Spinning Up Fun With Encoders

By Jon Williams

For Nuts & Volts Magazine, Column 6, May 2010

In my dual life that crosses between the technical and entertainment worlds I have the incredible good fortune to meet some really great people. Case in point: I was contacted by cool cat named Wayne (dubbed "the Brain" by his friends) who, like me, is an electronics enthusiast and embedded programmer, and who also works in "show biz." Wayne's entertainment gig is in music, engineering and mixing songs for some amazing A-List Pop, R&B, and Hip-Hop artists. Wayne needed help a little with an encoder for a personal project. Coincidentally, one of my own customers had asked about adding a local user interface to a product. While working with Wayne, I solved my own problem and I'm going to show you what I came up with so you can put it to use, too. And while we're on the topic of expanding inputs with just a few IO pins I'm also going to show you how to apply an old trick to this new processor.

Gray Code is Black & White

The encoder Wayne selected is called a Gray code encoder; in his particular case it is a two-bit encoder. Gray code is different from regular binary code in that two successive values differ by only one bit – like this for the two-bit encoder we'll be using:

%11 → %01 → %00 → %10 → %11

Note that as we move through the sequence, in either direction, only one bit changes. And yes, this is in fact different from a two-bit binary sequence where we would go from %11 to %00 and two bits would change. Why is all this so important? Well, despite best efforts in manufacturing, having two bits change at precisely the same time is darned near impossible, and a processor as fast as the Propeller could easily catch one changing before the other, resulting in a bad input; Gray code solves this with the single bit change between steps.

Dealing with a Gray code encoder is quite simple: scan the inputs, check for a change, and on a change determine direction. Adding to the mix, Wayne's encoder – and the one I went with for my own project board – has a push-button and detents, that is, it "clicks" when you turn the shaft. The button is no problem, we know how to debounce buttons and we can add that to the object code. The detents create an extra bit of work but once we understand how the encoder behaves you'll see it's also pretty simple.

Figure 1 shows the output of the encoder using normally-open pins that are pulled up to Vdd – this comes right out of the data sheet so I matched it (see Figure 2 for the schematic). Note the location of the detents (when both outputs are off) – we'll adjust the object for this so that one piece of code will work with "detented" encoders as well as those that freely spin and can stop at any output pattern.

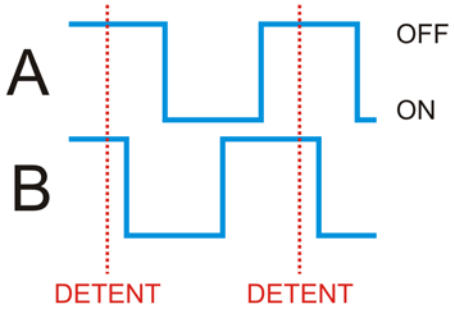

Figure 1

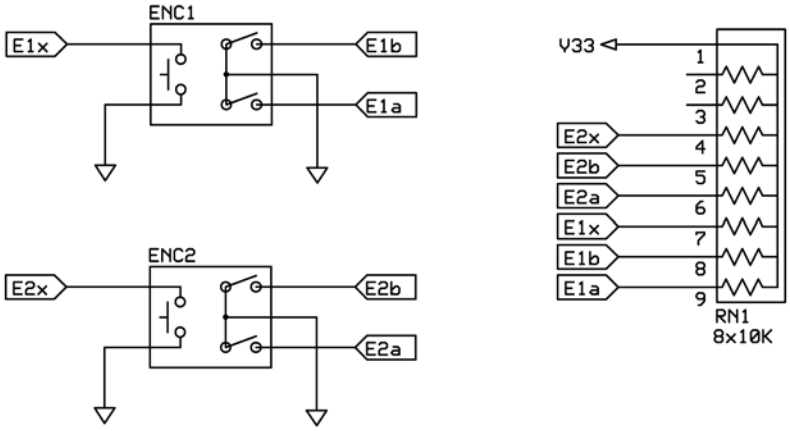

Figure 2

Let's jump in. My main goal with this object was to be able to initialize it, and then ask for the position value and button status – everything else is handled behind-the-scenes in the encoder cog. To keep things really flexible we'll allow the ability to reset the current position value, and this will be validated (and corrected if necessary) by the object.

Here's the method we'll call to initialize the encoder object:

```
pub init(base, btn, detent, lo, hi, preset)

  finalize

  enctiming := (clkfreq / 1_000) >> 2
  basepin   := base
  btnmask   := |< btn

  if detent
    hasdetent := true
    lolimit   := lo << 2
    hilimit   := hi << 2
    encoder   := preset << 2
  else
    hasdetent := false
    lolimit   := lo
    hilimit   := hi
    encoder   := preset
```

```
  cog := cognew(@grayenc, @encoder) + 1

  return cog
```

The first parameter is the base pin, which is the A pin of the encoder. To keep the code simple we expect the next higher pin will be the encoder's B pin. The second parameter is the encoder's button input. Note that all three pins are active low, that is, they are pulled-up to Vdd through 10K and will go low when active.

Next up is the true (non-zero) or false (zero) value that specifies whether the encoder has detents or not. Finally, we'll pass the low, high, and initial (preset) values for the driver. If, for example, we had an encoder with a base pin of 3, switch pin of 5, is "detented," will span from -100 to +100, and starts at zero, we would initialize it like this:

```
  encoder.init(3, 5, true, -100, 100,  0)
```

When a detented encoder is used the limits and preset values are shifted left by two, which is the same as multiplying by four. We have to do this to account for the steps in between each detent. And, no, shifting negative values left, for the range we would typically use, is not a problem. I tested this theory on values down to minus ten million – a value we'd never use in an actual project – just to make sure.

The range limits take priority over the preset value; if the preset is outside the specified range the object will fix it; let's have a look at that.

```
grayenc         rdlong  tmp1, par
                mins    tmp1, lolimit
                maxs    tmp1, hilimit
                wrlong  tmp1, par
```

At `par` is the address of the encoder value. After reading the preset value into *tmp1* we use signed versions of `min` and `max` to ensure that it is within the stated bounds. Yes, we could have done this in Spin in the `init()` method, but it's easy and fast so it just seemed like this was the best place to handle it.

Next up is basic initialization of the button debounce workspace, the previous scan result (stored in *oldscan*), and creating a timer. We don't need the timer for the encoder, but it does come into play for debouncing the button input.

```
setup           mov     btnwork, #0
                mov     tmp1, par
                add     tmp1, #4
                wrlong  btnwork, tmp1

                mov     oldscan, ina
                shr     oldscan, basepin
                and     oldscan, #%11

                mov     timer, cnt
                add     timer, enctiming
```

And now we get to the guts of it.

```
encloop         waitcnt timer, enctiming

scan            mov     newscan, ina
```

```
                mov     tmp1, newscan

chkbutton       test    btnmask, tmp1    wc
        if_c    mov     btnwork, #0
        if_nc   add     btnwork, #1
                max     btnwork, BTN_TM  wc
        if_c    mov     tmp1, #0
        if_nc   mov     tmp1, IS_PRESSED
                mov     tmp2, par
                add     tmp2, #4
                wrlong  tmp1, tmp2
```

As ever, delays are a breeze in PASM with the `waitcnt` instruction. I've set the encoder to run on a 250 microsecond loop, that way if we get really zippy with the encoder knob the program can still keep up.

The present state of the input pins are copied into *newscan* and that is copied into *tmp1* where we'll use it to check on the button. To check the button input we AND (using `test`) the button mask with *tmp1* and save the result in the Carry flag. We're using an active-low circuit, so a set Carry flag means that the button is not pressed. If that's the case the value of *btnwork* is cleared, otherwise it's incremented.

The next step, using `max`, actually does two things for us: 1) it keeps the value of *btnwork* at the debounce timing limit to prevent a roll-over on a stuck switch, and 2) the Carry flag indicates whether *btnwork* is less than `BTN_TM` (set for 25 ms). If the Carry flag is set the button is not fully debounced and we move zero to *tmp1*, otherwise we move `IS_PRESSED` (true) to *tmp1*, then write it to the hub at the address for the button status. Again, the encoder value address is stored in par, so four (for a long value) is added to this value to get the correct address of the button status variable.

With the button debounced (or not) we can check to see if the encoder moved. Well start by isolating the encoder inputs and comparing them to the last scan.

```
chkencoder      shr     newscan, basepin
                and     newscan, #%11
                cmp     newscan, oldscan  wz
        if_e    jmp     #encloop
```

Now you can see why we want the A and B pins in contiguous, ascending order on the inputs; we're simply shifting the scan value right by the base (A) pin number and masking off the other bits. This is compared to *oldscan* and if they're equal (i.e., no change) we jump right back to the top.

Okay, I know there's more than one of you hardcore types that might want to go willy-nilly on input mapping; maybe a PCB routing problem prevents keeping the pins contiguous and in ascending order. Here's what to do: Create pin masks for the A and B pins (just like we did for the button input) and then change the code like this:

```
chkencoder      mov     tmp1, #0
                test    amask, newscan   wc
                muxc    tmp1, #%01
                test    bmask, newscan   wc
                muxc    tmp1, #%10
                mov     newscan, tmp1
                cmp     newscan, oldscan wz
        if_e    jmp     #encloop
```

As you can see, this version tests each input and moves them, through the Carry flag, into the correct locations in *newscan*. I really like the `muxc` operator; this code snippet shows how useful it is, allowing us to move what's in C to any bit position of a variable. Remember, if you update the PASM code to handle non-contiguous encoder pins you'll need update the initialization of *oldscan* and add a B pin parameter to

the `init()` method. Actually, I've done the work for you (see *jm_grayenc2btnx.spin*; this version allows us to disable the button pin as well).

Okay, let's say we have a change. What I use is an assembly version of a case structure, using the previous scan and comparing it to the value for a positive (clockwise) change.

```
case11          cmp     oldscan, #%11  wz
        if_ne   jmp     #case01
                cmp     newscan, #%01  wz
                jmp     #update

case01          cmp     oldscan, #%01  wz
        if_ne   jmp     #case00
                cmp     newscan, #%00  wz
                jmp     #update

case00          cmp     oldscan, #%00  wz
        if_ne   jmp     #case10
                cmp     newscan, #%10  wz
                jmp     #update

case10          cmp     oldscan, #%10  wz
        if_ne   jmp     #encloop
                cmp     newscan, #%11  wz
```

You'll see that each section is identically constructed. If the new scan represents a clockwise move the Z flag will be set, otherwise the Z flag will be cleared. The program then jumps to `update` which does the final routing.

```
update          rdlong  tmp2, par
        if_nz   jmp     #decvalue

incvalue        adds    tmp2, #1
                maxs    tmp2, hilimit
                wrlong  tmp2, par
                mov     oldscan, newscan
                jmp     #encloop

decvalue        subs    tmp2, #1
                mins    tmp2, lolimit
                wrlong  tmp2, par
                mov     oldscan, newscan
                jmp     #encloop
```

At `update` we retrieve the encoder value from the hub (because it could have been changed by the top-level program) and then increment or decrement it (based on the state of the Z flag), using the previously-defined limits. The updated value is written back to the hub and we're done.

Well, almost – we need a method to read the current encoder value in our top-level program.

```
pub read

  if hasdetent
    return (encoder ~> 2)
  else
    return encoder
```

Remember that situation with the detented encoders and the 4x multiplier? We multiplied the limits and preset value by four to accommodate the changes between clicks. Well, if we're using a detented

encoder we can't simply shift the value right to return it to the intended range; doing this on a negative number would cause it to go positive (because a 0 would be shifted into bit 31, the sign bit). No worries, Spin has a really cool operator call Shift Arithmetic Right (~>). This operator does a right shift while maintaining the sign of the value, allowing the use positive and negative numbers with no muss or fuss.

More Easy Input Expansion

The product I mentioned in the opening is a 16-channel DMX controller that my business partner, [former Parallax engineer] John Barrowman, and I designed. As control of the outputs is paramount we use P0..P15 for the output channels. With the RS-485 circuitry, a two-position mode switch, and other IO requirements, we just didn't have enough pins on the Propeller to accommodate the nine-bit DMX address (like I did in my small DMX project last November).

The solution? Dirt easy: use a 74x165 input shift register. With three pins we get eight inputs; for the DMX address we simply connected bit 8 of the address switch directly to the Propeller. For my Propeller Platform add-on, I used an 8-bit switch and a 2x8 header (to allow off-board switches); see Figure 3 for the schematic.

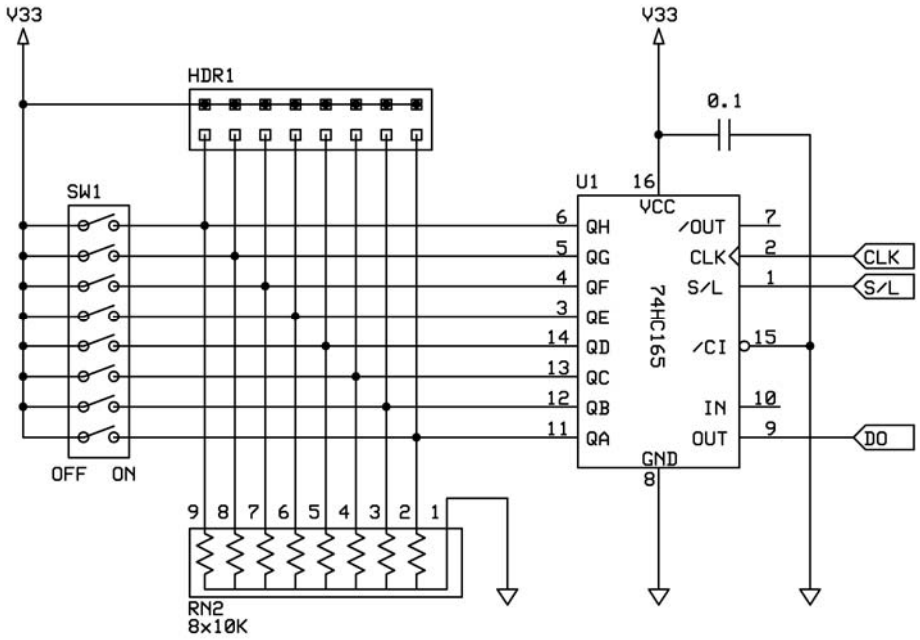

Figure 3

If you've used the 74x165 with PBASIC or SX/B you probably used `SHIFTIN` to read it. We can't do that with the Propeller, directly, anyway, as there is no built-in `SHIFTIN` instruction. You might be wondering why. For all its power, the Propeller has to squeeze the Spin interpreter into a single cog and that's tough work, so some niceties from PBASIC are not included.

No problem! – we'll just write our own method, and no assembly (PASM) is required. The code the follows is very similar to PBASIC's `SHIFTIN`, reading a single byte in MSBFIRST mode, though it is setup to accommodate the Shift/Load line of the 74x165.

```
pub in165(ld, do, clk) | tmp165

  outa[ld]~~
  dira[ld]~~
  outa[clk]~
```

```
  dira[clk]~~
  dira[do]~

  outa[ld]~
  outa[ld]~~

  tmp165~
  repeat 8
    tmp165 := (tmp165 << 1) | ina[do]
    outa[clk]~~
    outa[clk]~

  return tmp165
```

As with PBASIC and SX/B `SHIFTIN`, this method takes care of setting the IO pins used to the required states. We start by making the Shift/Load pin an output and high, the Clock pin an output and low, and the Data Out (from the x165) an input.

The Shift/Load line is blipped low, then back high; this latches the present state of the eight inputs to an internal register. With the data latched it can be shifted into the Propeller using a `repeat` loop.

The first line of the `repeat` loop does all the hard work; it preps the work value by shifting it left one bit and then OR'ing the state of the DO pin to the value (in bit 0). After the bit is moved into the work variable the Clock line is blipped high then back low to get the next bit.

One of the things that Spin does not have is dot notation for bits like we've used in PBASIC and SX/B. For example, what if we wanted to convert the routine above to shift the bits in LSB first? Here's the change that makes that happen:

```
  repeat 8
    tmp165 := (tmp165 >> 1) | (ina[do] << 7)
```

In this case we shift the work variable one bit right and then OR the DO value into bit 7. Note that we had to shift the DO bit before the OR operation. There's one more trick we could use, the reverse (><) operator. For example, we could create a global constant called LSBFIRST that would be applied after the value is shifted in – like this:

```
  repeat 8
    tmp165 := (tmp165 << 1) | ina[do]
    outa[clk]~~
    outa[clk]~

  if LSBFIRST
    tmp165 ><= 8
```

While I haven't personally needed to use more than one 74x165 with a Propeller project, it's certainly a possibility. Here, then, is a final version of the routine that accommodates the number of bits to shift as well as the shift mode. With this version of the routine we could read up to four, daisy-chained '165s into a single, 32-bit long.

```
pub in165x(ld, do, clk, bits, mode) | tmp165

  outa[ld]~~
  dira[ld]~~
  outa[clk]~~
  dira[clk]~~
  dira[do]~

  outa[ld]~
```

```
  outa[ld]~~

  tmp165~
  bits <#= 32
  repeat bits
    tmp165 := (tmp165 << 1) | ina[do]
    outa[clk]~~
    outa[clk]~

  if (mode == LSBFIRST)
    tmp165 >><= bits

  return tmp165
```

The changes should be apparent: We'll limit the bit count to 32 for obvious reasons, update the `repeat` loop to use the *bits* parameter, then use the *mode* parameter with *bits* to adjust if LSBFIRST (*mode* = 0) is desired.

Since I've already made use of the 74x165 in two Propeller designs and I anticipate using Gray code encoders in a couple more, I created a little prototyping add-on for the Propeller Platform that includes the '165 and two encoders. These components don't take much space so I filled the rest with pads to place other components. Figure 4 show the board attached to my original Propeller Platform. Figure 5 shows the output of a simple demo using the Parallax Serial Terminal through the programming connection.

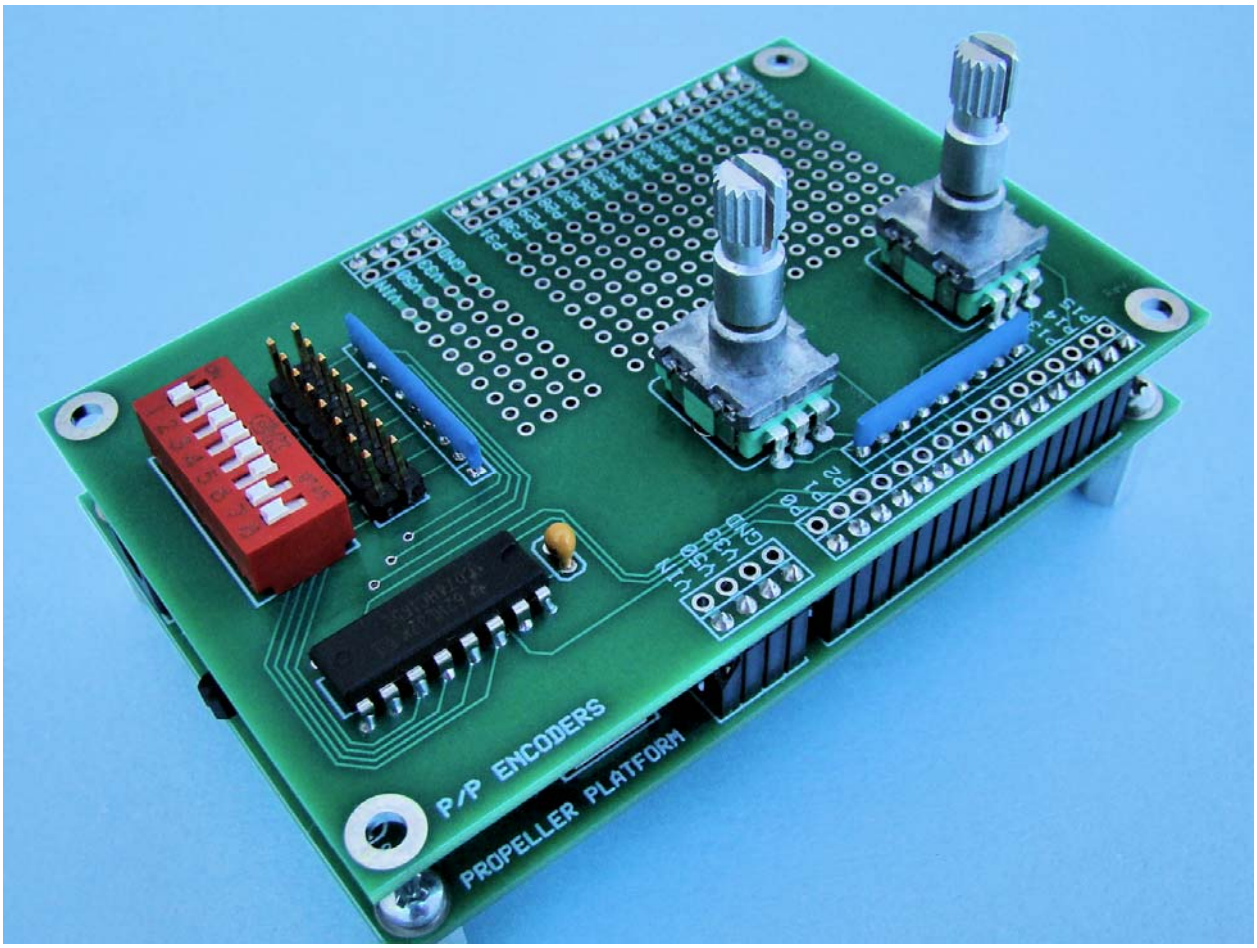

Figure 4

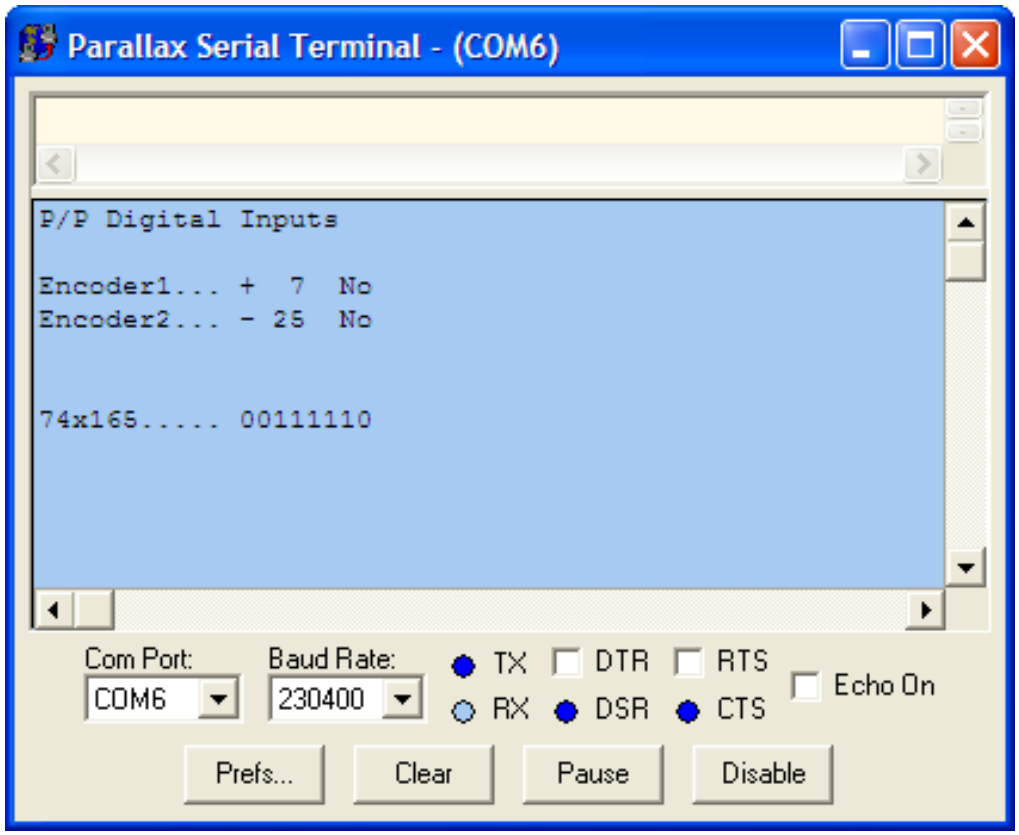

Figure 5

PropBASIC Follow-Up

Wow, the response to PropBASIC was really amazing. To be honest, I was a little surprised, but then I really shouldn't be, should I? Those of us with Parallax experience have a lot of time with BASIC and this new tool made moving to the Propeller easier for some PBASIC and SX/B users.

Of course things got even better when Brad Campbell, an Australian Propeller programmer (and very nice guy), integrated the PropBASIC compiler into his BST (Brad's Spin Tool) IDE. What does this mean? Well, if you looked past the Propeller to something like, say, the Arduino because of the availability of a cross-platform development tool, well... time to drop the single-core processor and move on up to the multi-core Propeller. With BST you can program the Propeller in Spin, PASM, or PropBASIC, on nearly any Windows, Mac, or Linux PC. Now, that's cool!

The easiest way to get the BST IDE and PropBASIC compiler files you need is through links at www.propbasic.com.

Before I close let me correct a small error in my last column. When an IDE like the Propeller Tool or BST is downloading to the Propeller it is the Propeller – not the IDE as I misstated – that makes the adjustment for baud rate. This makes better sense; typically the receiver does the "auto baud" detection and configuration. I apologize for any confusion.

Okay, then; until next time – on a PC, a Mac, or Linux box – have fun and keep spinning and winning with the Propeller.

Resources

Jon “JonnyMac” Williams
jwilliams@efx-tek.com

Parallax, Inc.
www.parallax.com

PropBASIC
www.propbasic.com

Bill of Materials

C1	0.1uF	Mouser 80-C315C104M5U
ENC1, ENC2	Encoder	Mouser 858-EN11-HSB1AF20
HDR1, X1-X4	0.1 male	Mouser 517-6111TG
RN1, RN2	8x10K	Mouser 81-RGLD8X103J
SW1	DIPx8	Mouser 611-BD08
U1	74HC165	Mouser 595-CD74HC165E
PCB		ExpressPCB