Column #30, August 1997 by Jon Williams:

PBASIC Programming with Style

I have thoroughly enjoyed my 29-column runas host of Stamp Applications, but it's time for some new blood. Please welcome my hand-picked replacement, Jon Williams, as your new host. Jon will dazzle you with fresh insights on Stamp programming and project design. Many of you already know him as a star contributor to the Stamps Internet mailing list, and author of many of the Stamp example programs we've all come to depend on. I can't wait to see what he comes up with next! – Scott Edwards

Based on the number of magazine articles on the subject, the popularity of Scott's column, and so many work-alike products, it's clear that the Parallax BASIC Stamp microcontroller has made quite an impact on the hobbyist electronics market. For my money, the strength of this stodgy little controller is its easy-to-learn programming language: PBASIC.

The purpose of this article is to help you start writing better PBASIC programs. PBASIC programs with style, that is.

I've been writing in PBASIC for about three years, and have received a number of requests for programming assistance from both newcomers and experienced programmers alike. The newcomers generally don't know where to start; the experienced programmers often find themselves handcuffed by the Stamp's limited resources.

There are no gigabyte hard drives or megabyte memories with the Stamp, so resource planning and conservation is mandatory.

Let's examine several aspects of PBASIC code writing that are important to your project's success: planning; Stamp variables and their conservation; the effective use of PBASIC symbols; and program format and structure. If all goes well, you'll be writing better-behaved, higher-performance Stamp programs in less time than you're spending now. Are you ready? Okay then, let's get started!

Plan Your Work, Work Your Plan

You've heard it a million times: plan, plan, plan. Nothing gets a programmer into more trouble than bad or inadequate planning. This is particularly true with the Stamp as resources are so limited. Most of the programs I've fixed were "broken" due to bad planning and poor formatting which, in turn, lead to errors. Once you've read and digested the following information, you'll be better prepared to plan your Stamp programs. In the meantime, here are some tricks I use during program development:

Trick #1. Talk yourself through the program. Don't just think it through, talk it through. Talk to yourself — out loud — as if you were explaining the operation of the program to a fellow programmer. Often, just hearing our own voice is what makes the difference. Better yet, talk it out as if the person you're talking to isn't a programmer. This will force you to explain details.

Many times we take things for granted when we're talking to ourselves or others of similar ability. Please understand that I currently live alone, so this is not a problem. If you don't live alone, you may want to explain to your housemate(s) what you're doing and that there's no need to bring in the guys with the white jackets!

Trick #2: Design the details of your program on a white (dry erase) board before you sit down at your computer. And use a lot of colors. I love the freedom of a white board and find that if I'm stumped, the visual aspect of this exercise gives me new insight into the problem. The cool thing about a white board is that I can write code snippets or draw functional diagrams.

Trick #3: Get out a pad of small sticky-notes. Write module names or concise code fragments on individual notes and then stick them up on the wall. Now stand back and take a look. Then move them around. Add notes, take some away; just do what feels right to you. This exercise works particularly well with groups. How do you know when you're done? When the sticky-notes stop moving!

It's a good idea to record the final outcome before starting your editor. Another tip: this trick works even better when combined with trick #2. You can draw lines between and around notes to indicate program flow or logical groupings. If it's not quite right, just erase the lines or move some notes. Try this trick, it really does work.

Neatness Counts

I can hear the groans now ("Yeah, yeah, yeah... "), especially from my good friend Steve. Most of us just love to sit down at our PCs and start flogging out code like there's no tomorrow and, quite often, this gets us into trouble. Come on, admit it. How many times have you sat down to write what was supposed to be a "simple" program only to spend many frustrating hours trying to make it work? If this hasn't happened to you, you're either incredibly well disciplined or you haven't been coding very long.

Keep in mind that it takes no more time to write a neatly formatted program than it does to make a big mess. "So why bother with all this neatness nonsense?" you ask. Two reasons: debugging and distribution. And very often these reasons are combined. The debugging part is obvious, or should be. Take a look at this code fragment:

```
loop:
for b0 = 0 to 10
for w0 = 1 to 5
gosub myout
b1 = b0 * w0 - 1
next
goto loop
```

How much easier would this code be to trou-bleshoot if it looked something like this?

```
Main: FOR loop1 = 1 TO 10
  FOR loop2 = 1 TO 5
    GOSUB myOut
    lastOut = loop1 * loop2 - 1
    NEXT loop2
  NEXT loop1
  GOTO Main
```

Do you see the problems in the second fragment that isn't as obvious in the first? The first fragment is missing a NEXT to complete the FOR statement. A bigger problem is that the second FOR/NEXT loop overwrites B0 and B1 and will cause unexpected results (more on this later).

Yes, I agree that this can happen with the use of symbols; however, when planning your program and deliberately defining symbols, you'll find this occurrence less likely.

I find it easier to write a neat program by starting with a template. Below is a template that works equally well for Stamp 1 and Stamp 2 programs. The difference is in the code syntax that you would use in the actual program. For the purposes of this article, we'll use Stamp 1 details and syntax. Notice how the template structure helps me keep the code organized and contains contact information about the author. This is a good idea if you choose to launch your programs into cyber-space.

```
' ---- [ Title ]--------------------------
' File...... NEW.BAS
' Purpose... PBASIC Programming Template
' Author.... Jon Williams
' E-Mail.... jonwms@aol.com
' WWW....... http://members.aol.com/jonwms
' Started...
' Updated...

' ---- [ Program Description ] ------------
'
' ---- [ Revision History ] ---------------
'
' ---- [ Constants ] ----------------------
'
' ---- [ Variables ] ----------------------
'
' ---- [ EEPROM Data ] --------------------
'
' ---- [ Initialization ] -----------------
'
' ---- [ Main Code ] ----------------------
'
' ---- [ Subroutines ] --------------------
'
```

The second reason for using neat programming is distribution. What I mean by this is that we'll often share our programs with others. Many of us have programs floating around the Internet. Or we may share coding responsibilities with one or more programmers. A well-coded, neatly for-matted program is easier for your partners and others to understand and work on. And consider that you might come back to one of your own programs days, weeks, or even months later. A nicely coded program makes it easy to pick up where you left off. It's very frustrating to try to decode a poorly written program — particularly one that you've written yourself!

Understanding Stamp 1 Resources

For all its power, the Stamp 1 has a very limited set of resources: only 255 bytes of (tokenized) program space and just 14 bytes of user RAM. And two of these bytes are lost if GOSUBs are used. Perhaps some of you that are new to Stamps are thinking, "Holy cow, what can I do with that?" Actually, you can accomplish quite a lot, because the Stamp is very flexible in its use of RAM space. To get the most out of it, however, you really need to understand its organization and the ability to overlay variables.

PBASIC 1 allows three variable types: bits, bytes, and words. You can think of the user RAM space as 14 bytes (B0 through B13) or seven words (W0 through W6), or some combination of the two (note that I am not including Dirs and Pins in my definition of user variables). Additionally, W0 (word 0) can be addressed as 16 individual bits (Bit0 through Bit15). For this reason, I generally reserve W0 (B0 and B1) when I start writing a program. I also reserve W6 (B12 and B13) for GOSUBs.

A map of the Stamp 1's user-variable space looks like this:

W0	B0	Bit0 - Bit7
	B1	Bit8 - Bit15
W1	B2	
	B3	
W2	B4	
	B5	
W3	B6	
	B7	
W4	B8	
	B9	
W5	B10	
	B11	
W6	B12	
	B13	

Let's look at W0 in detail:

W0															
B1								B0							
Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

Do you see what's going on here?

The same physical space occupied by W0 is also occupied by B0 and B1 which, in turn, is also occupied by Bits 0 through 15. A change in Bit0, for example, will cause a change

in B0 and also in W0. Understanding and taking advantage of this structure can be very useful in the conservation of code space. Let me demonstrate:

Instead of using two lines of code:

```
B0 = 0
B1 = 0
```

you use one line of code like this:

```
W0 = 0
```

By using the latter technique, you can save up to three bytes of code space. This may not sound like much now, but it will make a big difference later when you're trying to squeeze that critical project into a Stamp 1.

Here's a useful partial listing for converting four Stamp 1 pins to eight via a 4094 shift register. It takes advantage of bit addressing of B0. And since most shift registers can be stacked, this routine can be called multiple times for even more outputs.

```
SYMBOL Strobe = 0
SYMBOL Data = Pin1
SYMBOL Clock = 2

SYMBOL outByte = B0
SYMBOL outbit = Bit7              ' high bit of outByte
SYMBOL shift = B2

' other code here...

Out_8:
  FOR shift = 1 TO 8
    Data = outBit           ' make bit available
    PULSOUT Clock, 1        ' clock the bit
    OutByte = outByte * 2   ' shift bits left
  NEXT shift
  PULSOUT Strobe, 1         ' data -> outputs
  RETURN
```

SYMBOLic Programming

One of the most overlooked features of PBASIC is the use of symbols. Stated simply, a SYMBOL is just a new name for a Stamp variable or constant. If you've never used symbols, you may be wondering what all the fuss is about. The effective use of symbols can make your programs more readable and, in many cases, self-documenting.

Using symbols can dramatically mini-mize the amount of commenting needed to effec-tively document a program. Take a look at this PBASIC 1 example:

```
SYMBOL LGear = Pin0   ' landing gear input
SYMBOL Alarm = Pin1

SYMBOL Up = 0
SYMBOL Down = 1
SYMBOL On = 1
SYMBOL Off = 0

Dirs = %00000010
Alarm = Off

CheckLG:
  IF LGear = Up THEN SndAlrm
  PAUSE 50
  GOTO CheckLG

SndAlrm: Alarm = On
```

Notice how the code is self-documenting and very easy to understand. This is particularly important when sharing programs or coming back to programs that you've put down for awhile. Without symbols or comments, the code might look like this:

```
Dirs = %00000010
Pin1 = 0

CheckLG:
  IF Pin0 = 0 THEN SndAlrm
  PAUSE 50
  GOTO CheckLG

SndAlrm: Pin1 = 1
```

I agree that, even without comments, this is not difficult to analyze for process, but not neces-sarily for purpose. And this is only a small frag-ment of a program. An entire program without symbols and with bad (if any) comments can be very difficult to work out. During a recent code review, I had to E-Mail the author three times to confirm my beliefs about what the program was supposed to be doing.

With some practice, you'll find that symbols remove a lot of confusion from program debugging. This is particularly true when you want to reuse a variable. You might, for example, need a loop counter in one part of your program and a memory address pointer in another. Your definitions would look something like this:

```
SYMBOL looper = B3
SYMBOL addr = B3
```

This is perfectly okay in PBASIC and it will make your program easier to read. The use of the standard variable name, B3, in the program may have led to confusion, and certainly would not have helped its readability. Using this technique will also help you conserve variable space. Just make sure that your program does not need to save the value of B3 (from our example) or you may get some unexpected behavior.

Naming Conventions and Formatting Guidelines

Now that we've seen how symbols can make our PBASIC code more readable, let's add a bit of formality and really make things neat. After starting with a template, I use the following formatting conventions in my PBASIC programs:

- Variables use mixed-case and begin with a low-ercase letter. Examples: lcdChar, hrs, iCount

- Constants (including subroutine address labels) use mixed-case and begin with an uppercase letter. Examples: LGear, SndAlrm, DlyTm

- PBASIC keywords use all uppercase. Examples: FOR, NEXT, GOSUB, RETURN

- White space (spaces, tabs, etc.) is used liberally to improve the readability of the program. White space has no impact of the size of the compiled (tokenized) program, so feel free to use it. I suggest starting the working code in the first column following a tab and indenting code within FOR/NEXT loops with two spaces.

In conclusion, writing code for your BASIC Stamp is a carefully crafted mix of art and sci-ence. The science part of this process has hard-and-fast rules, and our compilers quickly point out any violation of those rules. The art, however, is purely subjective; there are no rules. Find or develop a coding style that serves you and your programming process. The guidelines I've presented here have served me well and I hope that you find them useful. And do try the planning tricks that I outlined at the beginning — I'm sure you'll be happily surprised. If you have questions or comments, please feel free to send me an E-Mail.