Column #35, January 1998 by Jon Williams:

Back to the Basics (PBASIC) With Fun and Games

If necessity is the mother of invention, then experience is our grandest teacher. The problem for many Stamp experimenters is the lack of PBASIC experience. Most of us rush headlong into our first Stamp project without the slightest idea of how we will actually get to our goal. Invariably, we stumble around longer than necessary before finishing the project.

The basis for this installment of Stamp Applications is my first complete Stamp program. Like most projects, it started with a need — but not mine.

Way back in the spring of 1994, when I was playing with my first Stamp, I found a somewhat desperate posting on the Parallax BBS. The author of the posting was a gentleman named Jim, and it seemed that Jim had promised his young son that he would build an electronic roulette game. Jim figured it would be a snap with the BASIC Stamp; then he tried and hit a wall. The BBS posting followed shortly thereafter. Jim's experience certainly isn't unique. In fact, I believe that anyone who ever designs anything — hobbyist or professional — will hit this "wall" at some point in their experience.

Okay, so how do we soften the inevitable impact with the wall? I said it back in August, and will repeat myself here: plan, plan, and plan. Perhaps this — the planning — is the sticking point. One of the tricks I suggested was to talk yourself through your project. I'll expand on this by suggesting that you ask yourself questions. Do this out loud so that you hear the words — they'll have more impact that way. And ask hard questions. Be tough with yourself at this stage and, generally, the actual code writing will go easier. Imagine that you were building the project for someone else. What questions should you ask your customer before proceeding?

This month, we're going to build Jim's roulette game with the BS1. Along the way, we're going to ask and answer many questions necessary to complete the project. I think this will be very helpful to those of you who are just getting started. We're also going to step back just a bit and go over some programming basics, starting with how programs flow. A large majority of the E-Mail I've received since taking this column has asked me to revive Scott's "BASIC For Beginners" feature. I will also go into more detail on my program analysis. So, until I hear from you differently, we'll try this format (basics, plus a project) each month. And I promise that not all projects will be as simple as this one. My goal is to teach good programming fundamentals so that, unlike Jim way back when, you'll be able to solve your own programming problems. Let's get started.

Going with the Flow: PBASIC Program Structure

You've probably read or been told that a computer program is simply a list of statements and commands that the computer — in our case, the BASIC Stamp — executes. While this is true, it's important to understand that, unlike a list, a program rarely runs from start to finish in a linear fashion. There will be many linear sections, but branching and looping will interrupt the overall linear nature of a program.

A branching command is one that causes the flow of the program to change from its linear path. In other words, when the program encounters a branching command, it will, in almost all cases, not be running the next [linear] line of code. The program will go somewhere else.

There are two categories of branching commands: unconditional and conditional. PBASIC has two commands, GOTO and GOSUB, that cause unconditional branching. GOTO simply redirects the program to another location. The new location is specified as a parameter of the GOTO command and is referred to as an address. Remember that addresses start a line of code and a colon (:) must follow the address label. You will typically see a GOTO at the end of the main body of code, forcing the main program statements to run all over again.

GOSUB redirects the flow of the program to another section and then comes back to continue with the line of code following the GOSUB command. This allows you to reuse a section of code, saving valuable EEPROM space. Like GOTO, an address must be specified for the GOSUB command, and the last statement of the subroutine (the section of code called by GOSUB) must be RETURN.

There are a couple of details that you should keep in mind when using GOSUB. The Stamp 1 uses byte variables B12 and B13 (W6) for the GOSUB stack. This means that you cannot use these variables if your program contains any GOSUBs. And the sky is not the limit. The GOSUB stack is used to hold the return address and, therefore, has a limit. It works like this: When your program encounters a GOSUB, it saves the address of the next line of code (you do not have to declare it), then the program branches to the address specified in the GOSUB command. When RETURN is encountered, the last address on the GOSUB stack is pulled off and the program branches back to that location. This is why no parameter is required with the RETURN statement, and it allows the same subroutine to be called from anywhere within your program. The Stamp 1 allows up to 16 GOSUBs to be stacked; the Stamp 2 allows up to 256.

Conditional branching is just that: conditional — it may or may not happen, depending on conditions. There are two PBASIC statements used for conditional branching: IF-THEN and BRANCH.

The PBASIC IF-THEN statement is different from other flavors of BASIC. In PBASIC, THEN is always followed by a valid program address (other BASICs allow a variety of programming statements to follow THEN). If the condition statement evaluates as TRUE, the program will branch to the address specified. Otherwise, it will continue with the next line of code. Look at this short example. It monitors a switch on Pin 0 and, when that switch is pressed (active low), an LED connected to Pin 7 is lit.

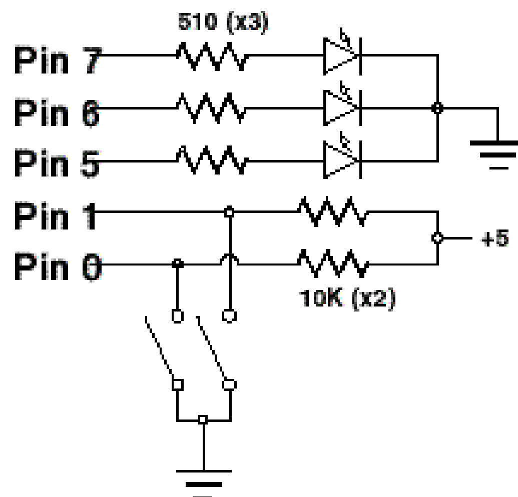
```
Init:  Dirs = %10000000
Pins = %00000000

Start: IF Pin0 = 0 THEN LedOn
               Pin7 = 0
               GOTO Start
LedOn: Pin7 = 1
               GOTO Start
```

Notice how the GOTO command (an unconditional branch) is used to cause the program to repeat itself (we'll talk more about looping in a bit).

The other conditional branching statement in PBASIC is called BRANCH. BRANCH is a little more complicated in its set-up, but very powerful in that it essentially replaces multiple IF-THEN statements. BRANCH requires a control variable and a list of addresses. Let's use some switches and LEDs again (refer to Figure 35.1 if you want to try this yourself) to demonstrate the BRANCH command. The purpose of this program is to monitor two switches and light an LED corresponding to the switch combination (left, right, or both).

Figure 35.1: LED Circuit to demonstrate BRANCH command




```
Init: Dirs = %11100000
Pins = %00000000

Start: B2 = Pins & %00000011
       BRANCH B2, (None, LLed, RLed, Both)
None:  Pins = 0
              GOTO Start
LLed:  Pins = %00100000
              GOTO Start
RLed:  Pins = %01000000
              GOTO Start
Both:  Pins = %10000000
              GOTO Start
```

The BRANCH command uses the control variable as an index into the list of addresses. Keep in mind that the first address will correspond to a control variable value of zero. Now, you may be wondering what happens if the control variable is greater than the number of addresses. In this situation, the BRANCH command is ignored and the program continues with the next line of code.

Like branching, the term "looping" should be pretty self-evident: looping forces a section of code to repeat. And, like branching, there are two categories: perpetual and counted. We've already seen perpetual looping in our short demo programs. Both programs used GOTO to force the main program statements to run again. This is perpetual looping. Make sure that this is what you want your program to do. Unless there are conditional statements somewhere in the loop code segment, your program cannot escape a perpetual loop.

The FOR-NEXT construct is used to cause a section of code to execute (loop) a specific number of times. FOR-NEXT uses a control variable to determine the number of loops. The size of the variable will determine the upper limit of loop iterations. For example, the upper limit when using a byte-sized control variable would be 255.

Structurally, FOR-NEXT looks like this:

```
FOR myVariable = startValue TO endValue
 'do something here
NEXT
```

"myVariable" is the control variable we just talked about. "startValue" and "endValue" may be variables or constants. Here's a simple example that flashes an LED connected to Pin 7 10 times:

```
Init:   Dirs = %10000000
Pins = %00000000

Start:  FOR B2 = 1 TO 10
          Pin7 = 1
          PAUSE 500
          Pin7 = 0
          PAUSE 500
          NEXT
        END
```

Here's how it works: On entry, the control variable is set to the starting value. Then the statements and commands between FOR and NEXT are executed. When NEXT is encountered, the control variable is compared with the ending value. If the control variable is equal to or greater than the ending value, the program continues with the line following NEXT, otherwise the control variable is updated (usually incremented by one) and the loop statements are run again.

Those are the basics, now let's get into the details. First, the control variable does not have to be changed by one. You may specify the amount of change with the STEP parameter. If you add 'STEP 2' after the 10 in the program above, the loop will only run five times.

There will be many occasions when you want to use the control variable inside your loop, and sometimes you'll want this value to count backwards. No problem. Just swap the starting and ending values and specify a negative STEP size. Like this:

```
FOR B2 = 10 TO 1 STEP -2
 ' do something here}
```

(using some bits, eliminating others) is called masking. Since the Stamp SOUND command generates white noise (a hissing sound) with values greater than 127 ($7F), we use $7F as our mask. The next line of code creates a pattern for the LEDs by masking the highest bit of the tone variable. I used binary notation in the code to make it easier to read. This pattern is placed on the LEDs and then we RETURN to the calling code.
Once we get back to the attention-getting routine, we send our sound to the speaker, pause a bit, then check the status of the switch with the BUTTON statement. The value of 255 in the third parameter forces BUTTON to debounce the switch (clean up the contact noise) without auto repeating. Since the user will not be holding the switch down, auto-repeat is not necessary. When the switch is pressed, the program branches to Spin, otherwise it falls through to a GOTO command that forces the attention-getting routine to run again.

The Spin section is the heart of the program. On entry, we turn on all the LEDs, pause a bit, and then turn them off. This is our simulated reset. A FOR-NEXT loop is used to control the spin. Our set-up is very simple since Jim suggested about 25 loop iterations. The first line of code in the loop calls GetRnd again to generate a pseudo-random value and put it on the LEDs. When we get back, we output a "bonk" tone. This tone simulates the tumbling wheels of a slot machine. Now you understand why we didn't output our random tone in the GetRnd subroutine. After playing the tone, we wait just a bit and then update our delay variable to simulate the natural decay of mechanical wheels.

Through experimenting, I found that adding 10% to the delay through each loop gave fairly natural results. Since the Stamp uses integer mathematics, we multiply delay by 11, then divide by 10. Be a little cautious with this technique. If your intermediate results are greater than 65535, you'll get something other than you expect. It's always a good idea to work through the math manually to make sure your results will be predictable.

After the final spin, the result (stored in pattern) is checked to see if all the LEDs are on. If they are, you win and a little tune is played. If you don't win, you'll hear a groan from the speaker and the program will return to the attention-getting routine.

The routine YouWin is used to flash the LEDs and play the tune. Notice that two loops are used, one nested inside the other. The LED pattern and the tone used in the Spin2 loop are stored in LOOKUP statements; one of the most powerful aspects of PBASIC. LOOKUP uses a control variable (in our case, spin2) as an index into a table. So long as the control variable is not greater than the number of elements in the table, the destination variable is loaded with the table value pointed to by the control variable. If the control variable is greater than the number of table elements, the destination variable does not change.

After our little celebration, we go back to Attn and start all over again.

That seems like an awful lot of text to describe such a simple program, doesn't it? Well, as I said back in August, we programmers often get ourselves in trouble by taking things for granted. It's never too late to start developing good code-writing habits:

- Plan your programs (ask questions — then answer them).
- Use SYMBOLs to make the program easy to read and update.
- Write and test small sections of code.

Now get out your breadboard and build Jim's game (see Figure 35.2 for the schematic). Since this program only uses about half of the Stamp's code space, there's plenty of room for new features. Ask yourself what you'd like to add, answer the questions, then dig in and have a great time.

```
' ----[ Title ]-----------------------------------------------------------
' Program Listing 35.1
' File...... LASVEGAS.BAS
' Purpose... BS1 Slot Machine Game
' Author.... Jon Williams
' E-mail.... jonwms@aol.com
' WWW....... http://members.aol.com/jonwms
' FTP....... ftp://members.aol.com/jonwms/stamps
' Started... 13 MAR 94
' Updated... 01 DEC 97


' ----[ Program Description ]---------------------------------------------
'
' Simple BASIC Stamp slot machine
'
' Connections:
'
' - LEDs connected to pins 0 - 5 through 510 Ohm resistors (1 = ON)
' - 45 Ohm speaker connected to pin 6 through 10 uF electrolytic cap
' - N.O. switch between pin 7 and ground -- pin 7 pull up through 10 KOhm

' ----[ Revision History ]------------------------------------------------
'
' 15 MAR 94 : Version 1 complete and working
' 01 DEC 97 : Updated and reformatted for Nuts & Volts


' ----[ Constants ]-------------------------------------------------------
'
SYMBOL  LEDs    = Pins
SYMBOL  Spkr    = 6
SYMBOL  Swtch   = 7


' ----[ Variables ]-------------------------------------------------------
'
SYMBOL  rndW    = W0
SYMBOL  delay   = W1
SYMBOL  pattern = B4
SYMBOL  tone    = B5
SYMBOL  swData  = B6
SYMBOL  spin1   = B7
SYMBOL  spin2   = B8


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

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

```
'
Init:    Pins = %00000000                        ' setup port
        Dirs = %00111111
        swData = 0                              ' clear switch workspace


' ----[ Main Code ]----------------------------------------------------
'
' This section generates random patterns on the LEDs and plays a random
' tone through the speaker.  This acts as an attention getter ("play me")
' and seeds the random number generator so that the game plays differently
' each time.  Section repeats until button is pushed.
Attn:    GOSUB GetRnd                 ' get random number
        SOUND Spkr, (Tone, 3)         ' play the random tone
        PAUSE 100
        BUTTON Swtch, 0, 255, 10, swData, 1, Spin
        GOTO Attn

' The heart of the program.  We start first by 'reseting' (all LEDs on,
' then off momentarily), then the wheel is spun.  I found that 25 turns
' of the 'wheel' seemed about right.  A 'bonk' is output through the
' speaker with each wheel spin.  Natural wheel spin decay is created by
' multiplying the the decay variable by 110% through each rev.  If all
' LEDs are lit at the end, "You Win", and the LEDs dance a bit.  If you
' lose, and you nearly always do, an obnoxious groan is output.
'
Spin:    LEDs = %00111111                      ' look like we're resetting
        PAUSE 750
        LEDs = %00000000
        PAUSE 500
        delay = 75                            ' initial delay value

        FOR spin1 = 1 TO 25                   ' spin wheel - 25 revs
          GOSUB GetRnd                        ' get and display pattern
          SOUND Spkr, (100, 2)                ' bonk sound for each rev
          PAUSE delay
          delay = delay * 11 / 10             ' increase delay by 10%
        NEXT spin1

        IF pattern = %00111111 THEN YouWin    ' but what do you win?
        SOUND Spkr, (50, 125)                 ' groan!
        LEDs = %00000000                      ' clear the LEDs
        PAUSE 1000                            ' rest a bit
        GOTO Attn                             ' here we go again...


YouWin: FOR spin1 = 1 TO 5                    ' a little marquee action
          FOR spin2 = 0 TO 3
            LOOKUP spin2, ($00, $0C, $12, $21), LEDs
            LOOKUP spin2, (110, 113, 116, 119), tone
            SOUND Spkr, (Tone, 3)
```

```
           PAUSE 65
         NEXT spin2
       NEXT spin1
       LEDs = %00000000
       PAUSE 1000
       GOTO Attn


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

GetRnd: RANDOM rndW                          ' get a random number
       tone = rndW & $7F                     ' do not allow white noise
       pattern = rndW & %00111111            ' mask for 6 LEDs
       LEDs = pattern                        ' show new pattern
       RETURN
```