Design Considerations for the TAOS/Microchip Absolute Steering Angle Encoder Demo

Phil Pilgirm
Bueno Systems, Inc.

Introduction

The objective of this project was to design an absolute encoder using the TAOS TSL1401R linear image sensor, mated to a Microchip Technologies PIC16F819 microcontroller, capable of meeting the following specs:

- **Resolution:** 0.088 degrees (i.e. 4096 codes/revolution, or 12-bits)
- **Maximum Turning Rate:** 1000 degrees/sec.
- **Minimum Response:** 100 readings/sec.

This paper describes the design considerations required to meet these specs.

Code Wheel

It was decided to use a single-track, maximal-length sequence code. The basics of this technique are covered in the TAOS document, *Position Detection using Maximal Length Sequences* by David H. Mehrl. This method was chosen over multi-track Gray-coding due to its simplicity and lack of critical alignment requirements. It's a method very well suited to linear array sensors such as the TSL1401R.

The question became how many bits to encode. The answer to this was derived from the length (8.128mm) and resolution (.0635mm) of the sensor and the diameter of the code wheel. The latter was dictated by the size of a readily-available housing, resulting in a 46mm active diameter (144mm circumference). To read a seven-bit code, for example, we should be able to see several more than seven bits at a time. Given the dimensions available, we would have 127 bits spread out over a 144mm circumference, or 0.88 bits/mm. With an 8mm-long sensor, we could see 8 C 0.88 = 7.04 bits, which is not quite enough. With an eight-bit code, this number doubles to 14 bits, which should be more than adequate. This leaves 128 / 14 = 9 pixels per bit, which should also be adequate. If we went to a nine-bit code, we could see 28 bits at a time, but each bit would cover only 4.5 pixels. That's too few to be reliable, so eight was chosen as the best number.

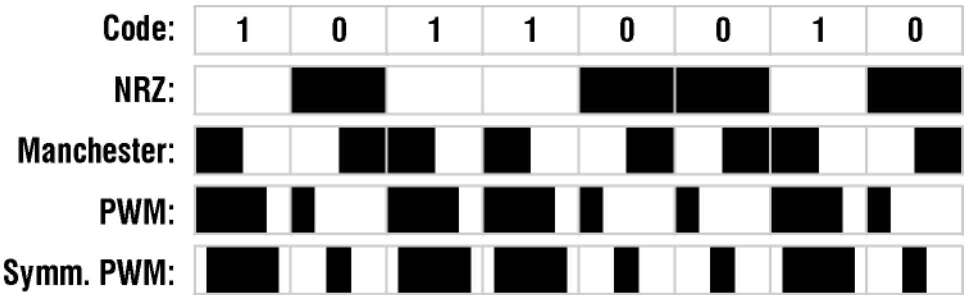
The usual maximal-length sequence code can be generated using a shift register and a handful of XOR gates, or by simulating such a method in a computer program. For an n-bit code, the length of such a sequence will be $2^n – 1$. This is because a code of all zeroes cannot occur. If initialized with such a code, the shift register will remain stuck there. In designing an encoder, this is a rather inconvenient situation. For computational simplicity, we'd like the number of unique codes around the codewheel to be a power of 2. It turns out that simply inserting an extra zero into the longest sequence of zeroes of the shift-register-generated code works fine. For example, a normal eight-bit maximal-length sequence will consist of 255 bits (128 ones and 127 zeroes) and contain the subsequence …100000001… That is to say that the eight-bit code 10000000 is immediately followed by the code 00000001. By sticking in the extra zero at this point, we obtain …1000000001…, which yields the 8-bit code sequence …, 10000000, 00000000, 00000001, …

The next change involves how to encode these bits on a codewheel. In the Mehrl paper, NRZ encoding was used. That is, the bits were assigned contiguous cells, and the entire cell was either black or white, depending on the value of the bit. Because we actually need 12 bits of resolution, we need to be able to infer the remaining four bits from the fine-scale position of the eight-bit code we read. But, in order to do this, each code bit, whether a one or a zero, needs to have an edge or two that we can discern. This is called “self-clocking”, and various methods are available to do this. One of the most popular, Manchester encoding, also has the advantage of a low spatial frequency. In this encoding method, each bit is represented as a transition at the center of its cell. A “one” bit is be encoded as a low-to-high transition; a “zero” bit, high-to-low. Manchester encoding has a couple disadvantages for this application, however:

1. It’s not always possible, given a snapshot of a few bits, to discern which transitions are at the centers of their bit frames and which ones are at the edges. For example, an isolated string of all “ones” will look identical to an isolated string of all “zeroes”. Clock synchronization needs to occur over a sequence that includes a mixture of “ones” and “zeroes” to be effective.
2. Manchester encoding is asymmetrical. This is to say that rising edges and falling edges do not surround the center of a bit frame equally. Each bit’s “position” is determined by the location of its central rising, or falling, edge. If a sensor were detecting bars slightly out of focus using a fixed analog threshold, say, changes in the overall pixel response would shift the apparent bar locations and throw off the results. Rising edges would appear to shift left as the overall pixel response increased; falling edges would shift right.

For these reasons, it was decided to use a symmetrical pulse-width modulation scheme. Wide bars would represent the “one” bits; narrow bars, the “zeroes”. Each bar would be centered within its bit frame, so the spacing from the center of one bar to the center of the next would always be the same. Moreover, any changes in overall pixel response might make a given bar look slightly wider or narrower, but it wouldn’t change its apparent center position, because its edges would “move” in opposite directions. Naturally, the bigger the ratio between wide bars and narrow bars, the more reliably they can be discriminated. But there’s a limit relating to the resolution of the detector. You don’t want bars so skinny that the sensor can’t see them. For this reason, a three-to-one ration was chosen. Wide bars are 75% of a bit frame wide; narrow bars, 25%. This leaves a minimum 25% gap between the widest bars. At nine pixels per bit frame, the narrowest bar will cover 2.25 pixels. Actually, because of blurring, it covers a little more than that, and the perceived ratio is somewhat less than 3-to-1.

Examples of the code representations discussed, plus a non-symmetrical PWM code, are shown below:



The actual eight-bit code was generated using a simulated shift-register/gate combination that employed the maximum number of XORs possible. This was done to facilitate future error correction, discussed in the last section of this paper. Taps were taken from bits 2, 4, 5, 6, 7, and 8. The following Perl script generates the sequence (adding the extra “0”). It prints out both an array used for the Postscript program that generates the codewheel and statements that plug into the PIC encoder firmware source as a lookup table.

```
use strict;
my $len = 8; my @taps = (2,4,5,6,7,8);
my $seq = '1' x $len;
my %mark;
my $zero = 0;
my $circum = '';
foreach (0 .. 2 ** $len - 1) {
    #print "$seq\n";
    my $digit;
```

```
   if ($seq =~ /^0+$/) {
      $digit = '1';
   } elsif ($seq =~ /^0+1$/ && !$zero) {
      $digit = '0';
      $zero = 1
   } else {
      my $nxt = 0;
      foreach my $tap (@taps) {
         $nxt += substr($seq, $tap - 1, 1)
      }
      $digit = sprintf('%1.1d', $nxt % 2)
   }
   $seq = $digit . substr($seq, 0, $len - 1);
   $circum .= $digit;
   $mark{bin($seq)} = $_;
}

if (keys %mark != 2 ** $len) {
   print "Wrong number of codes."
}

my $size = length($circum);
print "\n\n";
foreach (0 .. $size / 16 - 1) {
   print $_ ? '       ' : '/seq [ ';
   print join(' ', split(//, substr($circum, $_ * 16, 16)));
   print $_ == $size / 16 - 1 ? ' ] def' : "\n"
}

print "\n\n\n";
foreach (0 .. $size - 1) {
   print "\tRETW\t" unless $_ % 8;
   printf "%3.3XH", $mark{$_};
   if ($_ % 8 == 7) {
      printf "\t;%2.2X - %2.2X\n", $_ & 0xF8, $_
   } else {
      print ','
   }
}

sub bin {
   return unpack("N", pack("B32", substr('0' x 32 . shift, -32)))
}
```

The codewheel was created using a direct-to-film Postscript printer with a resolution of 2540dpi. It was decided to use dark bars on a clear background because the opaque portions of the film have cleaner light-transmission characteristics than the clear portions, due to possible surface scratches and smudges on the latter. The Postscript code for generating one each of a balanced PWM disk, a standard (left-justified) PWM disk, a Manchester disk, and a balanced PWM linear encoder strip is shown below.

```
72 72 scale

/seq [ 0 0 1 1 1 0 0 1 1 0 0 1 1 1 1 0
       0 1 0 1 1 0 1 0 0 1 1 0 1 0 0 0
       1 1 1 0 1 0 0 1 0 0 0 0 0 1 1 0
       1 1 1 1 0 1 0 1 0 1 1 0 1 1 0 0
       1 0 0 0 1 0 0 1 0 0 1 0 1 1 1 1
       1 0 1 0 0 0 0 1 0 0 1 1 0 0 0 1
       0 1 0 1 1 1 1 0 0 0 0 0 1 0 0 0
       1 1 0 0 0 0 0 0 0 0 1 0 1 0 0 1
       0 1 0 1 0 1 0 0 0 1 0 1 1 1 0 1
       1 1 0 1 0 1 1 1 0 0 1 0 0 1 1 1
       0 1 1 0 0 0 0 1 0 1 1 0 0 0 1 1
       1 1 1 1 0 1 1 0 1 0 1 1 0 0 1 1
       0 1 1 0 1 1 1 0 0 0 0 1 1 1 0 0
       0 1 1 0 1 0 1 0 0 1 1 1 1 1 0 0
```

```
       0 1 0 0 0 0 1 1 0 0 1 0 1 0 0 0
       0 0 0 1 1 1 1 0 1 1 1 1 1 1 1 1 ] def

/bits (8) def

/dia 1.95 def
/stripe .15 def

/outerrad dia 2 div def
/innerrad outerrad stripe sub def

/pi 3.14159265 def
/n seq length def


/findhalfangle
  { stringwidth pop 2 div
    2 xradius mul pi mul div 360 mul
  } def

/outsideplacechar
  { /char exch def
    /halfangle char findhalfangle def
    gsave
      currentpoint translate
      halfangle neg rotate
      radius 0 translate
      -90 rotate
      char stringwidth pop 2 div neg 0 moveto
      char show
    grestore
    halfangle 2 mul neg rotate
  } def

/insideplacechar
  { /char exch def
    /halfangle char findhalfangle def
    gsave
      currentpoint translate
      halfangle rotate
      radius 0 translate
      90 rotate
      char stringwidth pop 2 div neg 0 moveto
      char show
    grestore
    halfangle 2 mul rotate
  } def

/OutsideCircleText  % text size centerangle radius
  { /radius exch def
    /centerangle exch def
    /ptsize exch def
    /str exch def
    /xradius radius ptsize 4 div add def
    gsave
      centerangle str findhalfangle add rotate
      str
        { /charcode exch def
          ( ) dup 0 charcode put outsideplacechar
        } forall
    grestore
  } def

/InsideCircleText  % text size centerangle radius
  { /radius exch def
    /centerangle exch def
    /ptsize exch def
    /str exch def
```

```
    /xradius radius ptsize 3 div sub def
    gsave
      centerangle str findhalfangle sub rotate
      str
        { /charcode exch def
          ( ) dup 0 charcode put insideplacechar
        } forall
    grestore
  } def

/MakePWM { % x y moveto ratio ---
   /ratio exch def
   gsave
   currentpoint translate
   /dth 360 n div ratio 1 add div def
   0 1 seq length 1 sub {
      /i exch def
      /th0 i n div 360 mul def
      /th1 seq i get ratio 1 sub mul 1 add dth mul th0 add def
      newpath
      0 0 outerrad th0 th1 arc
      0 0 innerrad th1 th0 arcn
      closepath
      0 setgray fill
   } for
   .003 setlinewidth
   -.1 0 moveto .1 0 lineto
   0 -.1 moveto 0 .1 lineto stroke
   0 0 .1875 0 360 arc stroke
   /Helvetica findfont .06 scalefont setfont
   0 0 moveto
   (TAOSENC v2.0       8-bit Maximal Length PWM Sequence       \251  2004 Bueno
Systems, Inc.) .06 270 innerrad .06 sub InsideCircleText
   grestore
} def

/MakeBalancedPWM { % x y moveto ratio ---
   /ratio exch def
   gsave
   currentpoint translate
   /dth 360 n div ratio 1 add div def
   0 1 seq length 1 sub {
      /i exch def
      /th0 i n div 360 mul seq i get ratio 1 sub mul 1 add dth mul 2 div sub def
      /th1 seq i get ratio 1 sub mul 1 add dth mul th0 add def
      newpath
      0 0 outerrad th0 th1 arc
      0 0 innerrad th1 th0 arcn
      closepath
      0 setgray fill
   } for
   .003 setlinewidth
   -.1 0 moveto .1 0 lineto
   0 -.1 moveto 0 .1 lineto stroke
   0 0 .1875 0 360 arc stroke
   /Helvetica findfont .06 scalefont setfont
   0 0 moveto
   (TAOSENC v2.0       8-bit Maximal Length Balanced PWM Sequence       \251  2004
Bueno Systems, Inc.) .06 270 innerrad .06 sub InsideCircleText
   grestore
} def

/MakeManchester { % x y moveto ratio ---
   gsave
   currentpoint translate
   /dth 360 n div 2 div def
   0 1 seq length 1 sub {
      /i exch def
```

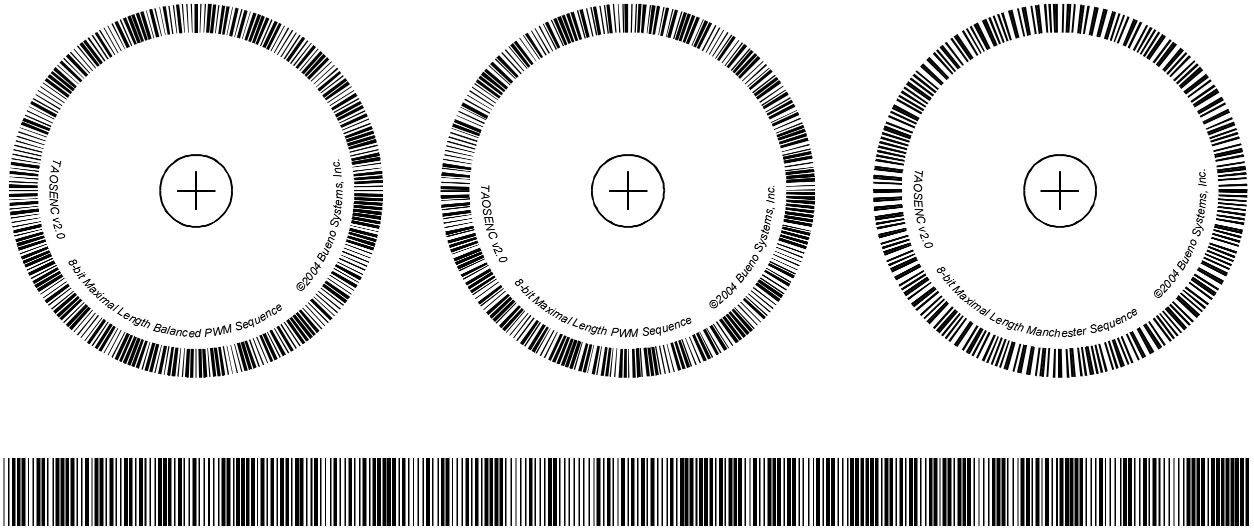
```
      /th0 i seq i get 2 div add n div 360 mul def
      /th1 th0 dth add def
      newpath
      0 0 outerrad th0 th1 arc
      0 0 innerrad th1 th0 arcn
      closepath
      0 setgray fill
   } for
   .003 setlinewidth
   -.1 0 moveto .1 0 lineto
   0 -.1 moveto 0 .1 lineto stroke
   0 0 .1875 0 360 arc stroke
   /Helvetica findfont .06 scalefont setfont
   0 0 moveto
   (TAOSENC v2.0       8-bit Maximal Length Manchester Sequence       \251  2004 Bueno
Systems, Inc.) .06 270 innerrad .06 sub InsideCircleText
   grestore
} def

/MakeLinearPWM { % x y moveto length width ratio MakeLinearPWM
   gsave
   0 setgray
   0 setlinewidth
   currentpoint translate
   /ratio exch def
   /wid exch def
   /len exch def
   /dx len n div ratio 1 add div def
   0 1 n 1 sub {
      /i exch def
      /x0 i n div len mul seq i get ratio 1 sub mul 1 add dx mul 2 div sub def
      /x1 seq i get ratio 1 sub mul 1 add dx mul x0 add def
      newpath x0 0 moveto x0 wid lineto x1 wid lineto x1 0 lineto closepath fill
   } for
}  def

2 2 moveto 3 MakeBalancedPWM
4.25 2 moveto 3 MakePWM
6.5 2 moveto 3 MakeManchester

1 .25 moveto 6.5 .355 3 MakeLinearPWM
```

This program will produce the output shown below.



Light Source

To image the film codewheel onto the sensor, two methods may be employed: 1) lensing, and 2) direct projection. In the former, a lens is required between the sensor and the film, which will focus an image of the film onto the sensor. The latter method requires only a point source of light opposite the film from the sensor. Light emanating from the source will cast shadows on the sensor corresponding to the dark areas of the film. Because of its simplicity, lower cost, and minimal space requirements, the direct projection method was chosen.

To obtain the most sharply-defined shadows, the smallest practical point-light source should be used. The bigger the light source, the fuzzier the shadows will appear to the sensor. For a given light-to-dark or dark-to-light transition, the apparent width of the "fuzziness" is given by the formula,

Fuzziness = Light diameter C Film-to-sensor distance / Light-to-film distance

For this reason, we want to keep the film as close to the sensor and as far away from the light as possible. The fuzziness should be no more than a fraction of a pixel, if possible, or, in the case of the TSL1401R, to a fraction of 63.5 microns. There are point source IREDs (infrared-emitting diodes) that have emitting surfaces around this scale. The one chosen for this application is the Optek OP230WPS. It has a 100-micron-square emitting area. By positioning the IRED at a distance from the film and the film close to the sensor, this fraction-of-a-pixel criterion can be achieved. Keeping the source back from the film also reduces the intensity (cosine-law) falloff that occurs near the ends of the sensor as well as any parallax distortion caused by fluctuating film-to-sensor distances. Of course, if it's too far back, the overall intensity reaching the sensor may become too low, due to the inverse-square law.

The requirement that the encoder be able to read the film while it's moving at 1000 degree/sec places severe restrictions on how much integration time can elapse during one exposure. Too long an integration time will result in severely blurred images. The speed requirement works out to about 711 code bits passing each pixel per second. Since each code bit is about nine pixels wide, a moving code-bit shadow will race across the sensor face at about 6400 pixels per second or, inverting, 156 microseconds per pixel. So, in order to "freeze" this motion to take a reading, we need an exposure time that's a fraction of 156 microseconds. Because the TSL1401R is an integrating light sensor, one might be tempted to limit its integration time to obtain the necessary "shutter speed". But during each integration, all 128 pixels need to be read out before the next integration can be started. To do this with an integration time of 20µSec, say, would require a pixel clock and A/D converter speed of 6.4MHz. But such speeds are not possible in PIC16F819. Therefore it was decided to strobe the IRED during integration and use however long an integration was necessary to obtain the sensor data and perform the necessary computations. This creates the necessity for zero ambient light. That's because any outside light will contribute to the pixel response *over the whole integration period* and not just during the IRED strobe pulse. So just a little ambient light (noise) could easily swamp out light from the IRED (signal). This requirement is met by putting the code wheel and all the electronics in a light-tight enclosure.

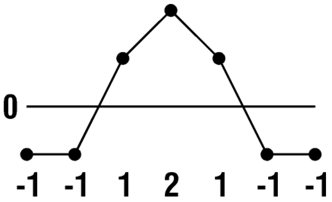
Pixel Data Input and Auto-Exposure

The analog pixel data from the TSL1401R is converted to digital values by the PIC16F819's built-in 10-bit A/D converter. Not much light reaches the sensor in each exposure, given the short IRED pulses, so the peak voltage from the TSL1401R's output is well below its 5V saturation level. For this reason, only the eight least-significant bits of the 10-bit A/D result are used. This gives us a more reasonable high end of 1.25 volts.

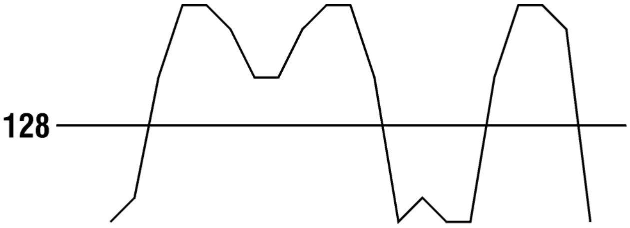
The IRED driver doesn't incorporate any current regulation – only a current-limiting resistor. Because light output can vary under these circumstances, there needs to be a way to regulate the pulse width to keep the sensor response constant. This, in fact, is easy to do. It's accomplished by checking the lowest pixel value recorded in each scan (exposure). If it's below a certain level, the strobe width will be incremented a notch; if above, decremented. If the value falls outside a window of acceptability, the encoder goes into a “not ready” state, the pulse is adjusted, and another attempt is made. This should only happen at powerup.

Limiter and FIR Filter

The codewheel-modulated output waveform from the sensor is not an ideal square wave, as one might like it to be. Because the ends of the sensor get slightly less light than the center, the amplitude is slightly “crowned” in the middle. And because of minor scratches, smudges, and the like, the waveform for the clear portion of the film is rather bumpy. So we have both low-frequency and high-frequency noise present. Thus simply thresholding the output at a fixed level to extract the binary code data is not an option. The ideal way to get rid of the noise would be to use a bandpass filter. A symmetric FIR (finite-impulse-response) filter is preferable here, because it won’t affect the relative positions (phase) of the features we’re trying to preserve and measure. But, especially to filter out the high-frequency noise, an adequate FIR filter would require many taps and be computationally burdensome on the PIC, lacking as it does a hardware multiplier – let alone a MAC (multiply-accumulate coprocessor). So we resort to a little chicanery. To get rid of the highest frequency noise, we simply lop it off. In other words, we pass the waveform through a *limiter*. Every pixel value above a certain level is limited to that level. Now it’s possible to use a little bit simpler FIR filter. But we need to simplify even more. In order to avoid software multiplication, each filter coefficient will be limited to a power of two. After considerable experimentation, the following set of filter coefficients was arrived at:



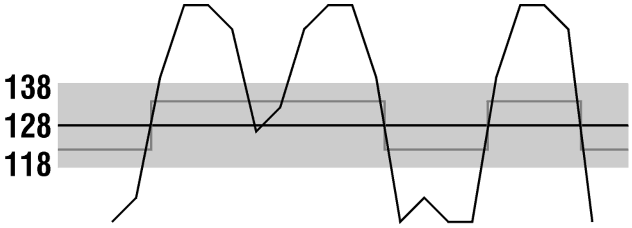
This is essentially a high-pass filter that removes the lowpass “crowning” noise from the waveform. In practice the result is added to a constant value of 128 to get the final analog waveform. This filter is sub-optimal, however, as the resulting waveform still has some filter artifacts consisting of dips in the wider peaks and bumps in the wider troughs, as shown below:



It would be nice just to be able, now, to use 128 as a threshold to delimit the bright pixels from the dark ones. But sometimes those artifacts come dangerously close to – or even cross – the threshold line. So more work needs to be done, as the next section describes.

Sub-Pixel Interpolation and Thresholding

Once we’ve determined the eight-bit code we’re looking at, we need to interpolate its position *within one bit frame* to another four bits, to get 12-bits of circumferential resolution total. But a bit frame is only 9 pixels wide, and we need at least 16 to get the extra precision. To accomplish this, we resort to *sub-pixel interpolation*. This means that between each pair of real pixels, we create a virtual pixel whose value is the average of the two real ones surrounding it. This gives us a total of 255 pixels – 18 for each bit frame. Moreover, because the center position of each dark bar is calculated by averaging the positions of *two* edges, we actually have 36 possible center positions from which to compute a four-bit extension to the eight-bit code. The sub-pixeling is actually done during the thresholding process. We look at a pixel to see if it's over or under a threshold value, then we look at the average of that pixel and the next one, then the next pixel by itself and so forth. To accommodate the filter artifacts discussed in the previous section, we include in the thresholding operation some hysteresis. This means that to transition from light-to-dark or dark-to-light, the waveform has to cross not only the center line, but that and a little more: the hysteresis value. This creates a hysteresis band, which the waveform must pass *all the way through* to be deemed as having changed state. The following illustration, with a twenty-point hysteresis band, demonstrates this principle:



Notice, however, that for maximum accuracy, we need to keep track of where the waveform crosses the centerline – *not where it exits the hystersis band* – to locate the edges of the bars. The hysteresis band exists simply to determine whether a given centerline crossing is real or not.

Decoding and Position Interpolation

Once the gray-level pixel waveform has been converted to 255 black/white bits, we can begin decoding the resultant wide and narrow bars into a meaningful position. (Actually, the thresholding and measuring are done in the program concurrently with the limiting and filtering. But it's more useful, for pedagogical purposes to discuss them as if they were separate steps.) We begin by measuring the widths of all the dark areas in the scan. Each of these widths is compared to a threshold to determine whether it's a wide bar or a narrow bar. Wide bars are shifted into a shift register as "one" bits; narrow bars, as "zero". There will be about fourteen bits shifted in altogether. The centroids of the two bars straddling the center pixel are also recorded. These are computed as the *sum* of the two edge positions, so can range from 0 to 510. Then the four shift register bits left of center and the four bits right of center are combined to form an eight-bit binary value. This value is an eight-bit substring of the 256-bit maximal-length sequence code. It is used to poll a lookup table, which returns the unique position (0 – 255) of that substring on the codewheel. Next, the two centroid positions (call them **L** and **R**) are used to calculate the last four bits, as follows:

$$\textbf{Four-bit-value} = \textbf{int}[16 * (R - 256) / (R - L)]$$

This is then tacked onto the end of the eight-bit decoded value to form a 12-bit result.

Looking Forward

What's been described to this point is enough to make a working demonstration encoder. Each scan, with all its computations, can be completed in well under 10ms, thus meeting the 100 samples-per-second requirement. But to make this into a product suitable for use as a production steering-angle encoder, much more needs to be done. This includes:

- **Better filtering of the sensor waveform.** This will probably entail a more capable microcontroller. Microchip's dsPIC is a a good candidate for this. It includes signal-processing functions such as a hardware MAC. This would allow a precisely-tailored FIR filter to be constructed and optimized to filter the encoder bars from both low- and high-frequency noise.
- **Finer position interpolation.** To get those last four bits, we've relied on a number that can take on no more than 36 values. This is pretty coarse and can lead to non-linearities in the encoder output. By extending the subpixel interpolation to four values per pixel, instead of two, we could theoretically

double this to 72. A larger codewheel would also prove helpful here, as it would increase the number of pixels per bit frame.

- **Error detection for failsafe operation.** Even with better filtering, one cannot discount the possibility of a misread code. Fortunately, the proper choice of a maximal-length sequence gives us the opportunity to do some error checking. Remember that each bit is some function of the eight that came before it. We used a formula that XORed bits 2, 4, 5, 6, 7, and 8, to get a new bit 0. The only bits not entering the computation were bits 1 and 3. But after the first computation, followed by a shift, these are now bits 2 and 4 and enter into the next computation. What this says is that if we can examine ten sequential bits, the two latest bits must result from some function that combines the values of *all eight* preceding bits. That is, they form a checksum of sorts for the other eight bits. If the computed and observed checksums disagree, we know we've misread the codewheel and can take corrective action.
- **Calibration.** Even though the maximal-length sequence code is forgiving of minor mechanical misalignment, such misalignment can still lead to errors. By creating fifteen bits of raw output, for example, instead of twelve, we could obtain enough extra headroom to calibrate out these errors post-production. This would be done by recording the encoder's output at equally-spaced, known mechanical positions. Thus, for each of these output values, we'd have a desired value. From these samples it would be possible to derive an interpolation function by which to compute a desired value from any measured value. The coefficients for this function could then be stored in the PIC's EEPROM. (The demo encoder, in fact, employed a small bit of calibration, allowing the position offset and code direction to be determined and recorded in its EEPROM.)
- **External interface.** The demo unit used an I^2C interface. As auto manufacturers are standardizing on the CAN bus for interfacing, this will have to be incorporated into the encoder as well. Again, the dsPIC could help in this regard.
- **Environmental hardening.** The demo encoder used a plastic housing that is marginally light-tight and certainly not hermetically sealed. Moreover, no consideration was given to the temperature characteristics of the components used to build it, and the circuit required a regulated five volts to operate. The enemies of ultimate success here are temperature extremes, voltage fluctuations, dust, vibration, moisture, and condensation. Each of these must be dealt with effectively to yield a unit rugged enough to survive the automotive environment.