Stamp Applications no. 14 (April '96):

When You Feel that Need for Speed: Hotter Hardware and Swifter Software

Accelerating PBASIC
And More Boolean Logic,
by Scott Edwards

THE STAMPS are no speed demons. Fortunately, in most of the control-oriented applications for which the Stamps are appropriate, speed is not an issue.

But in some cases, speed is critical. In my previous column on rotary encoders (October '95) we discovered that at the BS1's execution rate of 2000 instructions per second (3000 for the BS2), it simply could not keep up with the pulses from a low-resolution rotary encoder turned by hand. The solution was to use faster hardware: a Counterfeit controller (BS1 clone; Sources) accelerated to 8000 instructions per second.

This month, we're going to look at another speed-critical application and see how dramatic acceleration was achieved through changes in *software*. Then we'll look at a brand-new compiler that puts the pedal to the metal by generating assembly-language code from your PBASIC programs. Finally, in our BASIC-for-beginners series, we'll examine PBASIC's Boolean logic operators and summarize their uses in decision-making and bit manipulation.

Scanning LED Display. Almost two years ago, I presented a project called the Picture Stick in *Electronics Now* (Oct. '94). The stick consists of a microcontroller-driven array of 16 LEDs. When you wave the stick through the air, viewers see a two-dimensional image created by the rapidly changing pattern of lights.

Recently I received e-mail from a high-school instructor, Ralph Wallio of Indianola High School (IHS), who wondered if the project could be adapted to run on a spinning wheel and to display text instead of individual frames of graphics. I told him it could, but the modification would require rewriting the assembly language code for the project—something he'd have to undertake himself. He countered that it ought to be possible to create his spinning text display with a 4x accelerated Counterfeit (BS1 workalike in kit form; Sources). I had my doubts.

I was wrong, having underestimated the ingenuity and stick-to-it-iveness of Mr. Wallio and his students.

Their first try met with my pessimistic expectations, though. The main loop of the program had to look up a series of values representing columns of LEDs that made up a message ("IHS AUDITORIUM CREW" in this case); see figure 1. Here's the code they used:

```
Display:
FOR B3=0 TO 86
  LOOKUP B3,(126,9,9,9,126,...),B4
  PINS=B4
NEXT
```

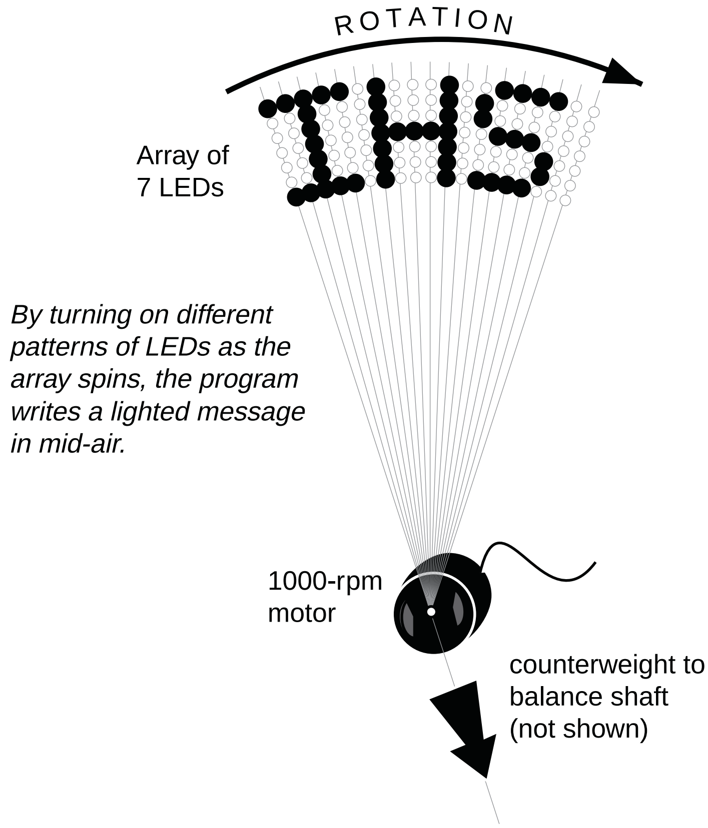

Figure 1. Spinning LED display required a fast program.

Variable B3 was used to pick one of 87 values from a lookup table and store it in B4. Then B4 was copied to the pins of the PBASIC chip. The IHSers measured the execution time of this loop and found it to be 1.18 seconds at PBASIC's normal execution rate.

Their plan was to spin the LED array with a 1000-rpm motor. That's 16.6 revs per second, or about 60 milliseconds (ms) per revolution. The program was about 20 times too slow!

Their ace in the hole was that the Counterfeit can be accelerated 4x by substituting a faster crystal. But the program would still miss the mark by a factor of 5.

My first speedup suggestion was to eliminate the step of copying B4 to PINS. After all, PINS is a byte variable like any other, with the sole exception that its bits protrude into the outside world. I asked the IHS crew to change the Lookup instruction to read:

```
  LOOKUP B3,(126,9,9,9,126,...),PINS
```

and recheck the timing. It improved only modestly, to 1.14 seconds.

My next suggestion was more radical. Instead of using the Lookup instruction, why not store the data in EEPROM, and use Read to retrieve it? My thought was that Lookup's extra features (error-detection, 16-bit values) might make it slower than Read. I suggested:

```
EEPROM 0,(126,9,9,9,126...)
Display:
FOR B3=0 TO 86
  READ B3,PINS
NEXT
```

Paydirt! Timing improved to 0.160 seconds. With the Counterfeit's 4x hardware speedup this would drop to 40 milliseconds, meeting the

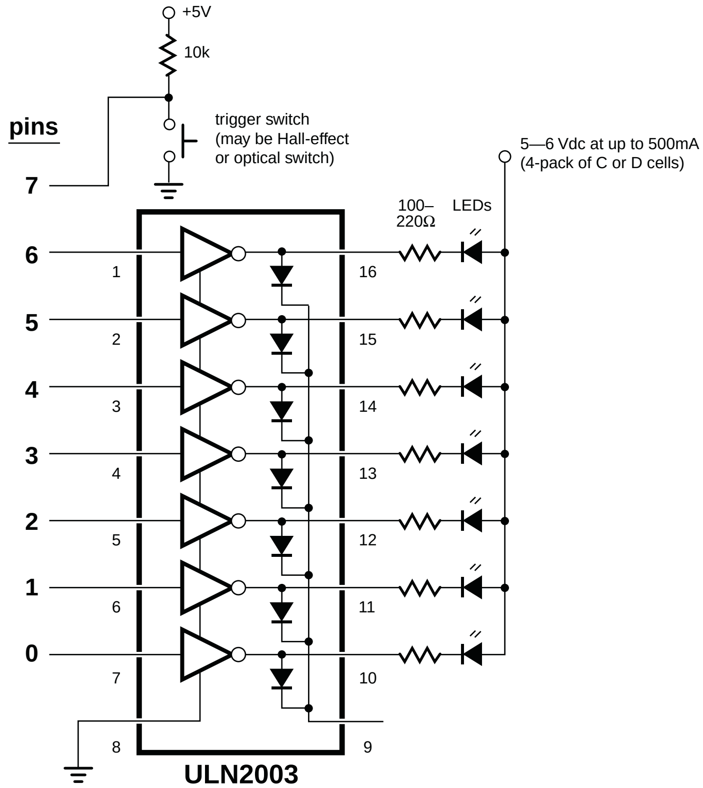

Figure 2. Schematic of the scanning LED display.

60-ms requirement.

There are two lessons here. One is the specific news that Read is dramatically faster than Lookup. If you don't need Lookup's special features, use Read. The other lesson is more general: If you're not getting the performance you need from a program, the answer may lie in changing your approach, rather than just fiddling with the existing code.

Listing 1 is the finished program with the EEPROM data pared down to just the first three letters of the message, "IHS," in the interest of space. Figure 2 is the schematic. The purpose of the trigger switch is to ensure that the LED message always started at the same point in the rotation of the array. This made the message stable through successive spins of the array; just like horizontal sync in a TV.

NOTE:

Obsolete information deleted.

BASIC for Beginners. The last couple of installments we've been looking at decision-making with Boolean logic. We've discovered that AND and OR have very precise meanings that can be used to specify the results of IF/THEN instructions under a range of possible conditions.

This time we're going to expand on those ideas, seeing how logic operators work on bits.

You may have noticed that the terms AND and OR turn up in a couple of places in the PBASIC manual; first in IF/THEN, and again in LET with the math operators. Only the AND and OR used with LET are represented by symbols, not words. Puzzling.

The logic operators listed under IF/THEN alter the outcome of a decision, while those under LET alter the contents of a variable. Same idea; different application. And under the heading LET, our buddies AND and OR are joined by XOR (as well as variations with NOT, which we'll see later). Let's take a look at how these operators work on bits, starting with AND.

AND (symbol: &)

first bit	second bit	result
0	0	0
0	1	0
1	0	0
1	1	1

You can test the correctness of this list, called a "truth table," by writing a little PBASIC program:

```
let bit0 = 0
let bit1 = 1
let bit2 = bit0 & bit 1
debug bit2: end
```

Try running the program several times with different combinations of bit0 and bit1 until you've convinced yourself of the *truth* of the truth table.

Here are truth tables for the other operators:

OR (symbol: |)

first bit	second bit	result
0	0	0
0	1	1
1	0	1
1	1	1

XOR (exclusive OR; symbol: ^)

first bit	second bit	result
0	0	0
0	1	1
1	0	1
1	1	0

Where things really get interesting with the logic operators is when you apply them to groups of bits, like bytes and words. A byte is a clump of eight bits, and PBASIC has 14 byte variables named b0 through b13. In PBASIC, to specify the bits of a byte, you precede the list of 1s and 0s with the symbol %. Using this notation, let's look at the effects of AND, OR, and XOR on some sample data:

```
%00001111 AND %10101010 = %00001010
%00001111  OR %10101010 = %10101111
%00001111 XOR %10101010 = %10100101
```

The logic shown in the truth tables applies to each pair of bits individually. In the AND example, the rightmost pair of bits are 1 and 0, so the resulting bit is 0, just like the truth table shows. The second pair of bits from the right are 1 and 1, so the corresponding bit of the result is 1.

Let's verify this with a PBASIC program:

```
let b0 = %00001111
let b1 = %10101010
let b2 = b0 & b1  ' AND the bytes.
let b3 = b0 & b1  ' OR the bytes.
let b4 = b0 & b1  ' XOR the bytes.
debug "AND result: " %b2
debug "OR result:  " %b3
debug "XOR result: " %b4
```

The logic operators are good for more than just mental exercise. Each has a classic application:

AND: The distinctive characteristic of AND is that there's only one way to get a 1 in the result when you AND two values, and that's by having a 1 in that position of both input values. Put another way, wherever there's a 0 bit in one of the input values, that bit in the output is guaranteed to be 0. So programmers say that AND is can *strip* or *mask off* particular bits of a variable.

Look at the example above. The 0s in the lefthand four bits changed those bits to 0s in the result. The 1s in the righthand four bits faithfully copied the bits of the other value (1010) to the result.

OR: Where AND can clear selected bits to 0, OR has the ability to set selected bits to 1. OR's truth table shows that if either or both bits contain 1, the outcome will be 1. You can see this effect in the example above.

XOR: This is an interesting operator. The XOR of two bits is 1 if one or the other (but not both) input bits is 1. One way to look at this is that XOR allows you to selectively invert particular bits—anywhere one value contains a 1, the result will be the opposite of the other bit. 1 XOR 1 = 0; 1 XOR 0 = 1. Any bit XORed with 0 is unchanged.

Another way of looking at XOR is as a test for matching bit patterns. XORing two identical bytes together yields a byte containing all 0s. XORing two non-identical bytes together yields a byte with 1s in the bit positions at which the two input bytes are different. Very handy.

Variations with NOT. I mentioned that each of the logic operators in PBASIC has a variation with NOT; AND NOT, OR NOT and XOR NOT, symbolized &/, |/, and ^/. The truth tables for these guys are the same as shown above, but with the result bits inverted (1 replaced by 0 and 0 by 1).

Summary. You've worked hard to understand the logic operators, so here's a reward: a tiny program that creates a marquee-style chase light. Connect LEDs to the outputs (as shown in the Button listing in the Counterfeit Development System manual) and run the following:

```
let dirs = %11111111
pins = %10101010
again:
  let pins = pins ^ %11111111
  pause 200
goto again
```

The program sets all pins to output (dirs=); turns on every other pin (pins=), then enters a loop marked by “again:” that XORs the pins with byte consisting of all 1s. This has the effect of inverting all of the output bits, creating the marquee-flashing effect. After a 200-millisecond pause, the *again* loop repeats.

NOTE:

The Counterfeit Controller is no longer manufactured. See Parallax (www.parallaxinc.com) regarding their OEM-BS1 kit.

NOTE: This article was originally published in 1996. The Stamp Applications column continues with a changing roster of writers. See www.nutsvolts.com or www.parallaxinc.com for current Stamp-oriented information.

Listing. Scanning LED Display

```
' Program: SCANLED.BAS (Scanning LED display developed by Ralph Wallio
' and his crew at Indianola High School)
' This is a partial listing of the scanning LED display program. I've
' shortened it to conserve space, since the majority of the program
' is EEPROM data making up the message "IHS AUDITORIUM CREW." Just
' the portion "IHS" is enough to convey the principle.

DIRS=%01111111   ' Bit 7 is input for trigger; the rest outputs.

' LED bit patterns forming the letters IHS.
EEPROM 0,(65,65,127,65,65,0,127,8,8,8,127,0,49,73,73,73,69)
'         ---------------   -------------   --------------
'                I                H                S

' Wait until triggered before displaying bit patterns on LEDs.
Trigger_Loop: BUTTON 7,0,0,0,B2,1,Display
              GOTO Trigger_Loop

Display: FOR B3=0 TO 16     ' Get data from EEPROM.
           READ B3, PINS    ' Copy each EEPROM byte to pins.
           PINS=0           ' Blank the LEDs.
         NEXT               ' Get next byte of data.

         GOTO Trigger_Loop ' Done. Wait for next rotation.
```