

Exterminating Common Bugs With Little-Known Stamp Info

Miscellaneous Tips and Techniques, by Scott Edwards

THE STAMP is so easy to use and program that I sometimes forget what a complex little computer it is. Fortunately, I have you, the readers of this column, to remind me. This month I'll present an assortment of bug fixes, explanations, hints, and tips based on questions and comments I've received by e-mail.

Programs without END

In figure 1, an LED is wired so that it lights when a low (0) appears on pin 0. Suppose you wrote a program that consisted of just the following line:

```
LOW 0 ' Light the LED.
```

What would happen? When the Stamp first turned on, all pins would be in input mode, and the LED would be dark. A fraction of a second later, the LOW instruction would execute, changing pin 0 to output/low. The LED would light.

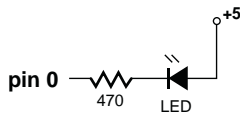


Figure 1. An implicit END instruction makes the LED blink.

After that, there are no more instructions for the Stamp to execute. With its work complete, the Stamp does the only sensible thing; it executes an END instruction and goes to sleep.

During sleep, the LED stays on, but every 2.3 seconds it blinks off for a fraction of a second.

The Stamp resets every 2.3 seconds during sleep, and its pins go into input mode during reset. This reset takes about 18 milliseconds—long enough for you to see a visible blink in the LED.

This behavior can affect other devices as well. Suppose the Stamp were connected to a serial device. Every 2.3 seconds it would lose control of the pin being used for output. The glitch could cause “garbage characters” to appear at the serial receiver.

Problems caused by the implicit END instruction are rare in finished programs, since most are loops. But during casual testing it's quite common to type in a few lines to see what happens. If you want to avoid letting the Stamp execute that unwritten END, try this:

```
LOW 0          ' Light the LED.  
stop: GOTO stop ' Freeze here.
```

Mighty Frustratin' Power Dangers

While we're on the subject of resets, let's talk about unwanted resets caused by an inadequate power supply.

The Stamp and Counterfeit both come with 9V battery clips. A built-in voltage regulator drops this input voltage to a steady 5-volt supply for the rest of the electronics.

The regulators are quite efficient. They only require an input of slightly over 5 volts in order

to maintain regulated 5 volts out. However, if the input voltage falls below their *dropout voltage*—the voltage at which they can no longer regulate the output—all bets are off.

This can happen more often than you might think, even with a battery that measures well over 5 volts. Although a battery is shown as a single component in schematic diagrams, it acts like two components, a voltage source and a series resistance. As the battery wears out, the voltage decreases *and* the series resistance increases. It's the increased series resistance that usually gets you.

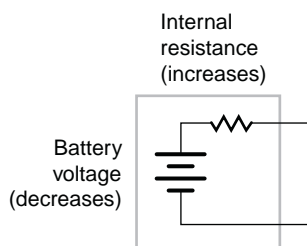


Figure 2. As a battery wears out, its internal resistance increases.

When a 9V battery is down to 7 volts, it's still well above the voltage regulator's dropout voltage, which may be as low as 5.1 volts. But the battery's internal resistance has increased to, say, 50 ohms. If your circuit draws a total of 50 mA, the voltage drop across that internal resistance becomes $0.05 \times 50 = 2.5$ volts. (Ohm's Law—current in amperes times resistance = voltage.) The battery delivers just $7 - 2.5 = 4.5$ volts to the regulator's input. The result is that the supply is no longer in regulation, and a reset can occur.

I saw a particularly acute example of this in a recent on-line help message. A user had connected a radio-control servo (positioning motor with built-in electronics) to the Stamp, and was powering both from the same four-pack of AA penlight cells. The circuit was totally erratic. The cause was fairly obvious—whenever the batteries (only 6 volts total) tried to deliver large currents to move the servos, the large voltage drop across their internal resistance caused a brownout to the Stamp, which reset itself.

This problem is more severe with newer Stamps, which have a “brownout reset” IC on board. These reset the processor any time the regulated supply falls below 4 volts. Older Stamps had no such circuit, and could continue to operate at 2.5 volts or less. This sometimes caused problems, because different parts of the circuit shut down at different voltages, causing a kind of temporary insanity. (The Counterfeit also has a brownout circuit, which kicks in at approximately 3.4 volts.)

Here are some hints for tracking and fixing power-supply related problems:

- If a circuit behaves erratically and you suspect that varying loads are to blame, try removing those loads and rerunning the program. If it works OK without the load, you need to beef up the power supply, or use a separate supply.
- Don't draw more than 50 mA from the Stamp's built-in 5-volt supply, or 100 mA from the Counterfeit's supply.
- If your application involves motors or other high-current loads, use separate sets of batteries for the Stamp and the load. Just connect the grounds together.
- If you have multiple Stamps or other circuits running from a single power supply, don't daisy-chain the wiring; wire each module back to the power supply terminals separately.

Divide and Conquer

PBASIC can perform simple arithmetic on 16-bit positive integers. A 16-bit variable can range from 0 to 65535. What if your application involves calculations that generate numbers larger than 65535?

I recently answered an e-mail help request that illustrates the problem and a way to solve it. Dan DiLuzio was building a pH meter, a device that measures the acidity or alkalinity of a solution on a scale of 0 to 14. He had a pH probe and some analog circuitry that converted the pH to a 0- to 5-volt output, where 0 volts represented a pH of 14 and 5 volts a pH of 0.

An LTC1298 analog-to-digital converter (ADC) (described in *Stamp Applications* no. 4, June 1995) allowed the Stamp to read in the 0- to 5-volt signal as a 12-bit number from 0 to 4095. To

convert this number to a pH reading, Dan needed to use the following formula:

$$\text{pH} = 14000 - (\text{ADC_result} * 14000/4095)$$

This formula uses the number 14,000 to represent the full-scale reading of 14. Just moving the decimal point three places to the left (14.000) would scale the readings to real pH units. The trouble is that when the pH is 0 the ADC reads 4095, and the calculation $\text{ADC_result} * 14000$ equals 57,330,000. That's beyond PBASIC's maximum integer of 65,535.

Dan was starting to think that he'd have to abandon some of the precision offered by the 12-bit ADC, and content himself with readings of 0 to 14 or 0.0 to 14.0 at best. But I showed him how to chop the problem up into smaller pieces by factoring, preserve precision, *and* stay within the bounds of 16-bit math.

I started by factoring the constants 14,000 and 4095 into smaller integers. Since 4095 is close to 4096 (2^{12}) I cheated and changed it to the more convenient value. Here are the factors:

$$14,000 = 7 * 5 * 5 * 5 * 2 * 2 * 2 * 2$$

$$4096 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2$$

I then rearranged Dan's formula into a series of smaller alternating multiplications and divisions. I eliminated cases in which the number would be multiplied by 2 then divided by 2, since these have no net effect. I wound up with:

$$\text{ADC_result} * 7 / 4 * 5 / 4 * 5 / 4 * 5 / 4$$

Trying this with the troublesome maximum value of 4095, I got an answer of 13,995—within a fraction of a percent of the correct answer of 14,000. And no calculation exceeded PBASIC's limit. I passed this info to Dan, along with the suggestion that he use a correction factor or calibrate the analog circuitry to eliminate the small bias introduced by fudging 4095 to 4096 and by the lost fractions due to integer arithmetic. He reported back: problem solved!

Timing NOT to Set Your Watch By

On the BS1-series Stamps, you can generate or measure pulses with 10-microsecond (μs) resolution. On the BS2, pulse resolution is 2 μs .

This fine timing precision leads some users to believe that the Stamps are also very accurate. They're not, and the discussion that follows should interest both Stamp users and folks who never quite grasped the distinction between *precision* and *accuracy*.

First, you must understand that all Stamp timing is referenced to an internal oscillator whose frequency is set by a ceramic resonator. Resonators are similar in properties to quartz crystals, but are built to withstand rough treatment like shock and vibration. The tradeoff is that a mediocre crystal's actual frequency is usually within ± 0.005 percent [50 parts per million (ppm)] of its nominal or rated frequency, while a good ceramic resonator may be within ± 1 percent.

Putting that into perspective, a clock controlled by a 50-ppm crystal might be off by 4 seconds a day; with a 1-percent resonator the error could be close to 15 *minutes* a day!

On the BS1, the nominal 4-MHz resonator can operate at 3.96 to 4.04 MHz. A BS2's 20-MHz resonator can range from 19.8 to 20.2 MHz. (These ranges are based on the hypothetical 1-percent tolerance; some resonators are as sloppy as 3 percent.)

If a BS1 is used to measure a 200-millisecond (ms) pulse, the result should be 20,000 units of 10 μs apiece. But if the resonator is off by 1 percent, the actual reading could range from 19,800 to 20,200.

This resulted in a Stamp user calling me for help. He was trying to use a BS1 to produce pulses that were measured by a BS2 in order to simulate his final application involving motor-speed measurement. The two Stamp clocks were not only off frequency, but may have been off in opposite directions (above and below nominal), resulting in an error of close to 2 percent, and hundreds of pulse-timing units.

He was shocked, having expected that the fine *precision* of the pulse commands (units of 10 and 2 μs) also meant high *accuracy*, with results within a few units of right on the money. Nope; accuracy can't be any better than the timing reference, which is only good to ± 1 percent. Bear this in mind when designing any time-critical application.

Bowlegs, Brackets and Bugs with the BS2

If you bought one of the early BS2s, you received the “preliminary documentation,” a dump of just the essential information required to get started with the new Stamp. Because of some major improvements in PBASIC’s structure, the BS2 dialect is not backwardly compatible with the BS1.

One change that’s causing headaches for users who are accustomed to the BS1 is the new role of parentheses (like the ones surrounding this text). On the BS1, bowlegs enclose lists, such as data items for Serout, Lookup, Lookdown and Sound. The BS1 does not support the use of parentheses to change the order of math and logic operations. The BS2 does.

As a result, Parallax had to find a new way to enclose lists of data items, which might include expressions enclosed in bowlegs, and it chose square brackets [like these].

The change is sufficiently subtle that I didn’t notice it on first reading of the new docs, but the host software brought it to my attention several times when I wrote my first BS2 program! I’m hoping this explanation will help you remember when to use bowlegs and when to use brackets.

Here are more tips for BS2 users:

- The serial commands now require baud rate expressed in terms of microseconds – 20. To convert a desired baud rate, divide it into 1, multiply the result by 1 million and round off, then subtract 20. Try 9600 baud: $1/9600 = 104.167 \times 10^{-6}$. Multiply by a million and round off: 104. Finally, subtract 20: $104 - 20 = 84$. That’s your baud-rate timing. For other baud-mode options (expressed as hex numbers in the documentation), just add them to the calculated timing value. For example, to invert the serial output at 9600 baud, use $\$4000 + 84$.

- If you’re planning to use the new X-10 remote control command Xout, or the synchronous-serial instructions Shiftin and Shiftout, make sure to get the new application notes from Parallax. And get the whole notes, not just the source code. The accompanying text and illustrations are probably more important than the programs to really understanding how these new features work.

- Having trouble downloading programs to a BS2? The problem may be that the host software can’t figure out which serial port the BS2 is connected to. You can specify the port when you boot the software by adding the switch /1 for com 1 or /2 for com 2. For example:

```
STAMP2 /1
```

launches the host program for com port 1.

- The new BS2 method of reading and writing the I/O pins seems to be puzzling some people. It’s probably because you no longer have predefined variables called “pin0, pin1...pin7.” Under the new PBASIC, you have a pair of 16-bit variables called INS and OUTS which hold the input and output states of the pins.

To use the individual bits of these registers, you need to define bit variables, like so:

```
in_pin1 var INS.bit1 ' Pin 1 input.
```

This may take some getting used to, but in the long run you’ll appreciate the flexibility of the new approach.

- If you’re using the BS2 to display data on the LCD Serial Backpack, there’s a neat trick you should know. By using the new comparison feature of Lookdown and the decimal-digit function Dig, you can right-align a numeric display. See the listing for an example.

NOTE:

Obsolete information deleted.

NOTE: This article was originally published in 1995. The Stamp Applications column continues with a changing roster of writers. See www.nutsvolts.com or www.parallaxinc.com for current Stamp-oriented information.

' Program: RJ_DEMO.BS2 (Right-justified printing with Stamp 2)

' This program demonstrates how to print numbers on the LCD Serial Backpack with right justification. This means that the ones place is always in the same location on the screen regardless of the number of digits in the number. This program uses a Lookdown table as a function that returns the number of decimal digits in a given 16-bit value.

I	con	254	' Instruction toggle command.
ClrLCD	con	1	' Clear-LCD instruction.
prn_at	con	140	' Display RAM, address 13 (128+12).
j	var	word	' 16-bit counter variable.
pos	var	byte	' Cursor position to print at.
numDig	var	nib	' Number of digits of number.
N96N	con	\$4054	' 9600 baud, inverted, no parity.

```
low 0                ' Make the serial output low
pause 1000           ' Let the LCD wake up.
serout 0,N96N,[I,ClrLCD,I] 'Clear the LCD.
serout 0,N96N,["Number: "] ' Print the fixed label.
```

' The loop below counts from 0 to 20000 and displays the current count on the LCD Serial Backpack. The Lookdown table determines how many digits are in the current value of the count in order to position the printout aligned on the rightmost digit. It works by determining whether a given number is less than 10 (1 digit); between 10 and 100 (2 digits), etc. The table is good for values from 0 to 65534.

```
Loop:
  for j = 0 to 20000      ' Count to 20,000.
    lookdown j,< [0,10,100,1000,10000,65535],numDig      ' Get # of digits.
    pos = prn_at - numDig  ' Adjust the screen position.

    serout 0,N96N,[I,pos,I,DEC j,"    "]      ' Print j at adjusted screen pos.
    pause 50                                ' Slow the count a little.
  next
  goto Loop                                ' Do it again.
```