Column #96 April 2003 by Jon Williams:

Thinking Like An Embedded Programmer

As I told you back in January, the new BASIC Stamp editor/compiler from Parallax has made programming BASIC Stamps easier than ever; even more fun for many that struggled with the older syntax. That said, the similarity of PBASIC 2.5 to many desktop versions of BASIC has caused some of us to [temporarily] forget that we're still dealing with and embedded controller. Above all else, we must remember that we don't have megabytes or RAM or gigabytes of storage space for our programs – something that we all take for granted with our desktop PCs.

So, this is one of those annual reminder columns: how to keep things thin and trim and running efficiently in the BASIC Stamp. We're also going to unravel some of the mysteries of the BASIC Stamp RAM memory. There's not much, only 26 bytes for our programs (not counting I/O variables). Thankfully, the BASIC Stamp's designer (ironically, a guy named "Chip") built some very cool tricks into the BASIC Stamp that let us make the most of that small memory space.

Project Development Pointers

I was working with a BASIC Stamp customer a couple weeks ago and was reminded about the importance of a development process. I use the same process for my hobby projects as I have used in my professional life. It's nothing magical:

1. What is my project supposed to do?
2. Design hardware to support #1.

3. Design software to work with design of #2.
4. Test for specification outlined in #1.

I can hear the big, thunderous "Duh!" echoing through your minds now. But, honestly, I work with BASIC Stamp customers nearly every day of the week, and there is rarely a lack of desire, but very frequently a lack of planning ... which is why I end up writing a column like this about every year; as a reminder to all of us that as simple as the BASIC Stamp is to use, our projects – in order to succeed without hassles – require proper planning. You know the saying, "You don't plan to fail, you fail to plan." Cheesy, but very true.

Back to my friend, for example. He was building an interesting clock project that would use displays that required special 4-bit decoders for each digit. No problem. Since he had the displays and the decoders working, he decided to start in on the software in an effort to be efficient (he wanted to specify a digit position and value, then call one routine to display it). While he ran into software problems that we'll deal with later, the bigger issue was hardware. You see, he was using four Stamp pins per decoder and ultimately wanted to display six digits. That means he needs 24 pins to control his final display. No problem, except that he was designing his project around a BS2sx that only has 16 I/O pins! Uh oh.... My point is that he spent a lot of his valuable time working on software that he couldn't ultimately use on the BASIC Stamp that he had.

This is why it is important to understand the results of a project and then design hardware that will accommodate that. Of course, my friend could have changed to a BS2p40 – but that would have been a significant expense. I pointed out to him that for about three bucks, he could use three 74HC595 shift-registers to drive his digit decoders. This was less expensive than changing to a BS2p, and in the end gave him more flexibility. "What was the outcome?" you wonder? Well, he took my advice on the 74HC595s and now is using just three – count 'em, three – BASIC Stamp pins that get expanded to 24 outputs from the '595s and he is able to drive his cool clock display. And his code is a lot thinner too!

The Anatomy of BASIC Stamp Variables

I use lots of tricks in my BASIC Stamp programs that save memory and programming space. I can do this, I think, because I understand the guts of the BASIC Stamp. Since most BASIC Stamp users have more trouble managing variables than program space, I'm going to talk about that first.

As you know, the BASIC Stamp supports four native variable types. In order of size they are Word, Byte, Nib and Bit. Of course, the core processor of the BASIC Stamp (PIC or SX) uses bytes as its native variable type; it is Chip's PBASIC interpreter that expands things for the convenience of the BASIC Stamp programmers.

One of the things I find myself reminding Stamp users who claim to be running out of variable space is to use these types appropriately. There is no sense using a byte to store a value that will never exceed 15, right? Of course not. You certainly wouldn't spend the dollar you have in your pocket on a fifty-cent candy bar … you're probably going to want that extra fifty cents later. Treat the BASIC Stamp memory the same way.

This goes along with all that planning stuff I harp on: if you know what a variable is going to do while the program is running you can set its type appropriately. For review and you new folks, here are the sizes and value ranges for each variable type:

Bit	1 bit	0 to 1
Nib	4 bits	0 to 15
Byte	8 bits	0 to 255
Word	16 bits	0 to 65535

Be aware that the BASIC Stamp compiler does not error-check attempts at variable assignments. If you try to assign a nib variable a value that exceeds 15, the value will be truncated. What you'll end up with is the lower four bits of the value you tried to assign. This is not always a bad thing and can be used to our advantage. I'll show you how later.

A big part of the convenience of the BASIC Stamp interpreter is the access it provides to the internals of a given variable. Look at Figure 96.1. This diagram shows how a word is broken down into its constituent parts; the parts that PBASIC gives us access to. If, for example, we didn't have this access, changing the upper byte of a word variable without affecting the lower byte would require code like this:

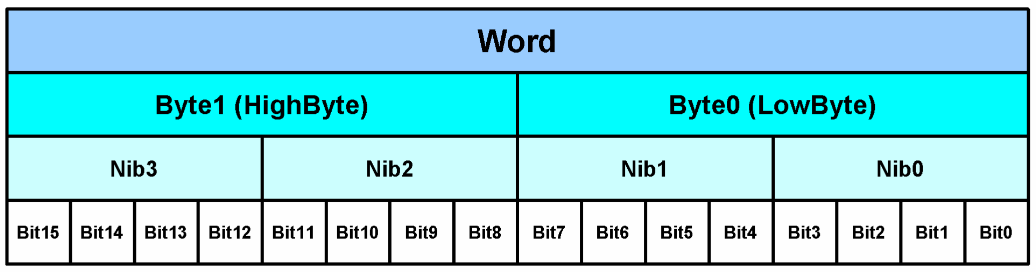
```
  myWord = (myWord & $00FF) | (newUpper << 8)
```

How much easier is it to do this?:

```
  myWord.HIGHBYTE = newUpper
```

I think we can all agree that the latter is the easier to use and understand. Along with ease-of-use, it also consumes less code (EEPROM) space to implement – about half to perform the same function.

Figure 96.1: Word Broken Down into Constituent Parts



The PBASIC Editor/Compiler has a bunch of built-in names (dot modifiers) to access the internals of a given variable. Since more people seem to read this column than will crack open the PBASIC manual ... I'm providing a table – adapted from the manual – to show you.

Symbol	Definitions
LOWBYTE	Low byte of a Word
HIGHBYTE	High byte of a Word
BYTE0	Low byte of a Word
BYTE1	High byte of a Word
LOWNIB	Low nibble of byte or word
HIGHNIB	High nibble of byte or word
NIB0	Low nibble of byte or word
NIB1	High nibble of byte; nibble 1 of word
NIB2	Nibble 2 of word
NIB3	Nibble 3 of word
LOWBIT	Low bit of nibble, byte or word
HIGHBIT	High bit of nibble, byte or wordv
BIT0	Bit 0 of nibble, byte or word
BIT1	Bit 1 of nibble, byte or word
BIT2	Bit 2 of nibble, byte or word
BIT3	Bit 3 of nibble, byte or word
BIT4 .. BIT7	Bit 4 through 7 of byte or word
BIT8 .. BIT15	Bit 8 through 15 of word

That's lots of access and what it means is lots of convenience when writing our programs.

What makes PBASIC even more flexible is that everything within the memory can be treated like an array element – without specifically defining an array. This is probably the trickiest aspect of PBASIC memory management to grasp. Once you do, though, you'll find lots of ways to take advantage of it to trim your code and do things not possible with other versions of BASIC.

First things first. When the compiler is sorting through your variable declarations, it starts with the words and maps them into the Stamp memory in the order of their declaration. Then it maps the bytes, then nibs, and finally, the bits. This strategy allows the compiler to make the most of limited resources.

When I say mapped, what I mean is that each variable is assigned a physical position in the Stamp's memory. We can use aliasing to reuse a previously assigned space. For example, the following definitions actually occupy a single byte of memory space (use the Memory Map function of the editor to see for yourself).

```
timer           VAR     Byte
secs            VAR     timer
secs01          VAR     secs.LOWNIB
secs10          VAR     secs.HIGHNIB
```

When we alias a new name to an existing variable, the size of the new variable will match the base declaration. In the example above, secs will be a byte since we aliased it to timer which is declared as a byte. Likewise, secs01 and secs10 will be nibs. Remember my warning above about attempting to assign out-of-range values to variables, especially aliased variables

Aliasing can also help us reduce code space. Let's expand out definition from above.

```
clock           VAR     Word
hours           VAR     clock.HIGHBYTE
mins            VAR     clock.LOWBYTE
hrs10           VAR     hours.HIGHNIB
hrs01           VAR     hours.LOWNIB
mins10          VAR     mins.HIGHNIB
mins01          VAR     mins.LOWNIB
```

Again, despite all these definitions, we've only consumed one word of memory space. In actual application, we may not really use the clock variable as a whole, but defining it as we have gives us a convenient method of setting the clock with a single line of code. This technique saves us a bit of typing and a few bytes of program space in our EEPROM.

```
  clock = $1235
```

Note that by using hex, all the digits get put into the correct places. Most RTC chips use BCD registers, so these definitions are useful.

Let's go back and see how we can use arrays without actually defining them. Enter and run this little program:

```
bigVal          VAR     Word
digits          VAR     bigVal.Nib0
idx             VAR     Nib

Main:
  bigVal = $1234
  FOR idx = 0 TO 3
    DEBUG DEC digits(idx), CR
  NEXT
  END
```

Before you click the run button, how much memory was used? What will be the output from the program? Okay, go ahead.

Pretty interesting, right? Let me show you an example that's a bit more practical.

```
colors          VAR     Byte
red             VAR     colors
green           VAR     Byte
blue            VAR     Byte
```

What we're doing here is using aliasing to place red in the same location as colors, and taking advantage of the compiler's behavior of assigning locations in the order they appear in our code [for the same type]. What we can do, then, is access items as an array, like this:

```
  for idx = 0 TO 2
    colors = 0
  next
```

Or, deal with individual elements to make our code easier to read:

```
  green = 128     ' same as colors(1) = 128
```

I use this assignment trick quite frequently; look for it in my programs. The reason it's done this was is that the BASIC Stamp compiler WON'T let you do is this:

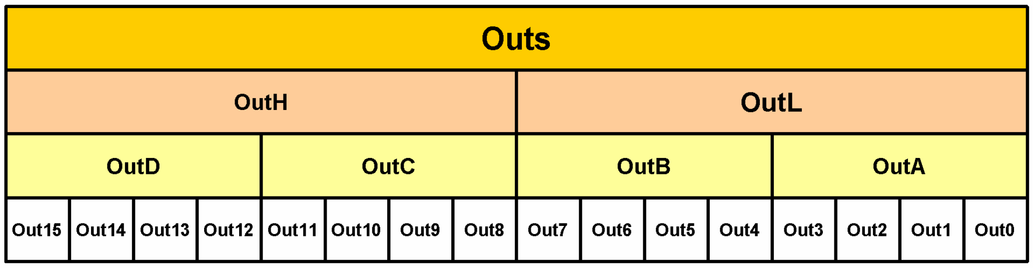
```
colors          VAR     Byte(3)
red             VAR     colors(0)    ' can't do this!
green           VAR     colors(1)
blue            VAR     colors(2)
```

Hardware Control Through Variable Manipulation

I've spent a lot of time discussing the manipulation of variables. One of the less obvious reasons, especially for those new to the BASIC Stamp, is that the PIC/SX (core microcontroller of the BASIC Stamp) architecture is such that manipulating special variables allows us to affect the IO pins. We can use what we know about variable manipulation to affect devices connected to the BASIC Stamp.

The I/O pins of the BASIC Stamp are connected to an internal variable called Outs; these pins are enabled as outputs when the associated bit in the register called Dirs is set to 1. Since there are 16 pins on the BASIC Stamp, Outs and Dirs are a word variables. Figure 96.2 shows how Outs is divided up into its various pieces. All of these pieces are names known byte the compiler so we can use them directly in our code. By using aliasing as we discussed above, we can make our programs easy to read.

Figure 96.2: OUTS is Divided Up Into Memory Pieces



Let's say, for example, that we wanted to create a two-digit counter with a couple 7-segment LEDs. By direct connection, we'd have to use 14 BASIC Stamp pins – maybe that's too many for our application. Okay, so we run over to the electronics store and pickup a couple 7447A display drivers. With the 7447A we can use just four lines to drive each display. Okay, we're down to eight I/O lines for both digits and that will work.

Here's where some planning comes in. If we're clever – and we are – we will connect the drivers at the nibble boundaries of the Outs registers. This works really nicely since the drivers take BCD (nibble, 0 – 9) input, so all we have to do is connect Bit0 of an output nibble to input A (low bit) of the 7447A, Bit1 to input B, etc. With this simple hardware planning, look how easy it is to display digits.

First, the variable declarations:

```
counter         VAR     Byte
count01         VAR     OutA
count10         VAR     OutB
```

Remember, we have to enable the outputs by setting the associated bits in Dirs to 1:

```
Setup:
  DirL = %11111111
```

Now we can write a bit of code to update the displays based on the current value of our counter:

```
Update_Display:
  count10 = counter DIG 1    ' update 10's digit
  count01 = counter DIG 0    ' update 1's digit
  RETURN
```

Does it get any easier than that? I don't think so. But then, we made the code easy by planning our hardware and working with the architecture of the BASIC Stamp and the facilities provided to us by PBASIC.

Remember the friend I talked about earlier? As it turned out, his drivers are a bit esoteric and don't use BCD – he needed to provide specific patterns to his drivers. At first blush, we might be attempted to use a LOOKUP table to move custom patterns to the output. That works, but the technique I share with him offered more flexibility as he added more digits.

What I did was create a DATA table with the custom digit patterns for his driver. The first pattern creates a zero, the next a one, and so on.

```
DigitMap        DATA    %0010, %1010, %1000, %0000, %1110
                DATA    %1100, %0100, %0001, %1001, %0110
```

Assuming a two-digit output as above, we can move the patterns to the drivers with this bit of code:

```
Update_Display:
  READ DigitMap + (counter DIG 1), count10
  READ DigitMap + (counter DIG 0), count01
  RETURN
```

In this case READ is used to move the pattern from the DATA table to the outputs – all done by aliasing the output nibbles. The value for each digit actually provides an offset into the table where the digit patterns are stored – this is why we carefully assigned the patterns in digit order (0 – 9). As it turned out, my version of the code took less than one-fourth the space of the code written by my friend. Generally speaking, smaller code is faster and easier to trouble-shoot.

Remember what I said about variables being truncated when you try to assign a bigger variable to a smaller? Here it is in action. The READ function will assign a byte to the output variable. Since the output variables above are nibs, we only get the lower four bits of the byte read from the DATA table.

Now, things won't always be easy, there will be times when we're stuck with less than ideal design parameters – like someone else designing the hardware and asking us to code for it. This will happen and we just do the best we can.

But given the opportunity to design from scratch, we can save ourselves a lot of troubleshooting time by planning and DESIGNING our project. We often get in a hurry though, and skip right past the planning step. This almost always leads to trouble. I remember a poster in the engineering department of a former employer that stated: "If you don't have time to do it right, when will you find time to do it over?" Amen to that. Plan your work, work your plan, and have a lot of fun along the way!

I guess the point of this article is to say that you'll get more out of your BASIC Stamp programs when you take a bit of time to really learn about the machine that PBASIC runs on. Remember, there's lots of free documentation that can be download from Parallax. Download it and then read it. Hopefully, what we've talked about here will get you started and help you create smaller, cleaner, more efficient programs for your projects.

Until next time, Happy Stamping.