note we played. If the new note is in range, we'll play it and then wait just a bit before silencing the last one. This will mix the notes a bit and make the transition between them a little smoother (depending, of course, on the voice we have selected on the MIDI instrument).

Okay, let's go back and look at that line I skipped.

note = (note */ NMix) + (last */ LMix)

What this line does is apply a very simple digital filter to our input so that the note value doesn't jitter around so much – which can be incredibly annoying when this program is actually connected to a MIDI keyboard. What is does is slows the changes between notes.

In the constants definition of the program, you'll find these lines:

```
MixPercent      CON     35
NMix            CON     $100 * MixPercent / 100
LMix            CON     $100 - NMix
```

This should look somewhat familiar, since we just discussed a technique for finding the appropriate value for scale when using the */ operator. What this code does is sets the value of NMix (new mix) to 35% and LMix (last mix) to 65% (100% - 35%). Our program, then, reads a new note, then mixes 35% of the new value with 65% of the last note value. This slows and smoothes the transition between notes. If you'd like a faster transition, you can increase the MixPercent constant value.

This is a fun program to play with, but you really need to be careful with voice selection on your MIDI keyboard. Some voices sound great; some are just horrible. You can, of course, expand on the program by adding a second ADC0831 for volume control as with our keyboard program. Or, you could use a couple of sonar or IR range finders as control inputs. The possibilities are wide open.

Before I leave this program, I'll answer the question that a few of you are asking yourselves: Why did I use PBASIC 2.0 programming style when all the cool new features of PBASIC 2.5 are available to me? Because the 2.0 syntax made the program easier to read and was a direct reflection of my program flow-chart (yes, I still do that – and you should too). Don't believe me? Well, here's what that program looks like using PBASIC 2.5 syntax:

```
Main:
  DO
    DO
      GOSUB Get Note
      note = (note */ NMix) + (last */ LMix)
    LOOP UNTIL (note <> last)
    IF (note <= HighNote) THEN
      SEROUT MidiOut, MidiBaud, [NoteOn, note, velocity]
      PAUSE 5
    ENDIF
    SEROUT MidiOut, MidiBaud, [NoteOff, last, 0]
    last = note
  LOOP
  END
```

This works exactly like the code presented above but, in my opinion, is not nearly as easy to follow. My point is this: Just because we CAN do something, it doesn't mean that we SHOULD. A good programmer will always write clear, concise code, and the clearest code is not necessarily the fanciest.

Conditional Compilation

One of the neat new features of the new BASIC Stamp compiler is called "conditional compilation." What this allows us to do is to selectively compile portions of the program based on internal or user-created symbols.

Before we get to the details, let me show you something that's at the top of this month's MIDI programs:

```
#SELECT $STAMP
  #CASE BS2, BS2e, BS2pe
    MidiBaud    CON     $8000 + 12

  #CASE BS2sx, BS2p
    MidiBaud    CON     $8000 + 60
#ENDSELECT
```

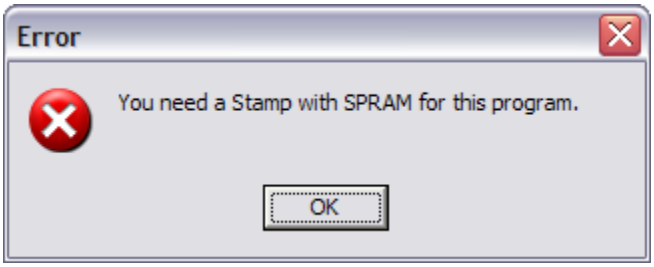
As you know, $STAMP is an internal symbol that tells the compiler what version BASIC Stamp we intend to compile for. The conditional #SELECT-#CASE structure lets us analyze that symbol and set the MidiBaud constant accordingly. This is really convenient for programs that you may be sharing with others and you're not sure which BASIC Stamp they'll be using.

As you might expect, there is also an #IF-#THEN-#ELSE structure as well. Let's look at another quick example:

```
#IF ($STAMP = BS2) #THEN
  #ERROR "You need a Stamp with SPRAM for this program."
#ENDIF
```

This should be pretty easy to understand: If the selected [or connected] BASIC Stamp is a BS2 and, therefore, doesn't have scratch pad RAM, the program won't run (like internal errors, user-defined errors halt the compiler and are flagged). The #ERROR directive works like DEBUG but creates a standard Windows error dialog as shown in Figure 4. Pretty cool, huh?

Figure 95.4: Error Message for Stamps with no SPRAM



It's important to understand that the compiler scans your program for conditional structures prior to the actual compilation process, so variables and program constants cannot be used in the conditional compilation expressions. For flexibility, the editor allows us to create custom symbols. The syntax is:

#DEFINE Symbol { = value }

If the optional value is not supplied, the compiler assigns the value of -1 (65535) which evaluates as True in expressions. If, during the evaluation of an expression, the compiler finds an undefined symbol, it will treat that symbol as defined with a value of zero (False).

Here's how we might use this new tool. Often, as we're developing a program, we'll insert DEBUG statements to track the progress of program variables. When everything is working, we can take them out since they just consume EEPROM space and slow the program a bit. The problem is when we make a substantial update to the program, we end up putting those DEBUG statements back in. There's another way. Let's start by defining a custom symbol:

```
#DEFINE DebugMode = 1
```

Then, at an appropriate place in our program we can create this block:

```
#IF DebugMode #THEN
  ' put DEBUG statements here
#ENDIF
```

We can turn the DEBUG statements on and off by changing the defined value of DebugMode (1 for on, 0 for off). Another approach for removing a symbol is to comment-out the #DEFINE line. And don't forget that #IF-#THEN includes an #ELSE block for additional flexibility.

Space Saving

There's another new feature in PBASIC 2.5 that we can use to save EEPROM space. As you might know by now, long program lines that have comma-delimited lists can be split across multiple lines at the comma breaks. For example, this code:

```
  DEBUG "The BASIC Stamp is an amazing microcontroller.", CR
  DEBUG "I can't imagine my life without it!"
```

Can be changed to:

```
  DEBUG "The BASIC Stamp is an amazing microcontroller.", CR,
        "I can't imagine my life without it!"
```

And we end up reducing the size of our compiled program by a few bytes because there is only one call to DEBUG. This works for SEROUT too because, in fact, DEBUG is SEROUT with the compiler setting the pin to 16 and the baud rate to 9600. Over the course of a program, especially with a lot of serial output, we can save enough space with this technique to make a difference.

If you have lots of long strings in your program, another space-saving strategy is to store them in DATA statements and call them when needed. I prefer to use zero-terminated strings so I can embed carriage returns in them. Like this:

```
Msg1    DATA    "The BASIC Stamp is an amazing microcontroller.", CR,
                "I can't imagine my life without it!", 0
Msg2    DATA    "My other car is a BASIC Stamp.", 0
```

Then we can send the string to any serial output with a simple subroutine. All we have to is point to the string by setting the value of eeAddr before calling this code:

```
Print String:
  DO
    READ eeAddr, char
    IF (char = 0) THEN EXIT
    DEBUG char
    eeAddr = eeAddr + 1
  LOOP
  RETURN
```

Storing strings in DATA statements makes them easier to find when making changes or translations too.

Okay, that's enough for this month. Happy Saint Patty's Day for those of you who share my Irish blood [on my mother's side] – and also to those of you who simply enjoy another reason to have a celebration. Sláinte (To your health!) and Happy Stamping!

```
' =============================================================================
'
'   File...... Light Music.BS2
'   Purpose... Simple MIDI Theremin
'   Author.... Jon Williams
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 22 JAN 2003
'
'   {$STAMP BS2p}
'   {$PBASIC 2.5}
'
' =============================================================================


' -----[ Program Description ]---------------------------------------------
'
' Play a MIDI instrument via light falling on a CdS photocell.  It forms
' part of a voltage divider with a 10K resistor which is read by and
' ADC0831.


' -----[ Revision History ]------------------------------------------------


' -----[ I/O Definitions ]-------------------------------------------------

MidiOut         PIN     15                      ' midi serial output

A2Ddata         PIN      8                      ' ADC0831 data line
A2Dclock        PIN      9                      ' ADC0831 clock
A2Dcs           PIN     10                      ' ADC0831 chip select


' -----[ Constants ]-------------------------------------------------------

#DEFINE DebugMode = 0                           ' for conditional DEBUGs

#SELECT $STAMP                                  ' check Stamp type
  #CASE BS2, BS2E, BS2PE
    MidiBaud    CON     $8000 + 12              ' 31.25 kBaud -- open

  #CASE BS2SX, BS2P
    MidiBaud    CON     $8000 + 60
#ENDSELECT

Channel         CON     0
NoteOn          CON     $90 | Channel
NoteOff         CON     $80 | Channel
```

```
ScaleMin        CON     32                      ' scale low value
ScaleMax        CON     96                      ' sclae high value
HighNote        CON     90                      ' highest playable note

MixPercent      CON     35
NMix            CON     $100 * MixPercent / 100 ' new note mix
LMix            CON     $100 - NMix             ' last note mix


' -----[ Variables ]------------------------------------------------------

note            VAR     Byte                    ' note to play
velocity        VAR     Byte                    ' volume level
last            VAR     Byte                    ' last note
scale           VAR     Word                    ' note scaling


' -----[ EEPROM Data ]----------------------------------------------------



' -----[ Initialization ]-------------------------------------------------

Setup:
  HIGH A2Dcs                                    ' deselect ADC0831
  scale = $0100                                 ' set scale to 1.00
  GOSUB Get Note                                ' read ambient light

  #IF DebugMode #THEN DEBUG DEC ?note : #ENDIF

  scale = ScaleMax * $0100 / note               ' high note = ambient

  #IF DebugMode #THEN DEBUG DEC ?scale : #ENDIF

  GOSUB Get Note

  #IF DebugMode #THEN DEBUG DEC ?note : #ENDIF

  last = note
  velocity = 96                                 ' fixed volume


' -----[ Program Code ]---------------------------------------------------

Main:
  GOSUB Get Note
  note = (note */ NMix) + (last */ LMix)        ' digital filter
  IF (note = last) THEN Main                    ' no change, look again
  IF (note > HighNote) THEN Last_Off            ' if out of range, silence

New_On:
```

```
  SEROUT MidiOut, MidiBaud, [NoteOn, note, velocity]
  PAUSE 5

Last Off:
  SEROUT MidiOut, MidiBaud, [NoteOff, last, 0]
  last = note
  GOTO Main

  END

' -----[ Subroutines ]------------------------------------------------------

Get_Note:
  LOW A2Dcs                                     ' select ADC0831
  SHIFTIN A2Ddata, A2Dclock, MSBPOST, [note\9]
  HIGH A2Dcs                                    ' deslect ADC0831
  note = note */ scale MIN ScaleMin             ' get new note
  RETURN
```

```
' =============================================================================
'
'   File...... Midi Keyboard.BS2
'   Purpose... Detect key changes and play through MIDI instrument
'   Author.... Jon Williams
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 20 JAN 2003
'
'   {$STAMP BS2p}
'   {$PBASIC 2.5}
'
' =============================================================================


' -----[ Program Description ]---------------------------------------------
'
' Creates a simple digital keyboard with the BASIC Stamp that sends note
' and volume [velocity] information to a MIDI instrument.


' -----[ Revision History ]------------------------------------------------


' -----[ I/O Definitions ]-------------------------------------------------

MidiOut         PIN     15                      ' midi serial output
Keys            VAR     INL                     ' custom keyboard inputs

A2Ddata         PIN      8                      ' ADC0831 data line
A2Dclock        PIN      9                      ' ADC0831 clock
A2Dcs           PIN     10                      ' ADC0831 chip select


' -----[ Constants ]-------------------------------------------------------

#SELECT $STAMP                                  ' check Stamp type
  #CASE BS2, BS2E, BS2PE
    MidiBaud    CON     $8000 + 12              ' 31.25 kBaud -- open

  #CASE BS2SX, BS2P
    MidiBaud    CON     $8000 + 60
#ENDSELECT

Channel         CON     0
NoteOn          CON     $90 | Channel
NoteOff         CON     $80 | Channel
StdVolume       CON     64                      ' half volume
MiddleC         CON     60
```

```
' -----[ Variables ]------------------------------------------------------

scan            VAR     Byte                    ' current scan
last            VAR     Byte                    ' last scan
changes         VAR     Byte                    ' changes between scans
cmd             VAR     Byte                    ' command (on or off)
note            VAR     Byte                    ' note to play
velocity        VAR     Byte                    ' volume level
idx             VAR     Nib                     ' loop counter


' -----[ EEPROM Data ]-----------------------------------------------------


' -----[ Initialization ]--------------------------------------------------

Setup:
  HIGH A2Dcs                                    ' deselect ADC0831
  last = Keys                                   ' no changes on start-up


' -----[ Program Code ]----------------------------------------------------

Main:
  DO
    GOSUB Get Keys                              ' check for key changes
    GOSUB Get Velocity                          ' get volume level
    GOSUB Play Notes                            ' update notes
  LOOP
  END


' -----[ Subroutines ]-----------------------------------------------------

Get Keys:
  scan = Keys                                   ' get current keys
  changes = scan ^ last                         ' find changes
  last = scan                                   ' save last keys
  RETURN


Get_Velocity:
  LOW A2Dcs                                     ' select ADC0831
  SHIFTIN A2Ddata, A2Dclock, MSBPOST, [velocity\9]
  HIGH A2Dcs                                    ' deslect ADC0831
  velocity = velocity / 2                       ' scale for MIDI
  RETURN


Play Notes:
  IF changes THEN
```

```
    FOR idx = 7 TO 0
      IF (changes.LowBit(idx)) THEN             ' look for note change
        cmd = NoteOn - ($10 * scan.LowBit(idx)) ' construct command
        note = MiddleC + (7 - idx)              ' bit 7 is MiddleC
        SEROUT MidiOut, MidiBaud, [cmd, note, velocity]
      ENDIF
    NEXT
  ENDIF
  RETURN
```