Column #93 January 2003 by Jon Williams:

PBASIC Gets a Make-Over

The phrase "something old, something new..." is usually associated with wedding ceremonies, but is a perfect description of the BASIC Stamp and the Version 2.0 editor combination: the same "old" (I write that lovingly) BASIC Stamp – that has been working reliably for years – combined with a brand new editor that includes some fantastic updates to the PBASIC programming language.
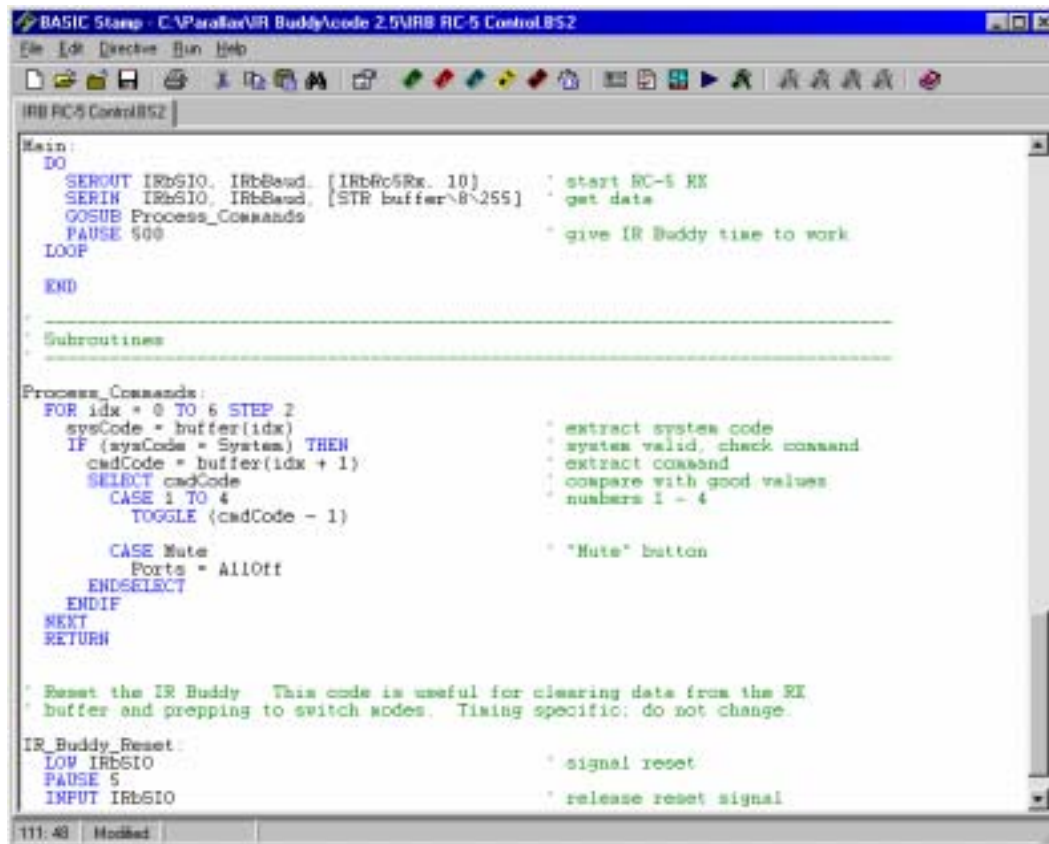
The phrase "something old, something new..." is usually associated with wedding ceremonies, but is a perfect description of the BASIC Stamp and the Version 2.0 editor combination: the same "old" (I write that lovingly) BASIC Stamp – that has been working reliably for years – combined with a brand new editor that includes some fantastic updates to the PBASIC programming language.

I've been programming BASIC Stamps for almost nine years now and I can say without hesitation that the last couple of months have been the most fun. The new PBASIC editor actually makes the BASIC Stamp seem like a brand new microcontroller. Have you ever owned an old car that you loved so much that you had it painted instead of trading it in and suddenly, that old car seems like it was brand-new? That's how the BASIC Stamp feels now – even the stock BS2 – using the new

editor. It genuinely feels like a brand-new machine. The best part is that this "new" feeling comes absolutely free!

So what, exactly, is new? In a nutshell, PBASIC has been "normalized" to include BASIC language syntax that most programmers consider standard. Things like IF-THEN-ELSE, DO-LOOP, and SELECT-CASE. The editor also includes color syntax highlighting that makes editing and debugging PBASIC programs a lot easier.

Figure 93.1: BASIC Stamp Windows Editor Colored Syntax Highlighting

What Took You Guys So Long?

Now, critics will [rightfully] ask: "What took you guys so long? – people have been doing this stuff for years..." The criticism is not entirely unfair and there is no excuse for the delay; especially since customers have asked for these updates for quite some time. There is a valid reason, though.

Keep in mind that the BASIC Stamp was developed in the early 90's, back when the Intel '386 was the king of the microprocessor hill and PC speeds were measured in the low tens of megahertz. Chip Gracey, the BASIC Stamp designer, decided to write the original development tools using Intel 80x86 assembly-language to get the best performance on the machines of the time.

Even though he's considered one of the world's best assembly-language programmers, Chip will freely admit that the BS2 tokenizer (the code that converts PBASIC to the EEPROM image that is actually downloaded into the Stamp) was a significant challenge – and a good reason why there are a few BASIC Stamp kind-of-work-alike products, but no actual third-party BASIC Stamp clone that works with the Parallax editor/compiler. You see, the Parallax tokenizer is more than a compiler; it actually compresses the tokens so that the 2K EEPROM on the BS2 can hold more than 2K worth of instructions and data. It's more than a bit tricky, to say the least.

Once it was all working, you can imagine that no one was anxious to tear into the tokenizer for more than updates as were required to support new BASIC Stamps. When Parallax decided to create tools for Windows, the assembly-language tokenizer was simply linked to the Windows editor, which worked because Windows and the tokenizer were written for Intel platforms. Ultimately, however, the popularity of non-WIntel operating systems and platfoms and many BASIC Stamp users' desires to have native tools for these platforms won out.

So, a little over a year ago, Jeff Martin of Parallax began the arduous process of converting the assembly-language tokenizer to C (for portability to virtually any OS). Testing the new tokenizer took nearly as long as writing it. And testing is something that Parallax is very cautious about, especially when it comes to anything that affects the millions of BASIC Stamps that have already been sold. This process took quite a while because literally thousands of BASIC Stamp programs were compiled and the output was compared with the existing assembly-language tokenizer to ensure a perfect byte-for-byte match. Once the new tokenizer was fully tested and approved, it was folded into the editor and has been there since version 1.32. The tokenizer is also available – pre-compiled only – for Windows, Mac and Linux developers.

And now for the good stuff. With the tokenizer much easier to update, making changes to it and the way it handles PBASIC language parsing becomes simpler to implement and to test. And that's what gets us to where we are today. Based on customer requests and using other versions of

BASIC as models, PBASIC has been updated. The nice thing is that if you do nothing with your old code, it will compile in the new editor as is. If you choose to use the new language features, you just tell the editor that's what you're doing and away you go.

PBASIC 2.5

Since the language update affects only the BS2 family, it is being labeled PBASIC 2.5. To tell the editor that you're writing using this syntax, you'll add a compiler switch at the beginning of your code:

```
' {$PBASIC 2.5}
```

This is similar in format to the $Stamp directive used to identify your target hardware and must be on its own line. There's also a new toolbar button that will drop this switch in for you.

Probably the most requested update to PBASIC has been the inclusion of IF-THEN-ELSE. In classic PBASIC, the IF-THEN construct is identical to the assembly code that it calls and takes this syntax:

```
IF (condition) THEN label
```

This syntax is still valid and does work under PBASIC 2.5. But what BASIC Stamp customers have long asked for and now they can use is this:

```
IF (condition) THEN
   ' statement(s)
ELSE
   ' statement(s)
ENDIF
```

For those with a background in other versions of BASIC, this looks quite normal – with the possible exception of the ENDIF keyword.   ENDIF was selected over "END IF" (as in Visual BASIC) to simplify the tokenizer parser and keep things efficient.

There is a bit of short-cutting the programmer can do, depending on the complexity of your code. For example:

```
IF (condition) THEN
   GOSUB label
ENDIF
```

can be simplified to:

```
IF (condition) THEN GOSUB label
```

While not frequently used, PBASIC does allow multiple statements on the same line. I don't generally advocate using more than one statement per line, the example above being an exception that I do use in my own code. If you have just one statement per block, you can put IF-THEN-ELSE on a single line like this:

```
IF (condition) THEN statement ELSE statement
```

Note that ENDIF is not used when IF-THEN-ELSE is on a single line. Finally, while not recommended, you can put multiple statements per block on one line by separating the statements with colons.

```
IF (condition) THEN statement1 : statement2 ELSE statement
```

I tend not to use this syntax style because it leads to long lines that can become error-prone when editing.

And while we're on the subject of long lines, another frequent request by BASIC Stamp users is the ability to split a single long line across two or more shorter lines. This is now possible – so long as you've got a list on the line that is separated by commas. If this is the case, you can separate the long line at the comma character. Here's an example:

```
BRANCH value, [Target1, Target2, Target3,
               Target4, Target5, Target6]
```

Breaking lines at commas also works with DEBUG, SERIN, SEROUT, LOOKUP, LOOKDOWN – anywhere you have a comma-separated list of items.

Back to language updates. I never asked Chip, but I'm pretty sure that BRANCH was inspired by the old GWBASIC/BASICA syntax:

```
ON value GOTO Target1, Target2, Target3, ...
```

For those of you that have used those "older" PC versions of BASIC, you'll also remember this:

```
ON value GOSUB Sub1, Sub2, Sub3, ...
```

Good news: both now exist in PBASIC. The latter is particularly useful in the style that I like to write programs for robotics. My preference is to have a main control loop that calls small routines

and can change the order that these routines run based on current conditions (sensor input, etc.).
In the past I used BRANCH, but this meant that all of my code blocks had to have a GOTO back
to a single location; a location that could change, which meant multiple edits. By changing
BRANCH to ON value GOSUB this is no longer a requirement as the RETURN at the end of my
subroutines will handle getting them back to the right place and also makes them callable from
other locations in the program.

This is really good stuff. Let's keep going!

Another new language feature for PBASIC is DO-LOOP. It's general form is defined a couple of
ways. Here's the first:

```
DO
   ' statement(s)
LOOP
```

This form is an infinite loop that will run until a GOTO or EXIT (new keyword) breaks us out.

```
DO
   idx = idx + 1
   IF (idx >= 10) THEN EXIT
LOOP
```

Note that EXIT will cause the program to jump to the line that follows LOOP. While valid,
another – arguably better -- way to write this loop is like this:

```
DO
  idx = idx + 1
LOOP UNTIL (idx >= 10)
```

If we look at the first two examples, the value of idx is incremented before testing and the loop code runs at least once. We can check the limit before running the loop code if we rewrite the it like this:

```
DO WHILE (idx <= 10)
  idx = idx + 1
LOOP
```

Another thing that I should point out is that UNTIL and WHILE can generally be interchanged, though most programmers will use them as I've demonstrated above.

The last major language update is the addition of SELECT-CASE. In PBASIC, its implementation is like this:

```
SELECT expression
  CASE condition(s)
    ' statement(s)

  CASE condition(s)
    ' statement(s)

  CASE ELSE
    ' statement(s)
ENDSELECT
```

The expression portion is a variable, constant or expression. The condition part of each CASE statement is very flexible, as in other versions of BASIC – but with some of the additional flexibility that we're accustomed to in Parallax BASIC.

Here's how the CASE syntax works:

```
CASE conditional_op expression
```

Where conditional_op can be = (implied), <>, <, >, >= or <=. Multiple conditions within the same CASE can be separated by commas. If, for example, you wanted to run a CASE block based on a value being less than five or greater than ten, the syntax would look like this:

```
CASE <5, >10
```

Here's another way to implement CASE:

```
  CASE value1 TO value2
```

In this use, the valid range is from value1 to value2, inclusive. Let's take a look at a bit of actual working code. This subroutine is lifted from an update of the device controller example using the Parallax IR Buddy. In this code, one of four outputs will be toggled if the corresponding numeric key is pressed on the a compatible TV remote. If the Mute button is pressed, all outputs are turned off.

```
Process_Commands:
  FOR idx = 0 TO 6 STEP 2
    sysCode = buffer(idx)
    IF (sysCode = System) THEN
      cmdCode = buffer(idx + 1)
      SELECT cmdCode
        CASE 1 TO 4
          TOGGLE (cmdCode - 1)

        CASE Mute
          Ports = AllOff
      ENDSELECT
    ENDIF
  NEXT
  RETURN
```

Notice the use of CASE 1 TO 4 for the decoding of numeric keys from the remote and the implied equality when handling the Mute button value.

For those that come from a C or Java background, you know that in those languages the statements in CASE block fall through to the next CASE block unless the keyword break is encountered. In BASIC, the code under an executed CASE block jumps to after ENDSELECT. In PBASIC 2.5, we have the option to make it drop through to the next CASE block for condition testing if we choose.

In order to make CASE code drop through to the next CASE block, we will change the subsequent CASE to TCASE (through CASE). After execution of the statements within the TCASE block, the program jumps to after ENDSELECT, unless followed by another TCASE.

A New PBASIC, A New Type

A big part of the BASIC Stamp's success is due to the variable type flexibility: Bit, Nibble, Byte and Word – I'm often wishing the PC versions of BASIC (and other languages) allowed bit and nibble variables. With the new language comes a new type definition: PIN. The reason for the PIN type is that some elements of PBASIC expect an I/O pin value in the form of a constant, while other elements require a variable (i.e., In0 or Out13). By using PIN instead of CON, we no longer have to concern ourselves with syntax, the editor adjusts automatically.

Let's look at an example from the IR Buddy. In classic PBASIC, the serial pin was defined like this:

```
IRbSIO  CON     15
```

... which works fine for SEROUT and SERIN. But one of the features of the IR Buddy is that it will indicate that it's busy transmitting by pulling the serial line low. In order to monitor the serial line, we had write code in this style to use the previously-defined pin constant:

```
TX_Wait:
  IF (Ins.LowBit(IRbSIO) = 0) THEN TX_Wait
```

Works, but is a bit clunkly. In PBASIC 2.5, we change the serial pin definition to:

```
IRbSIO  PIN     15
```

... which has no affect on SEROUT and SERIN, yet greatly simplifies the busy check.

```
TX_Wait:
  DO WHILE (IRbSIO = 0) : LOOP
```

The editor is intelligent in its analysis and evaluates the code as:

```
  DO WHILE (In15 = 0) : LOOP
```

If, on the other hand, we had written:

```
  IRbSIO = 1
```

The editor would evaluate this statement as:

```
  Out15 = 1
```

In the end, the PIN type lets us think about I/O pins as I/O pins and the editor correctly handles the syntax details. As with the other improvements, it just makes PBASIC easier and, dare I say, more joyful to use.

Okay, I'm almost out of space but I want to share just one more update. This will really be welcomed by those who use the BASIC Stamp's EEPROM as data storage or table space. WRITE and READ have now been updated to work with 16-bit variables. To make this happen, you put that modifier Word in the code. Like this:

```
WRITE address, Word value
READ address, Word value
```

The approach used is Little-Endian; the low-byte of value is stored at or read from address and the high-byte at or from address+1. Keep this in mind when writing data storage programs that expect 16-bit values in consecutive locations. If you're storing or reading Words, you'll need to increment your EEPROM address pointer by two for your next access. If you leave out the Word modifier, WRITE and READ will work with Bytes as in classic PBASIC.

The new editor with PBASIC 2.5 is scheduled for release toward the middle of this month, so be sure to check the [also new and improved] Parallax web site for details. Let me stress one more time that this is just an editor upgrade – you do NOT need to return your BASIC Stamps for reprogramming. How often does that happen ... a new BASIC Stamp without updating or buying a new BASIC Stamp? Now that's cool.

Happy New Year! And until next time, Happy Stamping.