Column #38, April 1998 by Jon Williams:

Getting Back to the BS1-IC

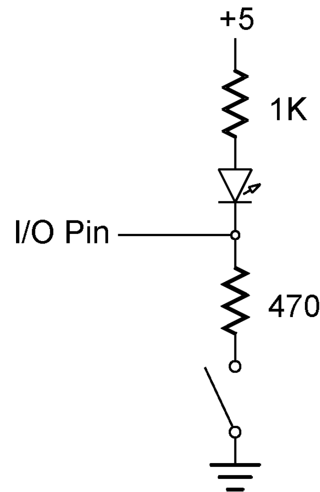
Having used it since the spring of 1994, I still turn to the BS1 as my default Stamp. Yes, the BS2 has more code space, more I/O pins, and extended features in PBASIC. I find, however, that the BS1 is suitable for most of my personal projects. It's only when I know I'll need the extra pins or code features that I use the BS2.

Back in 1994, when there was no BS2 to turn to, BASIC Stamp had to resort to clever coding and I/O schemes to make projects work. The introduction of serial peripherals like Scott Edwards' Serial LCD Backpack and Stamp Stretcher pulled a lot of us out of jams. This month, we're going to discuss some hardware and software techniques that just might help you out of a jam and squeeze your project back into the BS1.

Sharing Pins

From time to time, we'll run out of pins before we run out of code space. This is especially true with the BS1. With careful (read that again, I said, "careful") code writing and I/O hardware, we can share one or more pins. The trick is modifying the Dirs variable mid-stream. We don't normally do this, so we need to be cautious. Take a look at the circuit in Figure 38.1.

Figure 38.1: Circuit to display status of I/O pin



This circuit allows us to display the status of the I/O pin when it's an output, and to read it when it's configured as an input. Notice that this circuit is con-figured for negative logic. What this means is that a low output will light the LED, a high output will turn it off. Don't be alarmed by this; you can easily change positive logic (HIGH = On) to negative logic with the ^ (Exclusive OR) operator (0 ^ 1 is 1).

Okay, let's use this circuit. Program Listing 38.1 is a very simple program that waits for you to push the button connected to Pin0, then flashes the LED five times. Notice that the Dirs variable is reset to configure the pin before each section. In case you're wondering, the purpose of the 470-ohm resistor is to protect the I/O pin in case you push the button when the pin is in an output condition. If you pressed the button when the Stamp had placed a HIGH (5 volts) on the pin, you'd have a short to ground. The resistor limits the current through this short to a safe level.

For another good example of pin sharing, refer to the Stamp Application Note #1, "LCD User-Interface Terminal." You can download this App Note from Parallax. By the way, the BASIC Stamp Activity Board that I mentioned last month comes with four of these LED/switch circuits.

Sizing Up Your Code

Within minutes of opening my first BS1, I had it up and running code. Within hours, I was writing useful programs. Within days, I ran into the dreaded “Error – EEPROM Full” message from the Stamp compiler. “Yikes! What do I do now?” What I had to do was find code-saving techniques, but the pseudo-analog bar-graph display of the BS1 compiler didn’t help much with my experiments; its resolution is much too coarse.

At the time, the BSAVE command wasn’t really documented, but was made known through a small App Note that came with the BSLOAD utility. BSAVE writes a 256-byte binary file that is an image of the Stamp’s EEPROM for your program.

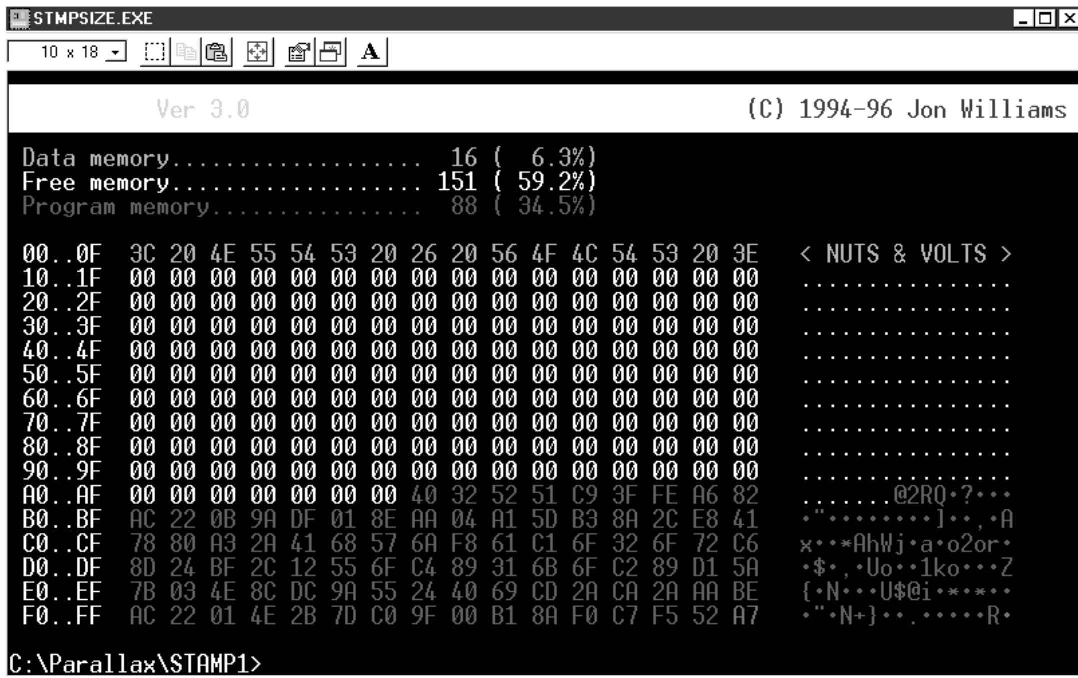
The EEPROM contains your code and user data and — more importantly — an address of the last EEPROM byte used by the program code. With this knowledge, I was able to write a program called STMPSIZE.EXE (for Stamp Size). What this program does is read CODE.OBJ (the file created by BSAVE) and displays the results with a straightforward text out-put.

Using STMPSIZE is easy; just include the BSAVE command in your program and hit [Alt]+[R] (run). The nice thing about BSAVE is that the file is created before the compiler actually tries to down-load the program to the Stamp. This means that you don’t need to con-nect your Stamp to do program analysis.

Let’s look at a quick example. I found, through experimenting and using STMPSIZE, that I can gain a few precious bytes of code space by replacing a LOOKUP table with EEPROM statements and using READ to get the data. This is particularly useful if the data is used more than once and is not part of a subroutine.

```
' Program Listing 38.1
' Nuts & Volts: Stamp Applications, April 1998
GetPin: Dirs = %00000000      ' all pins are inputs
IF Pin0 = 1 THEN GetPin       ' wait for pin to go low
Blink: Dirs = %00000001       ' make pin 0 an output
  FOR B0 = 1 TO 5
   PAUSE 500
   Pin0 = 0                   ' LED on
   PAUSE 500
   Pin0 = 1                   ' LED off
  NEXT
GOTO GetPin                   ' do everything again
```

Figure 38.2: StampSize Utility Screenshot



Listing 38.2a is a small program that sends a message to a standard (not serial) LCD. Analysis with STMPSIZE shows us that the program uses 111 bytes. Now replace the output section with the code in Listing 38.2b and check it again. Now we've used 104 bytes (code plus data) — a savings of seven bytes. This may not sound like much but, believe me, it is! You can find STMPSIZE.ZIP on the CD-ROM with this book. Figure 38.2 shows a screenshot of STMPSIZE with the modified code.

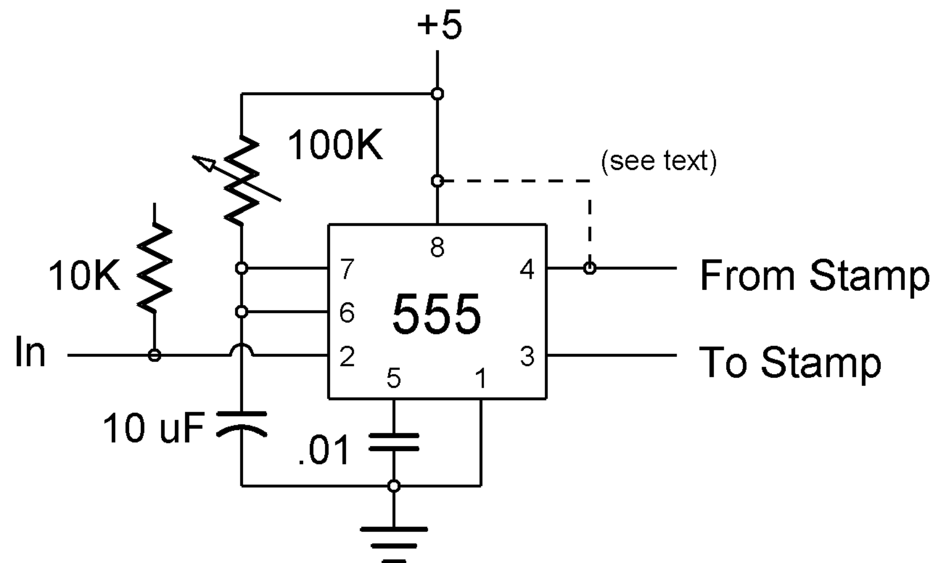
No Missed Inputs

Once you get comfortable with the Stamp, you'll probably want to start connecting it to external devices and circuits. The trouble starts whenyour external circuit does something when you're not expecting it. Your program may well be run-ning some process that takes longer than your input is active. There are two things we can do, and I'm going to recommend them both.

The first thing we can do is stretch a short input, thus making our input pin active long enough to come around and see it. The type of circuit we need is called a one-shot or, for you extreme technocrats, a monostable multivi-brator. What we have to do is design the out-put to be long enough for us to finish what we're doing so that it can see the input.

There are probably as many one-shot designs as there are people on the planet, so I'm going to use the ubiquitous 555 timer chip. They're tough, easy to use, and very easy to find. Take a look at the circuit in Figure 38.3, which is a 555-based adjustable one-shot. A low on Pin 2 of the 555 will cause the output (555 Pin 3) to go high for a period determined by R1 and C1 (multiply R1 by C1 to find the time).

Figure 38.3: 555-based adjustable on-shot



You need to be careful not to make your 555 output pulse too long, otherwise you might miss another input. If you have an extra Stamp pin, you can fix this problem by connecting an output to Pin 4 of the 555. This pin is the Reset input. When taken low, it will reset the 555 output. I used this technique in a project with great success. If you don't connect Pin 4 of the 555 to the Stamp, connect it to +5 volts.

Okay, what about software? If you analyze most of your programs, you'll probably find that they're a collection of small modules that run over and over in a continuous loop. Let's say, for example, that we want to check an input, then do three other things. Normally, our program flow would look something like this:

Look for the input
Do Process A
Do Process B
Do Process C
Do everything again

If we're worried about missing the input, we'll want to check it more often. Like this:

Look for the input
Do Process A
Look for the input
Do Process B
Look for the input
Do Process C
Do everything again

At first glance you might say, "Oh, that's easy, check the input in a subroutine." Sure, that's one way to do it, and it will work in most cases. There will be times, though, when you run out of GOSUBs (you overrun the GOSUB stack). We need another approach.

BRANCH is one of the most powerful — yet least-used — elements of PBASIC. What BRANCH does is direct the flow of your program based on the value of its control variable. Take a look a Program Listing 38.3. Notice that we start by checking our input (simulated with a DEBUG statement) then branch to the current process. At the conclusion of each process, the program loops back to the beginning.

Using BRANCH to control program flow is very powerful. The real power comes when you allow each process to dynamically modify the control variable. Using this technique will cause your program to become very dynamic in its operation, without a confusing mess of IF-THENs and GOTOs.

A New Serial LCD

There's been a lot of "knock-off" serial LCDs since Scott Edwards first introduced them, but none very notable until now. By the time you read this, the ILM-216 should be available from Scott Edwards Electronics. I had a chance to work with a pre-production unit and I must say that I'm thrilled. Take a look at the specifications:

- Two lines of 16 characters
- LED backlighting
- Output for a piezo speaker
- Four switch inputs
- EEPROM storage of six custom characters
- EEPROM storage of two-line "splash" screen

Not bad, huh? A two-line LCD and four switch inputs for the price of a single I/O pin. The ability of the ILM-216 to store custom character maps and a splash screen means that we don't have to implement those features in code. This gives us more space for the meat of our program.

The ILM-216 comes with Scott's typically excellent documentation, so I'm not going to go into a lot of detail, except to say that its serial input and serial output lines can be tied together through a 1K resistor. Since the BS1 can send and receive serial data with the same pin, this configuration is a potential project saver. I'm working on a BS1networking project that will use the ILM-216 for the node display. Look for details in a future article.

Beginner's Corner

The LED (light emitting diode) is probably the most common status indicator used in electronic circuits. Those of us with design experience rarely think about the details of incorporating them; those without experience (or who are new), should. LEDs are not like light bulbs. That is, you can't connect them directly between an I/O pin and ground (or +5) — they need to have a current-limiting resistor placed in the series to protect the LED and the Stamp. So how do we figure the correct size for that resistor? With Ohm's Law and a pocket calculator.

Without going into a lot of engineering, here are the basics: The current in a series circuit is the same through all components, and the total circuit voltage is divided among the components. In order to operate correctly, an LED needs a certain current (called forward current) at its specified forward voltage.

Let’s say we have an LED with a forward voltage of two volts with a forward current of 10 milliamps. With a five-volt output from our Stamp, our resistor needs to handle three volts at 10 milliamps. Using the Ohm’s Law formula R = E/I (resistance equals volts divided by current), we end up with a calculated resistance of 300 ohms (3 / 0.01). Chances are that your corner electronics distributor (the guys who ALWAYS ask for your address) doesn’t carry a 300-ohm resistor, but they do have them at 330 ohms. This will work fine. It’s always safer to use a slightly larger resistor until you get the hang of circuit design.

Now, what we’ve just gone through only considers one pin. What if we want to light six LEDs? Will it work? Sorry, no. Well, not without problems. You see, the Stamp 1 can only source 40 milliamps of current. What this means is that the Stamp can only provide 40 mA when the outputs are high (+5). We can get a little more juice (up to 50 mA) from the Stamp by connecting the LEDs to +5 and activating them with a low output (like Figure 38.1). This is called sinking the current.

What we have to do is divide our available current by the number of outputs. Be careful here. Most general-purpose resistors have a 5% tolerance. This means that the resistance could be up to 5% lower than the marked value. Lower resistance means more current. You can either figure in the tolerance, or start your calculations without trying to use all the available current.

We’ll be conservative and sink (active outputs are low) 40 mA of current. This gives us about 6.6 mA per LED, which means we’ll need a resistor for each of 454 ohms. The closest standard value is 470 ohms, so that’s what we’ll use.

```
' Listing 38.2
' Nuts & Volts: Stamp Applications, April 1998
' -----[ Title ]-----------------------------------------------------
'
' File...... SSDEMO.BAS
' Purpose... STMPSIZE Demo
' Author.... Jon Williams
' E-mail.... jonwms@aol.com
' WWW....... http://members.aol.com/jonwms
' Started... 07 MAR 1998
' Updated... 07 MAR 1998
' -----[ Program Description ]---------------------------------------
'
' LCD Connections:
'
' LCD (Function) Stamp
' --------------------- -----
' pin 1 Vss Gnd
' pin 2 Vdd +5
' pin 3 Vo Gnd (or wiper of 10K pot)
' pin 4 RS Pin 4
' pin 5 R/W Gnd
' pin 6 E Pin 5
' pin 7 DB0 Gnd
' pin 8 DB1 Gnd
' pin 9 DB2 Gnd
' pin 10 DB3 Gnd
' pin 11 DB4 Pin 0
' pin 12 DB5 Pin 1
' pin 13 DB6 Pin 2
' pin 14 DB7 Pin 4
' -----[ Revision History ]------------------------------------------
'
' -----[ Constants ]-------------------------------------------------
'
SYMBOL RS = 4 ' Register Select (1 = char)
SYMBOL E = 5 ' LCD enable pin (1 = enabled)
' LCD control characters
'
SYMBOL ClrLCD = $01   ' clear the LCD
SYMBOL CrsrHm = $02   ' move cursor to home position
SYMBOL CrsrLf = $10   ' move cursor left
SYMBOL CrsrRt = $14   ' move cursor right
SYMBOL DispLf = $18   ' shift displayed chars left
SYMBOL DispRt = $1C   ' shift displayed chars right
SYMBOL DDRam = $80    ' Display Data RAM control
' -----[ Variables ]-------------------------------------------------
'
SYMBOL char = B1      ' character sent to LCD
```

```
SYMBOL index = B2 ' loop counter
' -----[ EEPROM Data ]--------------------------------------------
'
' -----[ Initialization ]-----------------------------------------
'
Init: Dirs = %00111111       ' set 0-5 as outputs
Pins = %00000000             ' clear the pins
' Initialize the LCD (Hitachi HD44780 controller)
'
LCDini: PAUSE 500            ' let the LCD settle
Pins = %0011                 ' 8-bit mode
PULSOUT E, 1
PAUSE 5
PULSOUT E, 1
PULSOUT E, 1
Pins = %0010                 ' 4-bit mode
PULSOUT E, 1
char = %00001100             ' disp on, crsr off, blink off
GOSUB LCDcmd
char = %00000110             ' inc crsr, no disp shift
GOSUB LCDcmd
char = ClrLCD
GOSUB LCDcmd
' -----[ Main Code ]----------------------------------------------
'
Start: FOR index = 0 TO 15
LOOKUP index,("< NUTS & VOLTS >"),char
GOSUB LCDwr
NEXT
END
' -----[ Subroutines ]--------------------------------------------
'
LCDcmd: LOW RS ' enter command mode
' then write the character
' Write ASCII char to LCD
'
LCDwr: Pins = Pins & %11010000' save 7, 6 and RS; clear bus
Pins = char / 16 | Pins      ' output high nibble
PULSOUT E, 1                 ' strobe the Enable line
Pins = Pins & %11010000
Pins = char & $0F | Pins     ' output low nibble
PULSOUT E, 1
HIGH RS ' return to character mode
RETURN
```

```
' Listing 2b
' Nuts & Volts: Stamp Applications, April 1998
' Replace sections of Listing 2a with this code
' -----[ EEPROM Data ]----------------------------
'
EEPROM ("< NUTS & VOLTS >")
' -----[ Main Code ]------------------------------
'
Start: FOR index = 0 TO 15
READ index, char
GOSUB LCDwr
NEXT
END
```

```
' Listing 38.3
' Nuts & Volts: Stamp Applications, April 1998
' -----[ Title ]-----------------------------------------------------
'
' File...... BRANCH.BAS
' Purpose... Demonstrates the use of BRANCH
' Author.... Jon Williams
' E-mail.... jonwms@aol.com
' WWW....... http://members.aol.com/jonwms
' Started... 07 MAR 98
' Updated... 07 MAR 98
' -----[ Program Description ]---------------------------------------
'
' This program demonstrates the use of BRANCH as a means of replacing
' confusing IF-THENs and GOTOs in a dynamic program
' -----[ Revision History ]------------------------------------------
'
' -----[ Constants ]-------------------------------------------------
'
' -----[ Variables ]-------------------------------------------------
'
SYMBOL state = B1
' -----[ EEPROM Data ]-----------------------------------------------
'
' -----[ Initialization ]--------------------------------------------
'
Init: state = 0
' -----[ Main Code ]-------------------------------------------------
'
Main: DEBUG "Checking the Input",CR ' check the input (sim)
' do other important things here
       BRANCH state,(ProcA, ProcB, ProcC) ' do the next process

ProcA: DEBUG "In Process A",CR
       PAUSE 1000 ' simulate timing of process
       state = 1 ' point to next process
```

```
       Goto Main

ProcB: DEBUG "In Process B",CR
       PAUSE 1000
       state = 2
       Goto Main
ProcC: DEBUG "In Process C",CR
       PAUSE 1000
       state = 0
       Goto Main
END
' -----[ Subroutines ]----------------------------------------------
'
```