BASIC Propeller Programming

By Jon Williams

For Nuts & Volts Magazine, Column 5, March 2010

We knew it had to happen, right? After all, Parallax and BASIC go together like macaroni and cheese; on their own the elements are great – when you get them together you create absolute magic!

I think it's safe to say that more than a few Parallax customers were disappointed when the Propeller arrived and couldn't be programmed in BASIC. Like me, many of these customers had invested years learning and working with BASIC, and a good portion of that group had a lot of time with embedded projects thanks to PBASIC. Now, I'm not being critical – Spin is okay with me and I'm quite comfortable with it. That said, I've been programming in one form of BASIC or another for almost 30 years, and that kind of makes BASIC a little bit hard to give up.

Well... good news! The team that developed SX/B (headed by Terry "Bean" Hitt) has taken the lessons learned from that product and created a BASIC language compiler for the Propeller: PropBASIC. So, if you've been holding off getting into the Propeller because you couldn't use BASIC, you have no more excuses.

What is PropBASIC, Anyway?

Beyond the obvious, of course, PropBASIC is a single-pass compiler that generates Propeller Assembly code so that our programs run at the fastest possible speed. The output is one or more Spin files that can be downloaded to the Propeller. In version 1.0 the main program code is limited to a single cog. Don't let this bother you – Propeller Assembly is very powerful and processes that take several Assembly instructions in other micros are handled with a single instruction in the Propeller. To top that, we have seven additional cogs available and PropBASIC lets us use them with tasks. In the future it is very likely that PropBASIC will adopt [Propeller-whiz] Bill Henning's LMM (large memory model) architecture, allowing the compiled output to run from the Hub RAM, breaking the 2K limit imposed by a cog.

For those that have used SX/B, PropBASIC will seem very familiar and you should be able to migrate many of your projects with just a few changes. If you're coming straight from the BASIC Stamp (or one of its many work-alikes) you'll find PropBASIC similar to PBASIC, though not directly compatible. Trust me, it's not a problem; the learning curve is very shallow. Many of us, myself included, transitioned from the BASIC Stamp to the SX using SX/B. The transition from PBASIC to PropBASIC will be just as easy, and a lot of fun. Yes, you'll have to get used to a slightly-new way of doing things, but once you do you'll wish you'd made the transition to a multi-core processor sooner!

Hello, PropBASIC

Every PC programming book, no matter the language it teaches, starts out with the now-infamous *"Hello, World"* program. Starting simple is smart as it allows us to get the fundamentals in place before tackling the big stuff. In the microcontroller world we tend to blink an LED. Trivial? Yes. Important? Yes! – if we can't blink an LED then we certainly can't expect to control a multi-axis robot using GPS input now, can

we? Go ahead and connect an LED circuit as shown in Figure 1. If you have a Propeller Demo Board that LED is in place.

Figure 1

Nothing drives me nuttier than messy program listings – I hate them. An easy way to get a neat listing in the end is to start with a clean template. In the downloads file you'll *find template.pbas* which is what I use to start most projects (I have a few other templates, too, based on different project types). Starting with *template.pbas* I've created *hello.pbas* which we'll work through to introduce the elements of a PropBASIC program. Of course, not all sections of the template are used for a simple LED blinker, but we'll introduce them, anyway, and then explore them later.

Again, if you've worked with SX/B much of this will be very familiar. If you're coming from the BASIC Stamp there will be some new things to get used to; trust me, you can, and when you do so many neat things will open up for your [perhaps postponed] projects.

All right, let's go through it, section-by-section.

Device Settings

```
DEVICE          P8X32A, XTAL1, PLL16X
XIN             5_000_000
```

We'll start nearly every PropBASIC program with this "standard" setup; it declares the Propeller 1 (P8X32A), running with an external crystal, using a PLL multiplier of 16x. The external crystal frequency is 5 MHz which gives us a system frequency of 80 MHz (5 x 16).

There are, of course, options to the standard setup. For low-power applications that are not timing-sensitive we can select one of the RC modes. **RCFAST** runs the chip at about 12 MHz while **RCSLOW** runs the chip at about 20 kHz. To run in **RCFAST** mode we can reduce the setup to this:

```
DEVICE          P8X32A
```

When no mode is specified the compiler assumes **RCFAST**; to use **RCSLOW** we must specify it in the **DEVICE** directive. In RC modes the PLL is always 1x and the **XIN** value is ignored. Again, RC modes should only be used in programs that are not timing-sensitive, as the RC oscillator varies from chip-to-chip. RC modes would not, for example, be good to use when serial communications is a requirement.

That last statement probably caused you sharp folks to raise a Spock-like eyebrow.... If RC modes are no good for serial communications then how can the IDE reprogram the Propeller when no crystal is attached? Good question. The IDE actually times the Propeller as part of the download protocol and then adjusts the baud rate from the IDE to accommodate the individual chip. So, if you really have to use serial communications and need a low clock speed to reduce current consumption, you can do it but you'll need to do a couple extra steps:

- Measure the clock speed in RC mode (with no **FREQ** specified). You can do this by measuring the width of an output pulse using an oscilloscope.

- Override the assumed RC frequency by using the **FREQ** directive instead of **XIN** (which doesn't apply to RC modes, anyway).

Let's say we create a program (see *speed_test.pbas*) to output a 10 ms pulse and on a 'scope we measure – as I did – a pulse width of 9.06 milliseconds. This means that the processor is actually running a little faster than expected. We can calculate a scale factor for the assumed clock frequency by dividing the measured pulse width into the 10 milliseconds we expected.

scale = 10.0 / measured

In my case the scale factor works out to 1.1038. I had selected **RCSLOW** mode which has an assumed frequency of 20 kHz. With the adjustment the top of my program is now:

```
DEVICE          P8X32A, RCSLOW
FREQ            22_075
```

The **FREQ** setting will drive the compiler and in the end we'll have corrected timing for things like **PAUSE**, **SEROUT**, etc. Figure 2 shows the uncorrected output (without using **FREQ**) from my test, Figure 3 is after the **FREQ** setting has been added.

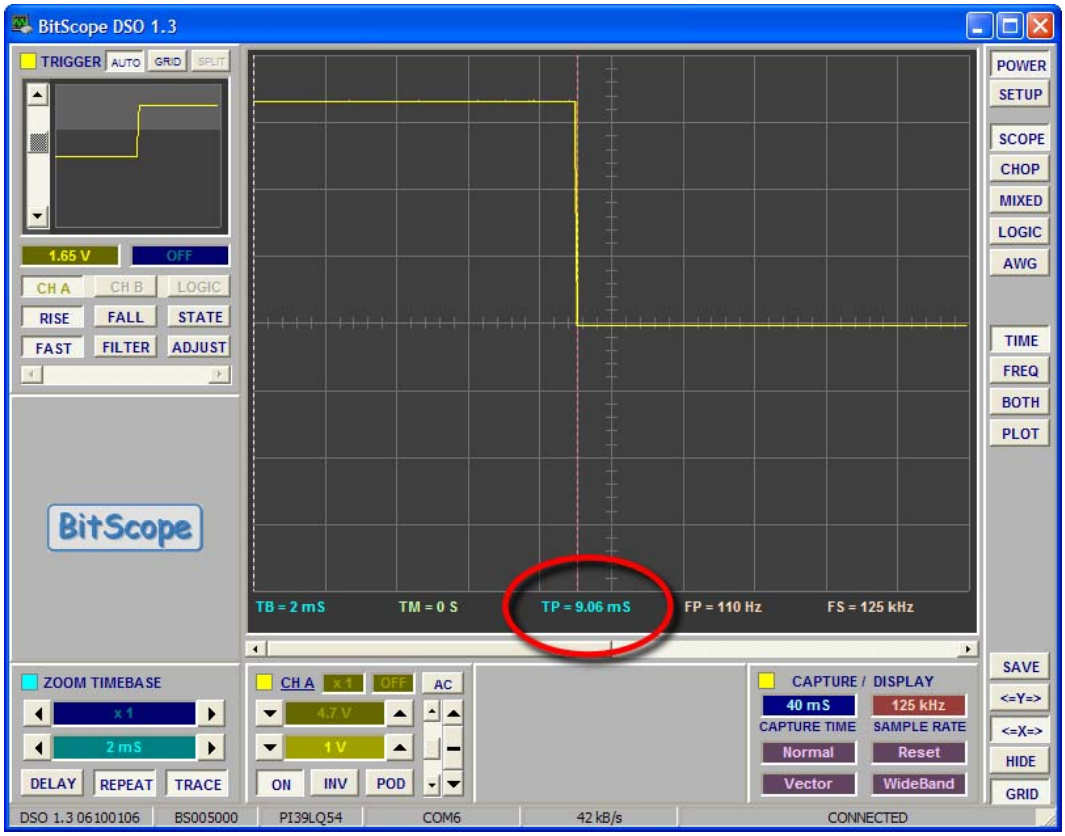

Figure 2

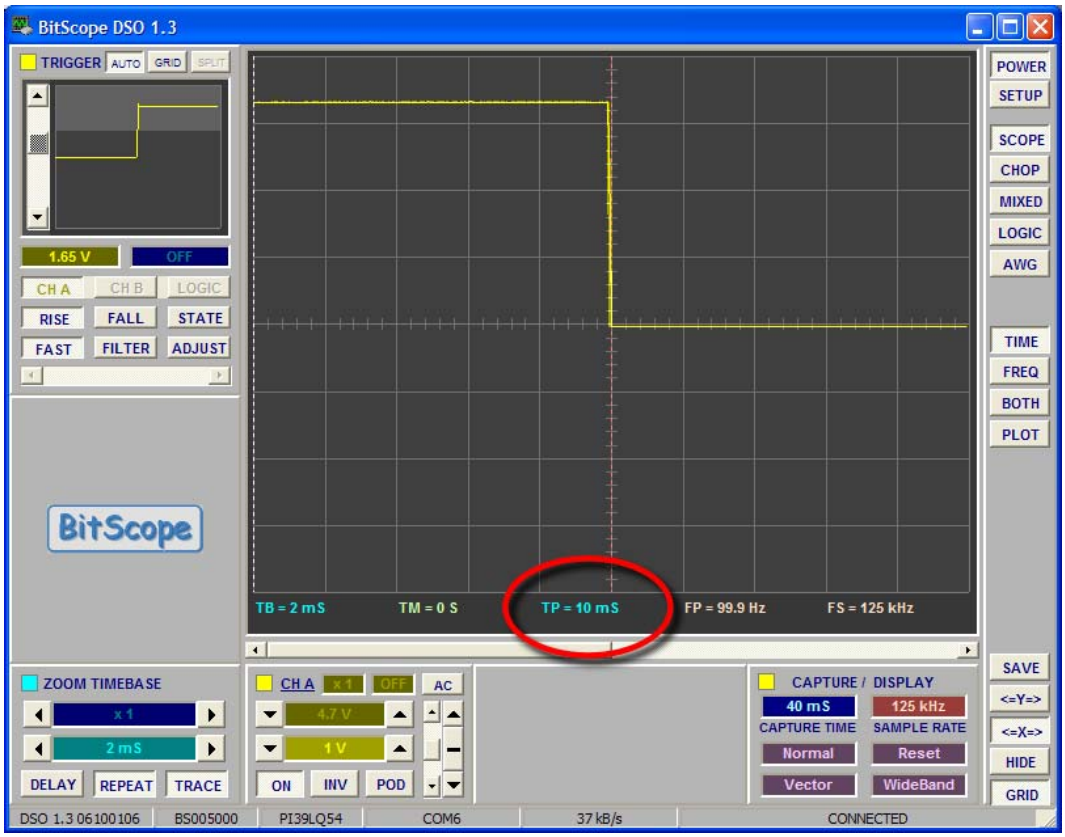

Figure 3

Program Constants

The next section of the template involves defining constants. There's no mystery here with standard numeric and single-character constants.

```
OnTime          CON     250
OffTime         CON     750
IsOn            CON     1
IsOff           CON     0
Asterisk        CON     "*"
```

If a program will be using **SERIN** or **SEROUT** then the baud mode constant is defined like a string.

```
Baud            CON     "T9600"
```

This definition can be used to drive **SERIN** and **SEROUT** in true mode (standard when using the programming port) at 9600 baud. Mind you I'm just using this as an example – PropBASIC compiles to Assembly and when running at 80 MHz we can have very fast baud rates.

Constants are global in a PropBASIC application and can be used in multiple cogs (i.e., the main program and any tasks that run).

IO Pins

```
LED             PIN     16 LOW
```

Declaring IO pins in PropBASIC is very much like PBASIC 2.5 and SX/B, while using the SX/B convention of allowing the pre-assignment of the pin state immediately after a reset. The options for a pin are **INPUT** (default), **OUTPUT** (same as **LOW**), **LOW**, and **HIGH**. In my programs that will communicate with a terminal program I include these **PIN** definitions

```
RX              PIN     31 INPUT
TX              PIN     30 HIGH
```

Note the use of the **HIGH** option; I do this so that auto-generated start-up code places the TX pin in the idle state (output and high) for serial communications through the Propeller programming port.

PropBASIC allows the definition of a group of IO pins as well. For example, there are eight LEDs on the Propeller Demo Board, connected to P16 through P23. We can define these as a group using the following declaration:

```
LEDs            PIN     23..16  LOW
```

In a pin group definition the first number is the MSB pin of the group, the second is the LSB pin of the group. The order is important as it affects how bits will be presented at outputs or read when the group is set as inputs.

In PropBASIC 1.0 we can only assign a simple value to an output pin group. That means that this line:

```
  LEDs = %100000
```

...is legal, but...

```
  LEDS = 1 << position
```

...is not. This is likely to be addressed in a future version. In the mean time we can use one of PropBASIC's temporary variables like this:

```
  __temp1 = 1 << position
  LEDs = __temp
```

Shared Variables

PropBASIC allows us to store variables and variable arrays in the Hub that can be accessed by any cog. Hub variables may be defined as bytes, words, or longs, or arrays of either type.

```
rxHead          HUB     Byte = 0
rxTail          HUB     Byte = 0
rxBuffer        HUB     Byte(64) = 0
```

Note that PropBASIC allows us to pre-assign values to Hub variables. If the pre-assignment option is not used the variable (and all elements of an array) will be set to zero. Though they are variables, what we store in the Hub cannot be used in expressions like regular variables. There are two instructions, **RDxxxx** and **WRxxxx**, that are used to read and write Hub variables (the instruction used will be based on the variable size, you'll see examples in a moment).

Shared Data

Along with variables we can store shared data in the Hub as well. With 32K of RAM this allows for very large tables, and being in the Hub these tables can be accessed from any cog. Hub data can be declared as bytes, words, or longs using, respectively, the **DATA**, **WDATA**, and **LDATA** directives.

```
Zip4            DATA    %0001
                DATA    %0010
                DATA    %0100
                DATA    %1000
```

As with Hub variables, we can access Hub data using the RDxxxx and WRxxxx instructions. For very large tables/data we can incorporate external data using the FILE directive.

```
SFX1            FILE    "BABY.WAV"
SFX1x           DATA    0
```

Defining Tasks

In order for a PropBASIC program to use two or more cogs we must define **TASK**s to handle the "background" processes. If you've ever programmed the BASIC Stamp you know how frustrating it is to have to wait around for a serial byte to come in. In SX/B we could use interrupts to create a background serial buffer, but that could be tedious, and many didn't try.

Well, in the Propeller we can simply spawn the physical serial input to another processor using Hub variables to hold everything. The **TASK** definition is simple but needs to be in place before the task can be started.

```
SERIAL_RX       TASK
```

Local (Cog) Variables

The variables we use in our program and in tasks are local to the cog and can only be defined as Longs. As with Hub variables, we can also declare arrays and pre-assign values. Here's a typical example:

```
idx             VAR     Long
```

PropBASIC creates several local variables for use in the Assembly code generated by the compiler, as well as for passing parameters back-and-forth to subroutines and functions. In the compiled output you'll find five variables, *__temp1* through *__temp5* that are used in the code generated for PropBASIC keywords. We can use these variables, but with some caution, as they may change if a keyword is used between accesses. I tend to favor using the *__param1* through *__param4* variables when I can. These are only used when passing parameters and do not get modified by the code generated for PropBASIC keywords.

Declaring Subroutines and Functions

I stated earlier that PropBASIC is a single-pass compiler, which means it makes no attempt to optimize the output code. The good news is that we can learn Propeller Assembly tricks by examining the output and minimize code space by using custom subroutines and functions.

In the early days of SX/B (and I'm betting it will happen in PropBASIC, too), some programmers ran themselves out of code space writing seemingly innocuous code. The **PAUSE** instruction, for example, is used in most programs. Let's say we use **PAUSE 100**. The PASM output for that instruction is this:

```
        mov     __temp1, cnt
        adds    __temp1, _1mSec
        mov     __temp2, #100
```

```
__L001
        waitcnt __temp1, _1mSec
        djnz    __temp2, #__L001
```

Okay, it doesn't look like much but if that code is generated for every appearance of **PAUSE** we can chew through the cog code space pretty quickly. This is easily overcome by encapsulating **PAUSE** in a subroutine. As I did in SX/B, my shell for **PAUSE** is called ***DELAY_MS***.

```
DELAY_MS        SUB     1
```

We only need to pass one parameter because the range of the 32-bit values of the Propeller variables allow for very long delays (two billion plus milliseconds is a *looonng* time).

For those of you coming from SX/B there is a slight difference in the **FUNC** declaration. In January, after SIRCS project, a Propeller forum member asked for a translation of that functionality to PropBASIC. You may remember that my SIRCS object could return the IR code bits as well as the bit count. PropBASIC doesn't care how many parameters are coming back from a function (typically just one), only how many parameters are passed to it, hence the definition is:

```
SIRCS_RX        FUNC    0
```

The Main Program

The body of a simple LED blinker program might look something like this:

```
PROGRAM Start

Start:
  FOR idx = 1 TO 3
    LED = IsOn
    DELAY_MS OnTime
    LED = IsOff
    DELAY_MS OffTime
  NEXT
  DELAY_MS 1_000
  GOTO Start
```

Let me explain the **PROGRAM** directive. In addition to indicating the starting point of our code this directive provides the location for the auto-generated start-up code. In the Propeller the start-up code is very simple, setting the IO pin directions and states before jumping to the label indicated in the directive.

As you can see, the code is pure BASIC, every bit as easy to understand PBASIC, SX/B, or any other variant we've ever dealt with.

Fleshing Out Subroutines and Functions

At the end of the listing is where we'll place the working code for our subroutines and functions. For the ***DELAY_MS*** subroutine we defined earlier the code is this simple:

```
SUB DELAY_MS
  PAUSE __param1
  ENDSUB
```

As in any other language, one subroutine can call another. In my programs that need to send data to a terminal you'll find these two subroutines:

```
SUB TX_STR

  strAddr       VAR     __param2
  strChar       VAR     __param3

  strAddr = __param1

  DO
    RDBYTE strAddr, strChar
    IF strChar = 0 THEN EXIT
    TX_BYTE strChar
    INC strAddr
  LOOP
  ENDSUB

SUB TX_BYTE
  SEROUT TX, Baud, __param1
  ENDSUB
```

The second subroutine is simply a shell for **SEROUT**. The first is used to send a string to the serial port. When calling ***TX_STR*** the address of the string is passed in *__param1* and captured by the routine. Since we'll need *__param1* to send a character to ***TX_BYTE*** we use *__param2* and *__param3* for internal work.

The reason I don't just declare new variables is that everything exists in RAM when using the Propeller, so it's a good habit to minimize variable declarations by using the *__paramx* variables when possible. A new variable that we don't declare frees up space for an Assembly instruction.

You probably noticed that the code uses **RDBYTE** to retrieve a character from the string. PropBASIC stores all strings, even those we declare inline, in the Hub RAM to conserve cog space. Of course, we create strings manually using the **DATA** directive:

```
Banner          DATA    "PropBASIC!", 0
```

Function code often looks just like subroutine code except that it is enclosed in a **FUNC..ENDFUNC** block and uses **RETURN** just before **ENDFUNC** to send one or more parameters back to the caller. Here's the PropBASIC version of the SIRCS receiver function that I mentioned earlier.

```
FUNC GET_SIRCS

  irCode        VAR     __param1
  irBits        VAR     __param2

  COUNTERA NEG_DETECT, IR, 0, 1
  COUNTERB FREE_RUN, 0, 0, 1

Wait_Start:
  WAITPEQ IR, IR
  PHSA = 0
  WAITPNE IR, IR
  PHSB = 0
  WAITPEQ IR, IR
  IF PHSA < BIT_S THEN Wait_Start

  irCode = 0
  irBits = 0
```

```
Check_Frame:
  IF PHSB > MS_044 THEN IR_Done

Wait_Bit:
  IF IR = 1 THEN Check_Frame
  PHSA = 0
  WAITPEQ IR, IR
  irCode = irCode >> 1

Measure_Bit:
  IF PHSA > BIT_1 THEN
    irCode = irCode | $8000_0000
  ENDIF
  INC irBits
  IF irBits = 20 THEN IR_Done
  GOTO Check_Frame

IR_Done:
  __temp1 = 32 - irBits
  irCode = irCode >> __temp1

  RETURN irCode, irBits
  ENDFUNC
```

I think you'll agree that this is pretty straightforward, and as it is ultimately compiled to pure Assembly code it runs full speed. So... if you've been wanting to experiment with other IR protocols, perhaps RC-5, now you have some high-level code to start with (see sircs_rx.pbas)

I should point out, too, that PropBASIC includes some Spin and PASM keywords. In the SIRCS example you can see that I'm using **WAITPEQ** and **WAITPNE** just as in PASM.

Fleshing Out Tasks

Since a task actually runs in a separate cog its construction is a little more involved than a subroutine or function. In fact, as a task is its own program we can – and sometimes must – declare subroutines and functions that run only within the task. All of those declarations and code elements will exist within the **TASK..ENDTASK** block.

Let's create that "background" serial input task that I suggested earlier. What we want to do is have a cog monitor the RX line and when something comes in write that byte to a circular buffer that can be accessed by another cog. To manage the buffer we need two variables: a head pointer which is the next open position in the buffer to write, and a tail pointer which is the next position in the buffer to read. When the buffer is empty the head and tail will be equal.

Here's the code for that task:

```
TASK SERIAL_RX

  rxb           VAR     __param1
  hPntr         VAR     __param2

  DO
    SERIN RX, Baud, rxb
    RDBYTE rxHead, hPntr
    WRBYTE rxBuffer(hPntr), rxb
    INC hPntr
```

```
    hPntr = hPntr & $3F
    WRBYTE rxHead, hPntr
  LOOP

  ENDTASK
```

Yep, that's the whole thing. Remember, tasks make use of **CON**, **PIN**, and **HUB** declarations, and we have all of those elements here. At the top of the loop we wait for a byte using **SERIN** – just as we've done in PBASIC and SX/B. The difference here is that the task will be launched into its own cog and waiting for a serial byte in that other cog won't block the main program (unless we let it).

After a byte arrives we retrieve the current value of the head pointer using **RDBYTE** and then use it as the offset into *rxBuffer* so that we can store the new serial input using **WRBYTE**. The local copy of the head pointer is incremented and ANDed with $3F to keep it within the legal range of the buffer. This value is then written back to the Hub for use by the caller.

There are a couple ways to start a task, though we normally use **COGSTART**.

```
  COGSTART SERIAL_RX
```

That's all it takes. We may want to wait a few milliseconds before attempting to access data provided by the task; this allows plenty of time for the other cog to get up and running.

One final note on the **TASK..ENDTASK** block. The PropBASIC compiler will create a Spin file with the name of the task. This means we have to be a little careful with naming tasks so that we don't overwrite regular Spin programs that may live in the same folder.

Okay, so now we have a means of receiving and buffering serial bytes, how do we use them in our main program? What we'll do is create a function that retrieves a byte from the buffer.

```
FUNC RX_BYTE

  rxh           VAR     __param1
  rxt           VAR     __param2
  rxchar        VAR     __param3

  DO
    RDBYTE rxHead, rxh
    RDBYTE rxTail, rxt
  LOOP UNTIL rxh <> rxt

  RDBYTE rxBuffer(rxt), rxchar
  INC rxt
  rxt = rxt & $3F
  WRBYTE rxTail, rxt
  RETURN rxchar
  ENDFUNC
```

The upper loop retrieves and compares the values of the head and tail pointers; when these values are equal the buffer is empty. As soon as they differ we can use the tail pointer as an index into the buffer and grab a byte from it. And just as we did with the head pointer we update the tail pointer and save it back to the Hub.

Since we took the trouble to write a background serial buffer let's free ourselves from not being blocked when the buffer is empty. Here's a function that will return the number of bytes waiting in the serial buffer;

if zero the buffer is empty and we can skip calling ***RX_BYTE*** which would cause us to wait for something to arrive.

```
FUNC RX_CHECK

  head          VAR     __param1
  tail          VAR     __param2
  bufcnt        VAR     __param3

  RDBYTE rxHead, head
  RDBYTE rxTail, tail
  IF head >= tail THEN
    bufcnt = head - tail
  ELSE
    bufcnt = tail - head
  ENDIF
  RETURN bufcnt
  ENDFUNC
```

This should be pretty obvious; we're returning the difference between the head and tail buffers. The buffer is circular and the tail chases the head, so we use **IF..THEN** to prevent returning a negative value when the head pointer has wrapped around past zero and is less than the tail pointer.

Assembly, Anyone?

Most of us enjoy compilers because they make writing programs faster and keep us from the nitty-gritty details of Assembly. Still, there are times when using Assembly is helpful. In the PropBASIC we can include inline Assembly using an **ASM..ENDASM** block, or one line at a time using ****. PropBASIC generates very nice code so we won't need to use inline Assembly often, but it's comforting to know that we can if we choose to do so.

Let's Wrap It Up

In one issue I can't cover every aspect of PropBASIC, but I think by now you have an idea of what it is and how to get started with it. As with its SX/B predecessor, PropBASIC includes conditional compilation directives, and the ability to include external assembly and PropBASIC files. It really does have a lot of muscle for a first-generation product – and you can't beat the price ($0).

The Future of PropBASIC

In a word: exciting. For those of us that use SX/B we'll tell you that it started out with very humble beginnings and grew into an extremely nice tool. PropBASIC 1.0 is light years ahead of SX/B 1.0 in terms of capability and will just get better. It's natural that the language will expand as we all spend more time with it and offer suggestions to Bean, and the migration to LMM will let us create very large programs, yet still running at the speed of Assembly.

So... if you've been waiting for BASIC to play with the Propeller, your wait is over. No more excuses because the compiler is free. Come on, jump on – the ride will be fun. I promise.

Until next time, have fun and keep *spinning* and *winning* with the Propeller and PropBASIC.

Resources

Jon Williams
jwilliams@efx-tek.com

Parallax, Inc.
www.parallax.com