The *Nuts and Volts* of BASIC Stamps

Parallax, Inc.
www.**parallaxinc**.com

Nuts and Volts
www.**nutsvolts**.com

Column #36, February 1998 by John Barrowman, Ken Gracey and Jeff Martin

Advanced Stamp Programming Techniques

Synthesizing a Real-Time Clock, Floating Point Math Simulation, and Counting Pulses on Multiple Pins

Have you ever wondered what Stamp topics Parallax technical support staff would discuss if given an opportunity to write a column for Nuts & Volts? How about synthesizing a real-time clock, floating point math simulation, and counting pulses on multiple I/O pins. These are Jeff and John's "Advanced Stamp Programming Techniques," named because they should provide a basis for more complex programming.

Real Time Clock with User Inputs

The BASIC Stamp doesn't have a real-time clock, and implementing a real-time clock is often done with external clocks from Dallas Semiconductor or Solutions Cubed. This example shows how to use four discrete registers and the PAUSE command to implement a real-time clock in PBASIC. A real-time PBASIC clock works well if the Stamp can be calibrated to a reliable source.

The timing is adjusted by using the BASIC Stamp II's PAUSE command — a PBASIC instruction that causes a delay for x milliseconds. The variable must be 16 bits or smaller (less than 65535). The example program begins by pausing 985 milliseconds (almost a second) since the rest of the program takes 15 milliseconds to execute.

Time is displayed on a Scott Edwards serial LCD using the SEROUT command from pin 0. If you don't own an LCD display, simply replace the SEROUT instruction with DEBUG and the time will be displayed on your computer screen.

The PAUSE command buys the time you need to insert additional PBASIC code and use the additional EEPROM space for other tasks. The Stamp II executes about 4,000 instructions per second, theoretically, an additional 400 lines of code could be executed in 100 milliseconds (less than the 985-millisecond delay). If you add code, you will have to decrease PAUSE to ensure the Stamp is still being a reliable clock. To ensure consistent operation, be sure each iteration of your added code takes the same amount of time to execute.

Now it's all up to you to add features to the sample. Try using pushbuttons to adjust the time, or modify the program to receive a serial message containing the current time before proceeding. Try to collect time-stamped data from the Dallas Semiconductor 1620 digital thermometer. Program Listing 36.1 shows this example.

```
' Program Listing 36.1
' TIME.BS2
' John Barrowman, Parallax Technical Support Engineer
' Stamp Applications: Nuts and Volts February 1998
' This program demonstrates how to implement a real-time clock on BASIC
' Stamp. Time is metered using the PAUSE command. Seconds, minutes, hours,
' and the AM/PM status are stored in discreet registers.  This program
' uses a 2x16 Scott Edwards LCD Display to show the formatted time.

Baud    con    $4054                   ' 9600, N,8,1 (Inverted Data)
hrs     var    byte                    ' Holds the Hours part of time
mns     var    byte                    ' Holds the Minutes part of time
sec     var    byte                    ' Holds the Seconds part of time
ampm    var    bit                     ' Keeps track of night and day.

INIT:
        pause 500                       ' Must wait for display to power up
        serout 0,Baud,[254,2,"  Current Time"] ' Write header

START:
        pause 985                       ' Adjust this to 'tune' the clock.
        serout 0,Baud,[254,175,"  ",dec2 hrs,":",dec2 mns,":",dec2 sec]
        if ampm = 1 then PRN_AM         ' After writing the time determine
```

```
        serout 0,Baud,["  pm"]       ' AM/PM status and write to display.
        goto INC_TIME
PRN_AM:
        serout 0,Baud,["  am"]

INC_TIME:
                                     ' Every iteration of the loop should
        sec = sec + 1                ' take exactly one second.  So, we'll
        if sec < 60 then START       ' increment the seconds register each
        sec = 0                      ' iteration.  If the seconds register
        mns = mns + 1                ' overflows (>59), we must: clear it,
        if mns < 60 then START       ' increment the minutes register, and
        mns = 0                      ' check the minutes register for over
        hrs = hrs + 1                ' flow. If the minutes register over-
        if hrs <> 12 then ROLL       ' flows, we must clear it then incre-
        ampm = ampm ^ 1              ' ment the hours register. And so on.

ROLL:
        if hrs < 13 then START
        hrs = 1
        goto START
```

Algebraic Tricks with the BASIC Stamp – Simply Floating Point Math Solution

BASIC Stamps work only with integer math, which means that no fractions are allowed and decimal results are truncated. Expressions and results are presented as integers and the remainder is simply lost. This means the Stamp does not support floating-point math. However, with a few algebraic tricks and additional variables, you can easily achieve simple floating-point math.

For example, let's assume you want to calculate and display a number from 0.0 to 100.0, and this number results from the expression $X = A / B$. A is any number from 0 to 1900, and B is any number from 20 to 160. If A = 1326 and B = 85, the equation result should be 15.6 based on the number of significant digits. The Stamp would return a number of 15.

There's a way to work around this. The decimal point can be moved one position to the right by multiplying one multiplicand by 10 before executing the equation. For example:

(A)/(B)=(X) is the same as (A)(10)/(B)=(X)/10 because the 10s cancel each other.

And with actual numbers:

(1326)/(85) = (15.6) is the same as (1326)(10)/(85)=(156)/(10)

However, the Stamp cannot perform the last division by 10 without truncating the number to 15. We can perform the last division logically when we display the number. In other words, we will use the equation:

(A)(10)/(B) = (X)

See Program Listing 36.2 as an example.

```
' Program Listing 36.2
' Tenths.BS2
' Jeff Martin, Parallax Technical Support Engineer
' Stamp Applications: Nuts and Volts February 1998
' First, set up our variables
A        VAR     WORD
B        VAR     WORD
X        VAR     WORD

'Initialize the values
A = 1326
B = 85

'Now do our modified calculation
'(multiply the first number by 10) then divide
X = A*10 / B

'Now display it on the PC screen
DEBUG DEC X/10, ".", DEC X DIG 0
```

So how does the code work? The last line of code with DEBUG is where we pull the last trick. It displays "15.6" on the PC screen. First, it displays the integer portion of the number by dividing it by 10: DEC X/10. Second, it places a decimal point on the screen: ".". And third, it uses the handy DIG operator to grab a single digit — the ones-place digit — from the number and display it: DEC X DIG 0. It is important to note that we still cannot exceed the maximum size for a number at any time during the calculation. The part we need to worry about is the A*10 operation. Since we can only hold a number up to 65535, this operation limits us to a maximum "A" value of 6553 since 6553 * 10 = 65530.

So you've mastered simple floating point math, but what if you don't care about the fractional portion of a result, but you need to multiply a variable by a floating point number? For example, X = A * 4.35 where A is a number from 52 to 196. The Stamp

does not understand the number 4.35 in the equation.

Your first thought may be to use a variation of the first trick, to multiply the floating point constant by 100, evaluate the expression, and then treat the right-most two digits of the result as the fractional portion (which we don't care about). This seems to work:

A*4.35=X is the same as: A*(4.35*100)=X/100

Since we are not concerned about the fraction in the result, the Stamp can feel free to truncate the fraction in the result. Now the equation is

X=A*(4.35*100)/100   or   X=A*435/100

If A is 52, the expression results in 226 (exactly what we expect). If A is 196, the expression results in 197 (we expected 852). So what happened? Unfortunately, 196*435=85260, which is much larger than the Stamp's 16-bit space, and so the result returned is wrong. The largest number the Stamp can handle is 65535.

Multiplying a Variable by a Known Fraction Using */

If you'd like to multiply a number by a known fraction, there's a little known and often misunderstood operator in the BS2-IC that can save the day. It is called the "star slash" operator ("*/"). The */ operator works with a 16-bit space, but it treats it as an eight-bit integer and eight-bit fraction.

This works with a 16-bit space, but it treats it as an eight-bit integer and eight-bit fraction. This accuracy works in most cases, even though it is not absolute.

Using the above floating point example of 4.35, we will convert the number on each side of the decimal point — the 4 and the .35 — into the proper 16-bit form. This needs to be done before writing the expression code in PBASIC. The equation for handling this is the following:

Converted Number = INT (INT(number) * 256 + FRAC(number) * 256)

Or =   INT (INT(4.35) * 256 + FRAC(4.35) * 256)

which is

=	INT (1024 + 89.6)
=	INT (1113.6)
=	1113

The final number looks strange, but the BS2-IC's */ operator translates the number to 4.34765625, very close to 4.35. Here's the source code which displays the number we expect to be debugged to your PC screen using the */ operator.

In case you are wondering, the fractional portion of the floating point number is expressed internally as 89/256 = 0.34765625. Combined with the integer portion, it is 4 89/256 or 1113/256. If you used a calculator and multiply this into 196 as in 196*1113/256, you'll end up with 852.14. This is exactly what the */ operator does, except that it truncates the fractional portion. Hence the name. First, it multiplies using * and divides using the /. Again, you might notice that 196*1113 results in a number much greater than 16 bits wide. The */ operator keeps track of the intermediate result in 32 bits, then divides by 256 (discarding the first eight bits) and truncating off the upper eight bits, leaving a nice 16-bit number.

```
' Program Listing 36.3
' Starslsh.BS2
' Jeff Martin, Parallax Technical Support Engineer
' Stamp Applications: Nuts and Volts February 1998
' First, set up our variables
A       VAR     WORD
X       VAR     WORD

'Initialize the value
A = 52

'Now do our modified calculation, multiply by 4.35 'with the */ operator
X = A */ 1113

'Now display it on the PC screen and move to the 'next line
DEBUG DEC A," * 4.35 = ",DEC X,CR

'Now let's try one more time
A = 196

X = A */ 1113

DEBUG DEC A," * 4.35 = ",DEC X
```

Simultaneously Counting Pulses

The Stamp's COUNT function counts cycles on a pin for x milliseconds. This works well if the Stamp has nothing else to do but wait for a pulse on a pin, but what if you need to count pulse transitions on multiple I/O pins while other code is executing using minimal code space? Using logic statements and storing pin transitions lets the Stamp count pulses on more than one pin.

This project was developed on the BASIC Stamp Activity Board with "pluggable jumpers" and a Scott Edwards 2x16 Serial LCD display. John and Jeff use these tools to expedite a quick project or simulate a customer's application over the telephone. If you don't have a serial LCD, replace the SEROUT with DEBUG to send the results to your PC screen and remove the "low 0" instruction since the display I/O is not used. You'll need the BASIC Stamp Activity Board unless you have a few pushbuttons and resistors to simulate the board's predetermined circuitry.

The program (Program Listing 36.4) begins by defining the variables. The display I/O pin and variables are then initialized. The START subroutine reads the status of two I/O pins, then calculates and adds the transitions only. The results are stored in two new registers and the data is written to a display. If P10 is grounded (button depressed), then the registers are cleared. The program repeats itself in an endless loop.

Like the real-time clock example, this project also provides the Stamp room for additional tasks. By placing additional source code before or after the SEROUT instruction, the Stamp can handle additional tasks while counting pulses. We'll leave this project up to you, but remember that the maximum count rates depend on the overall execution of one loop iteration. It isn't possible to press the BASIC Stamp Activity Board's buttons quicker than the one loop of PBASIC code is executed, so you could combine this project with the floating point math simulation described above.

```
' Program Listing 36.4
' COUNTS.BS2
' John Barrowman, Parallax Technical Support Engineer
' Stamp Applications: Nuts and Volts February 1998
' This program demonstrates a way to count pulses when the count function
' is not appropriate. Pulse transitions are counted on pins 8 and 9 use-
' ing calculations instead of logic statements. This minimizes code space
' used and reduces the execution time. This program writes the results to
' a 2*16 Scott Edwards LCD Display. This project was developed on the new
' Basic Stamp Activity Board. On the BSAC board, inputs pins P8-P11 are
' pulled high and grounded when the pushbuttons are pressed. Pressing the
' button connected to P10 results in clearing the counter registers.

Baud    con    $4054            ' 9600, N,8,1 (Inverted Data)
xctr    var    byte             ' Holds the #transitions on P8.
```

```
yctr    var    byte            ' Holds the #transitions on P9.
xprev   var    bit             ' Holds the previous state of P8.
yprev   var    bit             ' Holds the previous state of P9.
xcurr   var    bit             ' Holds the current state of P8.
ycurr   var    bit             ' Holds the current state of P9.

INIT
        low 0                   ' Initialize the display I/O line.
        xctr = 0                ' Initialize all variables.
        yctr = 0
        xcurr = in8
        xprev = xcurr
        ycurr = in9
        yprev = ycurr
START
        xcurr = in8             ' Read input pins.
        ycurr = in9
        xctr = xctr + (xcurr ^ xprev & xprev)' Calculate and add
        yctr = yctr + (ycurr ^ yprev & ycurr)' transitions only.
        xprev = xcurr                        ' Update the 'prev'
        yprev = ycurr                        ' registers.
        serout 0,Baud,[254,  2,"X Counts: ",dec3 xctr] ' Write data to the
        serout 0,Baud,[254,175,"Y Counts: ",dec3 yctr] ' display
        if in10 = 1 then START               ' If P10 is grounded
        xctr = 0                              ' then clear the
        yctr = 0                              ' count regisers.
        goto START                           ' Repeat forever.

' Please note that the calculations for XCTR and YCTR are different. XCTR
' is counting the negative transitions, and YCTR is counting the positive
' transitions. This is deliberate to exemplify both techniques.
```