599 Menlo Drive, Suite 100
Rocklin, California 95765, USA
**Office:** (916) 624-8333
**Fax:** (916) 624-8003

**General:** info@parallax.com
**Technical:** support@parallax.com
**Web Site:** www.parallax.com
**Educational:** www.stampsinclass.com

# Applying the Boe-Bot Digital Encoder Kit (#28107)

By Philip C. Pilgrim

## Introduction

This paper details my experiences designing and applying a set of digital wheel encoders (available from Parallax, Inc. as the "*Boe-Bot Digital Encoder Kit*", catalog number 28107) for Parallax's Boe-Bot, a two-wheel-plus-stabilizer-ball robot. I undertook this project in order to help the Boe-Bot navigate with some degree of precision without having to rely solely on external references such as a compass or sonic ranging module. By incorporating optical encoders, one is able to tell how far each wheel has turned and is, hopefully, able to coordinate the wheels' respective movements to guide the Boe-Bot to a desired destination. In addition, even without coordinated movement, and given any sequence of encoder outputs (along with the directions of rotation), one should be able to tell where the Boe-Bot is and in what direction it's pointing. This is known as "wheel odometry".

## The Goal

The goal of this exercise was to come up with an encoder system for the Boe-Bot wheels that was simple, easy to install, and didn't rely on a separate co-processor to manage the encoder pulses or servo pulses. This meant that the BASIC Stamp would not only have to send servo pulses to the motors but also have to count and process the resulting encoder pulses – all the while using this information to adjust and coordinate the servos on the fly. Simplicity dictated minimal modifications to the robot's hardware. Fortunately the new Boe-Bot wheels come equipped with eight evenly spaced holes – enough for eight full pulses or sixteen pulse edges per revolution. This is a big enough number to be useful, yet small enough not to overwhelm the BASIC Stamp with too high a pulse rate. With a simple optical sensor, the wheels themselves could become ideal encoder disks. But first, some math.

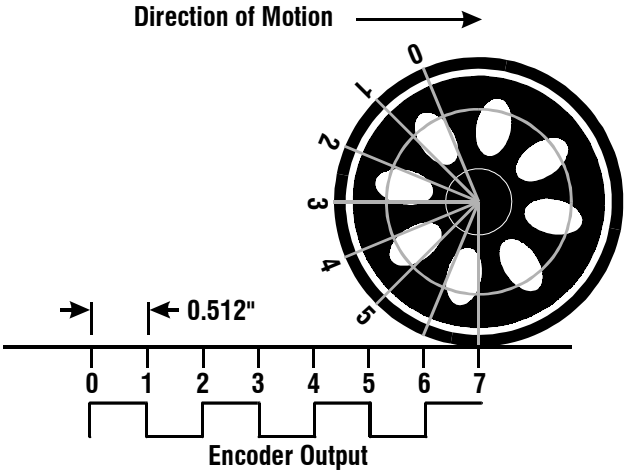
## The Theory

### *Moving*

Assume a robot with two opposing drive wheels, each of diameter **d** and distance **w** apart. Also assume that as each wheel turns, an encoder outputs **n** equally-spaced pulses per revolution. Each of these pulses will correspond to a distance of travel **D** equal to the wheel circumference divided by **n**, or

$$\mathbf{D = \pi d / n}$$

where $\pi$ is the familiar 3.14159265… So for a typical Boe-Bot wheel of 2.61" (67.06 mm) diameter and an encoder that puts out, say, 16 pulses per revolution, each pulse corresponds to 0.512" (13 mm) of travel, as the following diagram illustrates:




Therefore, if both wheels were synchronized and turning in the same direction for 7 pulses (actually *pulse edges*, but we'll call them pulses for brevity), the Boe-Bot would have moved 3.6" (91 mm) in a straight line. For the subsequent discussion, instead of inches and millimeters, we will use a new unit of length, the *ep*, equal to the distance (0.512" or 13 mm) traversed over the span of one encoder pulse (hence the name). If we had a way of monitoring and counting the encoder pulses as the Boe-Bot moved forward, we could control the servo motors to guarantee a certain distance of travel.

### *Turning*

By rotating the Boe-Bot's wheels in opposing directions, it is possible to effect a change in orientation. What does this mean in terms of encoder pulses? Starting with a Boe-Bot facing due north, suppose the left wheel were to move forward by 16 encoder pulses while the right wheel moved in reverse by 16. After pivoting on its center in this fashion, what direction will the bot then be facing? A typical Boe-Bot has wheels about 4.16" (8.125 eps) apart (*i.e.* **w = 8.125**), from the center of one tread to the center of the other. So when the Boe-Bot pivots on its center, the wheel contact points will form a circle 8.125 eps in diameter. This circle will have a circumference

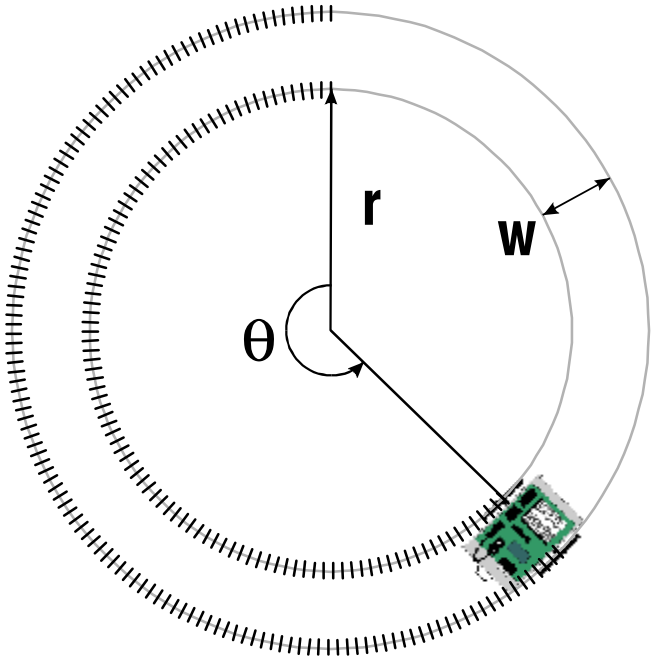
$$c = \pi w$$

or, for the Boe-Bot, about 25.52 eps. But we already know that each wheel has moved 16 eps along this circumference which, as a fraction of the full circumference is 16 / 25.52, or .627. This equates to 225.9 degrees or 160.6 brads (binary radians). So, for pivoting, we can write the formula for the angle of turn $\theta$ as

$$\theta \text{ (brads)} = m(\pi d / n) / (\pi w) \cdot 256$$
$$= (md) / (nw) \cdot 256$$

where **m** is the number of encoder pulses in each direction. Substituting the Bot-Bot's constants and a 16 pulse-per-revolution wheel encoder, we get $\theta = 10.038\ m$, or 10.038 brads per encoder pulse. For 16 encoder pulses, for example, we would get a net rotation of about 160.5 brads.

Center pivoting is only one kind of turn, though. Consider the case where the left wheel goes forward by 100 pulses, while the right wheel goes forward 132 pulses. Obviously, the Boe-Bot will veer to the left, but by how much? Assuming the motions of the two wheels are coordinated, the bot will move in an arc, as shown in the following diagram:




The inner track (arc) will have a length of 100 eps; the outer, 132 eps. Both of these tracks have a common center, but what are their radii **r**, and what angle **θ** do they cover? Well, we know their difference in radius is **w**, the wheel separation, or 8.125 eps. Using the formula for arc length **D = r θ** (**θ** here is in radians), we can write this relationship for both the inner and outer wheel, as follows:

$$\mathbf{D_{outer} = (r + w)\,\theta = r\,\theta + w\,\theta}$$
$$\mathbf{D_{inner} = r\,\theta}$$

Subtracting the bottom formula from the top one:

$$\mathbf{D_{outer} - D_{inner} = w\,\theta} \quad \textit{or}$$
$$\mathbf{\theta = (D_{outer} - D_{inner}) / w}$$

Now this is interesting: **r** completely disappeared! What this tells us is that the turn angle is a function only of the *difference in length* between the two wheel arcs (i.e. the difference in the number of encoder pulses) and the separation of the wheels. Lets see what this computes out to for our example:

$$\mathbf{\theta = (132 - 100) / 8.125}$$
$$\mathbf{= 3.938 \text{ radians}}$$
$$\mathbf{= 160.5 \text{ brads}}$$

How does this jibe with the pivoting case? In that case, the left wheel went back 16 pulses (*i.e.* forward by *negative* 16), while the right wheel went forward 16. So the difference between these is 32 (16 minus negative 16), just as it is here. And the net rotations are the same!

Suppose now that we know the angle we want to turn and want to compute the pulse difference necessary to obtain that angle. Since individual pulse counts don't matter – only their difference – let's write that difference **ΔD** instead of $\mathbf{D_{outer} - D_{inner}}$. From the above formula, then:

$$\mathbf{\theta = \Delta D / w,} \quad \textit{or}$$

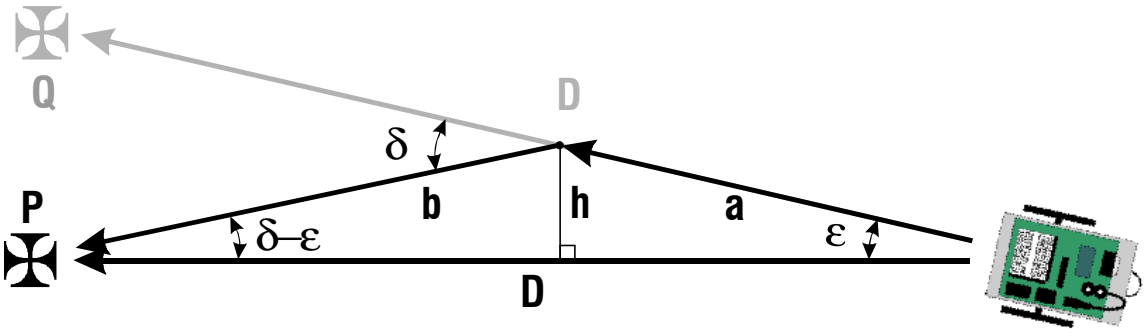
**ΔD = θ w**, *where* **θ** *is in radians, or*
**ΔD = 2π θ w / 256**, *where* **θ** *is in brads, or*
**ΔD = 0.1994 θ**, *given the dimensions of the Boe-Bot..*

So, to turn any angle **θ**, we just need to make sure that the difference in encoder pulses between wheels is **0.1994 θ**. Right?

Well, not quite. And here is where we run into our first problem. It's not even a practical one: it's theoretical. Suppose we wanted to turn 90 degrees (64 brads). That's not an unreasonable thing to want to do. So we apply the formula and come up with **ΔD = 12.76**. That's about twelve and *three-quarters* encoder pulses. But we can't measure fractional encoder pulses: they only come in whole integer amounts. So we will always have an angle error of up to ±0.5 pulses or close to ±2.5 brads. (This is to say that one encoder pulse applied to one wheel will result in a turn of about 5 brads. This is the smallest turn increment we can make.) That's a lot or error – especially if we want to aim and then travel some distance in a straight line to get to a certain point. The farther we go, the farther off the mark we get.

What to do? One obvious solution is to have more than 16 pulses per revolution – maybe even 256. Then we could count 204.16 encoder pulses for a 64-brad turn. Okay 204, with an error of 0.16 or 0.078%. That's not so bad is it? Well, no. But even at a wheel speed of 0.5 rpm, that's only about 8 ms per pulse. And that's too fast a clip to meet the goal of BASIC-Stamp-only encoder processing.

So let's examine the reasons for turning in the first place. A robot turns, mainly, to point it in the direction it needs to go next. And, as long as it gets where it needs to go, a little initial angle error probably shouldn't matter. Consider the case shown below:




The Boe-Bot wants to get to the target **P**, distance **D** away. But it's unable to aim directly for it due to the coarseness of its encoders. If it followed the path it set out on, it would end up at **Q.** The angle error (greatly exaggerated here for clarity) between the direction it *wants* to go and the direction angle it *can* go is denoted as **ε** (epsilon). So what can it do? As the figure suggests, it can travel a shorter distance **a**, then turn left by an angle **δ** (delta), then proceed a distance **b** to **P**.

Now **δ** is chosen ahead of time to be the smallest increment the Boe-Bot can turn (*i.e.* one encoder pulse forward in the right wheel, in this case; the left wheel remaining still). The trick is to calculate **a** and **b**. We assume that **ε** and **δ** are small, so that **sin ε ≈ ε**, and **sin δ ≈ δ**. Also, under this assumption, **cos ε ≈ cos δ ≈ 1**, so **a + b ≈ D**. We also assume that **ε < δ**. (We can assume this because if it were not true, we could just turn by another increment **δ** to further reduce **ε**. In fact, we can always make sure that **| ε | ≤ | δ / 2 |**. How? Well, again, if it were not true, we could turn past the direct path to **P** to come up with a smaller **ε** below that path.) Given these assumptions, we see that

$\mathbf{h = b \sin(\delta - \varepsilon) = a \sin \varepsilon}$ , *so*
$\mathbf{b\,(\delta - \varepsilon) - a\,\varepsilon = 0}$ . *And*
$\mathbf{b + a = D}$ .

Solving the above simultaneously for **a** and **b**, we get

$\mathbf{a = (\delta - \varepsilon)\,D / \delta}$ , *and*
$\mathbf{b = \varepsilon\,D / \delta}$ .

The following table illustrates the effectiveness of this path correction method. In this example, we let **D** equal 200 eps, and assume the value for $\delta$ that we derived above (*i.e.* 5 brads). We also assume that both **a** and **b** are integers, since that's what got us into this mess to begin with! We chart the original distance error $|\,\mathbf{P - Q}\,|$, versus the distance error after the correction, both given in eps:

| ε / δ | a | b | Original Error | Corrected Error |
|---|---|---|---|---|
| 0/16 | 200 | 0 | 0.000 | 0.000 |
| 1/16 | 187 | 13 | 1.540 | 0.109 |
| 2/16 | 175 | 25 | 3.080 | 0.166 |
| 3/16 | 162 | 38 | 4.620 | 0.240 |
| 4/16 | 150 | 50 | 6.160 | 0.284 |
| 5/16 | 137 | 63 | 7.700 | 0.332 |
| 6/16 | 125 | 75 | 9.239 | 0.356 |
| 7/16 | 112 | 88 | 10.779 | 0.379 |
| 8/16 | 100 | 100 | 12.318 | 0.379 |
| 9/16 | 87 | 113 | 13.857 | 0.378 |
| 10/16 | 75 | 125 | 15.396 | 0.356 |
| 11/16 | 62 | 138 | 16.935 | 0.331 |
| 12/16 | 50 | 150 | 18.473 | 0.284 |
| 13/16 | 37 | 163 | 20.012 | 0.238 |
| 14/16 | 25 | 175 | 21.550 | 0.166 |
| 15/16 | 12 | 188 | 23.087 | 0.107 |

The improvement in the final position accuracy is obvious.

### *Coordinated Motion*

Given a certain number of encoder pulses for each wheel to move, how then does one coordinate the two wheels to achieve an accurate turn, straight-line travel, or arced path? Without encoders, one would simply assign a velocity to each wheel in proportion to the distance it has to cover and run both wheels together for the right amount of time to cover those distances. If this were, in reality, a reasonable thing to do, we wouldn't need encoders! So, given marginally predictable wheel velocities and our encoder pulses, how do we enforce precision motion?

Let **D(L)** be the total distance we want the left wheel to move, and **D(R)** be the total distance for the right wheel. The directions are unimportant. We just want to make sure that both motions start and stop at the same time and proceed at constant rates. During the wheels' movement, let's keep track of how many encoder pulses each wheel has left before it can stop. We'll call these numbers **C(L)** and **C(R)** for the left count and right count, respectively. At the beginning of movement we know that

**C(L) = D(L)**, and **C(R) = D(R)**. At then end we want **C(L) = C(R) = 0**. And during motion, whenever we get a pulse from one of the wheels, we'll decrement its count **C** by one.

The Boe-Bot's wheels are (conveniently) driven by RC servos modified for continuous rotation. We won't belabor their operation here, since that's covered in Andy Lindsay's book, ***Robotics with the Boe-Bot*** (Parallax stock number 28155). Suffice it to say that the servos are driven by discrete pulses and that the speed of rotation is some function of the pulse width. As long as the servo is fed pulses, it will continue to rotate. When the pulses stop, the servo will stop.

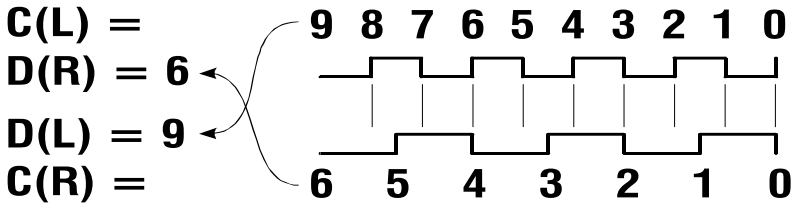
Through a procedure of calibration, it should therefore be possible to associate a certain desired velocity with a corresponding pulse width, subject to external limitations such as loading, battery reserve, and the like. As long as we provide a continuous train of such pulses to the servo, it will continue to turn at or near the desired speed. But if we leave out a pulse here and there, the servo will falter, and its net speed will decrease. So long as we don't do this too often, its motion will still appear smooth.

So here's the plan:

1. Find pulse widths that correspond roughly to the wheel velocities we're trying to achieve.
2. Drive each wheel with a continuous stream of such pulses, all the while monitoring the encoders.
3. If one a wheel gets proportionately ahead of the other one, leave out pulses to retard its motion until the other wheel catches up.
4. Continue until both wheels expend their allotted encoder counts.

How will we know if one wheel is ahead of the other – particularly when their total amounts to turn are different?

Consider the following scenario. One wheel has a total distance 1.5 times the other. The ideal encoder pulse strings would look like the those in the illustration:




(Notice that we actually count pulses on each edge. That way, given a wheel with eight holes and some sort of photodetector, we can get the 16 counts per revolution alluded to earlier.) At each position along the pulse trains, observe the two products, **C(L) · D(R)** and **C(R) · D(L)**. At the very beginning, both products equal 54, When the first edge on **L** comes along, **C(L) · D(R)** = 8 · 6 = 48. Then when the first edge on **R** occurs, **C(R) · D(L)** = 5 · 9 = 45. Next comes an edge on **L**, and **C(L) · D(R)** = 7 · 6 = 42. Then come simultaneous edges on **L** and **R**, whereupon **C(L) · D(R)** = 6 · 6 = 36, and **C(R) · D(L)** = 4 · 9 = 36. So we have a steady progression that continues to zero: 54, 48, 45, 42, 36, … , 0. Moreover at each point along this progression, **C(L) · D(R)** roughly equals **C(R) · D(L).** In fact, whenever **L** and **R** have simultaneous edges, the two products are *exactly* equal.

It would make sense then that if we can *keep* these products equal, we can maintain coordination between the two wheels. So here's the plan (referring to step 3. above):

3a. Monitor the two wheels' encoder outputs. For each one that changes, decrement its counter.

3b. Since we're trying to decide whether to pulse a particular wheel's servo or not, assume that if we do, we'll get another encoder edge next time we check; and if we don't, we won't. What we want to check is whether doing so will put one wheel ahead of the other *on the next step*.

3b. Compare each wheel's count, *minus 1* (looking ahead to the what-ifs, remember), multiplied by the total distance for the *other* wheel with the similar product for the other wheel. If it's significantly lower, withhold the wheel's pulse for the current interval. In math parlance, this is:

**(C(Wheel) – 1) · D(Other wheel) << (C(Other wheel) – 1) · D(Wheel)**

("**<<**" here means "significantly less than", not "shift left".) Okay, so what does "significantly less than" mean? This is an important question because, if we had just said "less than", we'd get pulses withheld rather frequently, resulting in jerky motion. One way to solve this is to add a fudge factor to the left-hand side and replace "<<" by "<". The fudge factor we choose should answer the question, "How much is **C(Wheel)** allowed to vary without being ahead of the other one?" The answer, because we're dealing with integers is, "½". If the anticipated count is ahead of the correct count by no more than one-half an encoder interval, it's safe to insert a servo pulse. We can't really do any better than that!

So, we can add ½ to **C(Wheel)** above and rewrite the equation thus:

**(C(Wheel) + ½ – 1) · D(Other wheel) < (C(Other wheel) – 1) · D(Wheel)** , *or*
**(C(Wheel) – ½) · D(Other wheel) < (C(Other wheel) – 1) · D(Wheel)** , *or*
**C(Wheel) · D(Other wheel) + D(Wheel) < C(Other wheel) · D(Wheel) + D(Other wheel) / 2**

When this condition holds for **Wheel**, we withhold it's servo pulse.

### *Ramping*

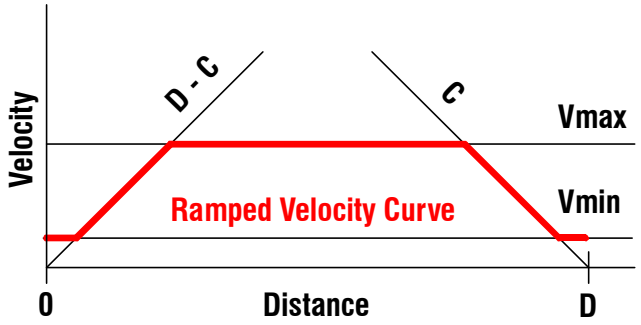
Inertia is a fact of life. "A body at rest will remain at rest, and a body in motion will remain in motion, unless acted upon by another force," said Sir Isaac. And this, of course, applies to robots and robot wheels. As Andy Lindsay points out in ***Robotics with the Boe-Bot***, starting or stopping motion without gradual acceleration and deceleration is not only jarring to the servo's internal mechanisms but wastes precious battery energy. What's even worse for us here is that running the servos until the counts reach zero, then simply ceasing to send servo pulses, *will not stop the wheel instantly*. Inertia will keep it moving a little, possibly for another servo pulse or two, completely messing up the precision we set out to achieve. Fortunately, ramping can solve the problem. Here's how to do it:

1. Figure out what the maximum velocity will be. Call it **Vmax**.
2. Determine a minimum velocity. Call it **Vmin**. This will be the starting and ending velocity. It could be zero. But sometimes, for snappier performance, setting it to a slightly higher value helps. Just be sure it's low enough that stopping at that velocity won't result in extraneous encoder pulses!
3. At each step in the servo pulsing process, compute the following for the wheel that's turning the farthest:

   **C**, the number of counts remaining, and
   **D – C**, the overall distance, *minus* the number of counts remaining.

4. Pick the *lowest* value among the above two figures and **Vmax**. This becomes the candidate instantaneous velocity value. If it's less than **Vmin**, use **Vmin** instead.

The following graph illustrates this principle:



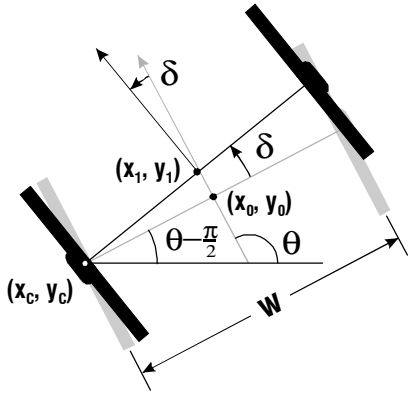

Purists may note that we're comparing apples and oranges here – velocities and distances. In reality, there's a scaling factor involved, which is the time **Δt** between the encoder pulses. So when we say "**C**", for example, what we mean is "**C / Δt** ". But when it comes down to programming this curve, we just scale things so **Δt = 1**. Super-duper purists (you know who you are) will further note that, during ramping, **Δt** isn't even a constant, and they're right. If the X-axis in the above curve were **Time** instead of **Distance**, the ramped portions wouldn't be straight lines, but rather more parabolic in shape. That's one reason for picking a **Vmin** > 0. It helps to avoid that *painfully* slow parabolic "wallow" near zero.

Once the instantaneous velocity for the fastest wheel is determined, the velocity for the other one can be made proportional to it.

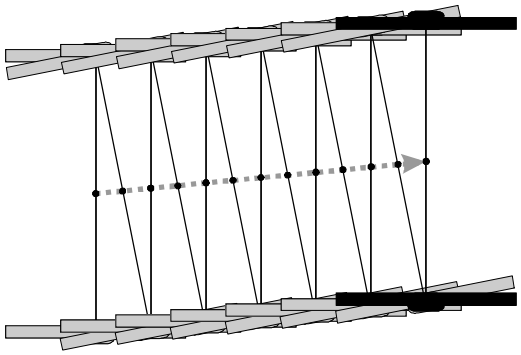
### *Wheel Odometry*

Up to this point, we've dealt with coordinated motion, using the encoders as part of a feedback loop to control the servos. But what if the servos are being controlled by other means (*e.g.* in response to mechanical feelers)? Is there still some way to keep track of the bot's location? Yes, so long as we know the *direction* each wheel is turning, we can infer the Boe-Bot's trajectory across the floor from the encoder outputs.

In the simplest case, look what happens when the right wheel moves forward by one ep, leaving the left wheel anchored in place (**δ** here is greatly exaggerated).



The Boe-Bot's center starts at $(x_0, y_0)$, with the bot facing in the direction $\theta$. When the right wheel moves forward, the bot rotates left on its left wheel by an angle $\delta$ to its new location, $(x_1,y_1)$, and new direction, $\theta + \delta$. It would be nice to treat the two wheels independently, wherein each encoder pulse from each wheel contributes to a change in position and direction without regard for what the other wheel might be doing at the same time. We can think of each encoder pulse from one wheel as representing a rotation by $\delta$ about the other wheel. By considering the left and right wheels separately, straight-line motion becomes nothing more than a sequence of left-right steps – almost as if the Boe-Bot were walking instead of rolling, as the following illustration suggests:



The illustration shows another consequence of this point of view: the tendency of the computed path to "crab" in a direction opposite the side that took the first step. If both wheels were turning simultaneously, this would not be happening. Even though the angle of each step is exaggerated here, it is still something we need to address later on.

Going back to the previous diagram, high-school trig tells us that

$$x_{0R} = x_{cR} + w/2 \cos(\theta_{0R} - \pi/2) = x_{cR} + w/2 \sin \theta_{0R}$$
$$y_{0R} = y_{cR} + w/2 \sin(\theta_{0R} - \pi/2) = y_{cR} - w/2 \cos \theta_{0R}$$
$$x_{1R} = x_{cR} + w/2 \cos(\theta_{0R} - \pi/2 + \delta) = x_{cR} + w/2 (\sin \theta_{0R} \cos \delta + \cos \theta_{0R} \sin \delta)$$
$$y_{1R} = y_{cR} + w/2 \sin(\theta_{0R} - \pi/2 + \delta) = y_{cR} + w/2 (\cos \theta_{0R} \sin \delta - \sin \theta_{0R} \cos \delta)$$
$$\theta_{1R} = \theta_{0R} + \delta$$

Solving the above for $x_1$ and $y_1$ in terms of $x_0$ and $y_0$, to eliminate $x_c$ and $y_c$, we get

$$x_{1R} = x_{0R} + w/2 \left[\cos \theta_{0R} \sin \delta - \sin \theta_{0R} (1 - \cos \delta)\right]$$
$$y_{1R} = y_{0R} + w/2 \left[\sin \theta_{0R} \sin \delta + \cos \theta_{0R} (1 - \cos \delta)\right]$$
$$\theta_{1R} = \theta_{0R} + \delta$$

Similarly, for a single-step forward movement of the left wheel while keeping the right one still, we get

$$x_{1L} = x_{0L} + w/2 \left[\cos \theta_{0L} \sin \delta + \sin \theta_{0L} (1 - \cos \delta)\right]$$
$$y_{1L} = y_{0L} + w/2 \left[\sin \theta_{0L} \sin \delta - \cos \theta_{0L} (1 - \cos \delta)\right]$$
$$\theta_{1L} = \theta_{0L} - \delta$$

What happens if we take a step with the right wheel, followed by one with the left? Substituting the expressions for $x_{1R}$ and $y_{1R}$ into $x_{0L}$ and $y_{0L}$ above, and $\theta_{1R} = \theta_{0R} + \delta$ for $\theta_{0L}$, we get (after drawing the curtain and simplifying)

$$x_{1L} = x_{0R} + w\ [\cos \theta_{0R} \sin \delta - \sin \theta_{0R} (1 - \cos \delta)]$$
$$y_{1L} = y_{0R} + w\ [\sin \theta_{0R} \sin \delta + \cos \theta_{0R} (1 - \cos \delta)]$$
$$\theta_{1L} = \theta_{0R}$$

These look a lot like the equations for $x_{1R}$ and $y_{1R}$ above, except that **w/2** is now **w**. This means that, in terms of distance traveled, this left-right sequence is identical to taking a *single step* with the right wheel alone at double the distance.

Okay, so what happens if we take a step with the left wheel, followed by one with the right? Doing a similar substitution as we did above, we find:

$$x_{1R} = x_{0L} + w\ [\cos \theta_{0L} \sin \delta + \sin \theta_{0L} (1 - \cos \delta)]$$
$$y_{1R} = y_{0L} + w\ [\sin \theta_{0L} \sin \delta - \cos \theta_{0L} (1 - \cos \delta)]$$
$$\theta_{1R} = \theta_{0L}$$

Notice that the result is the same, except for the signs of the $(1 - \cos \delta)$ terms.

Now suppose both wheels *had* moved at once. There's no direction change, just movement by an amount $w\delta$ in the direction $\theta$, so

$$x_{1LR} = x_{0LR} + w\delta \cos \theta_{0LR}$$
$$y_{1LR} = y_{0LR} + w\delta \sin \theta_{0LR}$$
$$\theta_{1LR} = \theta_{0LR}$$

Since $\delta$ is very small, $\sin \delta \approx \delta$, so we could write the above as

$$x_{1LR} = x_{0LR} + w \cos \theta_{0LR} \sin \delta$$
$$y_{1LR} = y_{0LR} + w \sin \theta_{0LR} \sin \delta$$
$$\theta_{1LR} = \theta_{0LR}$$

Look familiar? It's the same as the right-then-left and the left-then-right formulae, but without the $(1 - \cos \delta)$ terms. They are these terms that lead to the crabbing noted above, and the direction of the crabbing depends on which side takes the first step.

But again, since $\delta$ is small, $\cos \delta \approx 1$, so $1 - \cos \delta \approx 0$. Perhaps, we could just eliminate this term altogether. That way the left-right and right-left equations for position would be the same. Only the angle equations would be different. Let's write this now as functions of the vector $[x, y, \theta]$ and the increment $\delta$. These functions will return a vector with the new values of **x**, **y**, and $\theta$.

$$L([x, y, \theta], \delta) = [x + w/2 \cos \theta \sin \delta,\ y + w/2 \sin \theta \sin \delta,\ \theta + \delta]$$
$$R([x, y, \theta], \delta) = [x + w/2 \cos \theta \sin \delta,\ y + w/2 \sin \theta \sin \delta,\ \theta - \delta]$$

So does this fix the left-right and right-left discrepancy? Well, not exactly. For the right-left case we get, after cranking out the derivation:

$$L(R([x, y, \theta], \delta), \delta)$$
$$= [x + w/2 \sin\delta (\cos\theta (1 + \cos\delta) - \sin\theta \sin\delta),$$
$$y + w/2 \sin\delta (\sin\theta (1 + \cos\delta) + \cos\theta \sin\delta),$$
$$\theta ]$$
$$\approx [x + w \cos\theta \sin\delta - w/2 \sin^2\delta \sin\theta,$$
$$y + w \sin\theta \sin\delta + w/2 \sin^2\delta \cos\theta,$$
$$\theta ]$$ , by using the approximation $\cos\delta \approx 1$.

And for the left-right case,

$$R(L([x, y, \theta], \delta), \delta)$$
$$= [x + w/2 \sin\delta (\cos\theta (1 + \cos\delta) + \sin\theta \sin\delta),$$
$$y + w/2 \sin\delta (\sin\theta (1 + \cos\delta) - \cos\theta \sin\delta),$$
$$\theta ]$$
$$\approx [x + w \cos\theta \sin\delta + w/2 \sin^2\delta \sin\theta,$$
$$y + w \sin\theta \sin\delta - w/2 \sin^2\delta \cos\theta,$$
$$\theta ]$$ , by using the approximation $\cos\delta \approx 1$.

These look like the equations for simultaneous motion, except for terms containing $\sin^2\delta$ this time, differing in sign between the two. And these, once again, are crabbing terms.

It would seem, then, that our goal of treating the wheels independently is unobtainable. Perhaps we should relax that requirement just slightly. If we keep track of which side moved last, maybe we can invoke a slightly formula if the opposite side moves next, in order to undo the crabbing. For example, let's say the right side moves first then the left. But this time, *before* we apply the formulae for **x** and **y**, let's compute the new value of $\theta$ (which rolls it back to its previous value) and use it instead of the current one. So let's define new functions in which the **x** and **y** terms are computed with the *new* value of $\theta$ (*i.e.* $\theta$ is computed first, *then* **x** and **y**).

$$L'([x, y, \theta], \delta) = [x + w/2 \cos(\theta + \delta) \sin\delta,\ y + w/2 \sin(\theta + \delta) \sin\delta,\ \theta + \delta]$$
$$R'([x, y, \theta], \delta) = [x + w/2 \cos(\theta - \delta) \sin\delta,\ y + w/2 \sin(\theta - \delta) \sin\delta,\ \theta - \delta]$$

Now, when we apply **R** then **L'**, or **L** then **R'**, we get

$$L'(R([x, y, \theta], \delta), \delta) = [x + w \cos\theta \sin\delta,\ y + w \sin\theta \sin\delta, \theta]$$
$$R'(L([x, y, \theta], \delta), \delta) = [x + w \cos\theta \sin\delta,\ y + w \sin\theta \sin\delta, \theta]$$

But this is identical to the result we got when both wheels moved at once, and the crabbing terms are gone! So far, so good.

Up until this point, we've assumed that $\delta$ was always in the same direction. But what happens when we move forward with one wheel then back, or forward with one wheel and back with the other one? In the former case, we ought to end up right back where we started. In the latter case, **x** and **y** should be the same but $\theta$ should increment or decrement by $2\delta$. To summarize, the following table presents the results obtained by various combinations of **L**, **R**, **L'**, **R'**, $\delta$, and $-\delta$. The shaded entries are the preferred calculations for the given situation. (The small-angle approximation $\cos\delta \approx 1$ is used throughout for simplification. However, it was not necessary for the shaded formulae.)

| Description | Function Application | Result Vector |
|---|---|---|
| Move both wheels forward. | **L(R([x, y, θ]), δ), δ)**<br>*or*<br>**L'(R'([x, y, θ]), δ), δ)** | **$x + w \cos\theta \sin\delta - w/2 \sin^2\delta \sin\theta$,**<br>**$y + w \sin\theta \sin\delta + w/2 \sin^2\delta \cos\theta$,**<br>**θ** |
| | **R(L([x, y, θ]), δ), δ)**<br>*or*<br>**R'(L'([x, y, θ]), δ), δ)** | **$x + w \cos\theta \sin\delta + w/2 \sin^2\delta \sin\theta$,**<br>**$y + w \sin\theta \sin\delta - w/2 \sin^2\delta \cos\theta$,**<br>**θ** |
| | **L(R'([x, y, θ]), δ), δ)** | **$x + w \cos\theta \sin\delta - w \sin^2\delta \sin\theta$**<br>**$y + w \sin\theta \sin\delta + w \sin^2\delta \cos\theta$**<br>**θ** |
| | **R(L'([x, y, θ]), δ), δ)** | **$x + w \cos\theta \sin\delta + w \sin^2\delta \sin\theta$**<br>**$y + w \sin\theta \sin\delta - w \sin^2\delta \cos\theta$**<br>**θ** |
| | **L'(R([x, y, θ]), δ), δ)**<br>*or*<br>**R'(L([x, y, θ]), δ), δ)** | **$x + w \cos\theta \sin\delta$,**<br>**$y + w \sin\theta \sin\delta$,**<br>**θ** |
| Move right wheel forward, then back. | **R(R([x, y, θ]), δ), –δ)** | **$x + w/2 \sin^2\delta \sin\theta$**<br>**$y - w/2 \sin^2\delta \cos\theta$**<br>**θ** |
| | **R'(R'([x, y, θ]), δ), –δ)** | **$x - w/2 \sin^2\delta \sin\theta$**<br>**$y + w/2 \sin^2\delta \cos\theta$**<br>**θ** |
| | **R'(R([x, y, θ]), δ), –δ)**<br>*or*<br>**R(R'([x, y, θ]), δ), –δ)** | **x**<br>**y**<br>**θ** |
| Move right wheel forward, then left wheel back. | **L(R([x, y, θ]), δ), –δ)**<br>*or*<br>**L'(R'([x, y, θ]), δ), –δ)** | **$x + w/2 \sin^2\delta \sin\theta$**<br>**$y - w/2 \sin^2\delta \cos\theta$**<br>**$\theta + 2\delta$** |
| | **L'(R([x, y, θ]), δ), –δ)** | **$x + w (\sin^2\delta \sin\theta + \sin^3\delta \cos\theta)$**<br>**$y - w (\sin^2\delta \cos\theta - \sin^3\delta \sin\theta)$**<br>**$\theta + 2\delta$** |
| | **L(R'([x, y, θ]), δ), –δ)** | **x**<br>**y**<br>**$\theta + 2\delta$** |
| Move left wheel back, then right wheel forward. | **R(L([x, y, θ]), –δ), δ)**<br>*or*<br>**R'(L'([x, y, θ]), –δ), δ)** | **$x - w/2 \sin^2\delta \sin\theta$**<br>**$y + w/2 \sin^2\delta \cos\theta$**<br>**$\theta + 2\delta$** |
| | **R'(L([x, y, θ]), –δ), δ)** | **$x - w (\sin^2\delta \sin\theta + \sin^3\delta \cos\theta)$**<br>**$y + w (\sin^2\delta \cos\theta - \sin^3\delta \sin\theta)$**<br>**$\theta + 2\delta$** |
| | **R(L'([x, y, θ]), –δ), δ)** | **x**<br>**y**<br>**$\theta + 2\delta$** |

What this table tells us is this:

1. If two wheels are moving in the same direction, apply the normal function for the first one and the reverse function for the second.
2. If one wheel moves one direction then the other, apply the normal function to the first movement and the reverse function to the second.
3. If two wheels are moving in opposite directions, apply the reverse function to the first one and the normal function to the second.
4. In all other cases, apply the normal function.

Number three is a problem, though. How are we to know when the first wheel's encoder pulse comes along what's going to happen next? In all other cases, we'd just apply the normal function to the first in a two-part sequence and wait to see what happens next. But rule three is an exception requiring either putting the movement in abeyance until the next pulse comes along or applying the normal function with the option of reversing it later. A third option would be to ignore the small error that results from applying the normal function first and double checking that it doesn't accumulate. Because the last choice has the additional benefit of simplifying the rules considerably, we'll choose it with the option of modifying it if errors accumulate. So the rules can be restated as:

1. If different wheels are moving in the same direction, or if the same wheel changes direction, apply the normal function for the first movement and the reverse function for the second.
2. In all other cases, apply the normal function.

The discussion to this point assumes we have some means to calculate all these sines, cosines, and fancy multiplications. If the BASIC Stamp had floating-point transcendental functions, yes, that part would be a cinch. But with only its integer math, and **SIN** and **COS** functions that work with byte values, we could be in trouble.

All is not lost, however. Instead of trying to *calculate* sines and cosines from **θ** and **θ ± δ**, we'll just start with known values for **sin θ** and **cos θ** and update them directly whenever the angle changes. The following identities will help:

$$\cos(\theta + \delta) = \cos\theta\cos\delta - \sin\theta\sin\delta$$
$$\sin(\theta + \delta) = \sin\theta\cos\delta + \cos\theta\sin\delta$$
$$\cos(\theta - \delta) = \cos\theta\cos\delta + \sin\theta\sin\delta$$
$$\sin(\theta - \delta) = \sin\theta\cos\delta - \cos\theta\sin\delta$$

Now **sin δ** and **cos δ** are constants for any given Boe-Bot. Once they're determined through some sort of calibration procedure, we can just replace them with named constants, say, **SD** and **CD**. In a similar vein, **sin θ** and **cos θ** are now just variables, which we'll call **Ydir** and **Xdir**, for Y-direction and X-direction. To update **Ydir** and **Xdir,** for example, when we add **δ** to **θ**, we might use BASIC statements like:

```
Ydir = (Ydir ** CD) + (Xdir ** SD)
Xdir = (Xdir ** CD) - (Ydir ** SD)
```

This is an oversimplification, of course, since we need to pay attention to details like scaling and negative values. The other concern, since we're using integer math, is whether **Ydir** and **Xdir** will

start to diverge from the unit circle after many repeated calculations. One way to guarantee this won't happen is to compute

$$\mathbf{Xdir} = \sqrt{\mathbf{r}^2 - \mathbf{Ydir}^2}$$

(where **r** is the radius of the circle used to account for integer scaling of **Ydir**), instead of using the original formula for **Xdir**. Unfortunately, to do this with the precision needed would require 32-bit math, and that's a little out of reach on the Stamps.

But not to despair. There's a rather interesting algorithm from the annals of computer graphics for computing points on a circle. It goes like this:

$$\mathbf{Xdir_p} = \mathbf{Xdir_0} - \mathbf{Ydir_0} \cdot \mathbf{m/(2n)}$$
$$\mathbf{Ydir_1} = \mathbf{Ydir_0} + \mathbf{Xdir_p} \cdot \mathbf{m/n}$$
$$\mathbf{Xdir_1} = \mathbf{Xdir_p} - \mathbf{Ydir_1} \cdot \mathbf{m/(2n)}$$

These three computations are applied *in sequence*, using integer math. The remarkable thing about them is that for certain initial conditions and values for **m** and **n** (m < n), **Xdir** and **Ydir** do not accumulate errors, staying on a circle centered at the origin. An additional nice feature of doing the computations in sequence is that no auxiliary variables are needed for intermediary results. $\mathbf{Xdir_0}$, $\mathbf{Xdir_p}$, and $\mathbf{Xdir_1}$ can all be the same variable, as can $\mathbf{Ydir_0}$ and $\mathbf{Ydir_1}$.

The following Stamp program implements this algorithm. Try it out with the values provided (and with other values) to see how well it keeps from wandering off.

```
'{$STAMP BS2}
'{$PBASIC 2.5}

R    CON 4096
M    CON 51448   'As used here, M/N = 51488/(65536 * 4) = 0.1964
N    CON 4
P    CON 129

Xdir VAR Word
Ydir VAR Word
i    VAR Word

Xdir = R
FOR i = 1 TO 250
  Xdir = Xdir - ((1 - (Ydir.BIT15 << 1)) * (ABS(Ydir) ** M / N >> 1))
  Ydir = Ydir + ((1 - (Xdir.BIT15 << 1)) * (ABS(Xdir) ** M / N))
  Xdir = Xdir - ((1 - (Ydir.BIT15 << 1)) * (ABS(Ydir) ** M / N >> 1))
  DEBUG SDEC Xdir, " ", SDEC Ydir, CR
NEXT
```

If we write **m**/**n** as a single constant $\delta$ (rather suggestively), **Xdir** as **C**, and **Ydir** as **S**, the above sequence becomes

$$\mathbf{C_p} = \mathbf{C_0} - \mathbf{S_0} \cdot \delta / 2$$
$$\mathbf{S_1} = \mathbf{S_0} + \mathbf{C_p} \cdot \delta$$
$$\mathbf{C_1} = \mathbf{C_p} - \mathbf{S_1} \cdot \delta / 2$$

Now let's see what we get when we solve for $\mathbf{C_1}$ and $\mathbf{S_1}$ in terms of $\mathbf{C_0}$ and $\mathbf{S_0}$ only:

$$C_1 = C_0 \cdot (1 - \delta^2/2) - S_0 \cdot (\delta - \delta^3/4)$$
$$S_1 = S_0 \cdot (1 - \delta^2/2) + C_0 \cdot \delta$$

For small $\delta$, $\sin \delta \approx \delta$. In fact, an even better approximation for $\sin \delta$ is $\delta - \delta^3/6$. Again, for small $\delta$, $\cos \delta \approx (1 - \delta^2/2)$. Just to see how good these approximations are, let's use a typical value from the Boe-Bot. For the Boe-Bot, it takes about 50 encoder pulses from one wheel to spin in a full circle around the other one. So let's say $\delta = 2\pi / 50$ or **0.12566**. Now compare values:

| $\delta$ | $\delta - \delta^3/4$ | $\delta - \delta^3/6$ | $\sin \delta$ | $(1 - \delta^2/2)$ | $\cos \delta$ |
|---|---|---|---|---|---|
| 0.12566 | 0.12516 | 0.12533 | 0.12532 | 0.99210 | 0.99212 |

This isn't half bad, so it's not too far-fetched to write the above equations as

$$\begin{aligned} C_1 &= C_0 \cdot \cos \delta - S_0 \cdot \sin \delta \\ &= r \cos \theta \cos \delta - r \sin \theta \sin \delta \\ &= r \cos(\theta + \delta) \end{aligned}$$

$$\begin{aligned} S_1 &= S_0 \cdot \cos \delta + C_0 \cdot \sin \delta \\ &= r \sin \theta \cos \delta + r \cos \theta \sin \delta \\ &= r \sin(\theta + \delta) \end{aligned}$$

where $\theta$ is the angle formed by $x = C$ and $y = S$ on the circle about the origin with radius **r**. So we've come full circle (so to speak). Starting from a simple, stable, integer-only algorithm, we see that it's equivalent to the more complex sum-of-angles sine and cosine identities.

So now we have methods we can use in the BASIC Stamp to compute not only the Boe-Bot's **x** and **y** position coordinates, but also its direction $\theta$ as expressed by the cosine and sine factors **C** and **S**. Once $\delta$ is determined for a given Boe-Bot, all that remains is finding integer values for **r**, **m**, and **n** that not only keep **C** and **S** confined to a circle but maintain the relationship $\delta = m/n$. A sample BASIC Stamp program that implements these findings in given in the last section.

## Practical Constraints

### *Sources of Error*

As is always the case, a good dose of theory is seldom sufficient to deal with the real world. And this is certainly true in the case of robot wheel encoders. Up to this point, we've been talking about the Boe-Bot as if it were some idealized creature existing only on graph paper. Such a creature would have super hard, infinitesimally skinny wheels, each touching the rolling surface at a single point. They would have infinite friction in the rolling direction and zero turning friction about the point of contact. Plus, the wheels would be *exactly* the same size, and their axles would align perfectly. And finally, they would roll only on super hard, perfectly flat, smooth, level surfaces. Determining the two measurements needed for accurate navigation – the wheels' diameter and the distance between them – would thus be an exercise simply in counting decimal places.

But the observable universe doesn't afford such luxuries. Any wheeled robot will have a tread to grip the rolling surface. With that tread comes a finite width, and with that width comes uncertainty about just how far apart the wheels are. What's worse, if the wheels wobble a little, that uncertainty can vary with rotational position. Also, because the tread may be soft (or worse yet, if the rolling surface

is soft or has pile like a carpet), the effective wheel diameter carries its own uncertainties. Add to that the fact that when a robot turns, each wheel must skid a little about some point of rotation. It's the distance between the points of rotation of the two wheels that determine their effective separation. With a wide, grippy tread – even on a hard, smooth surface – these points can vary, depending upon how sharp the turn is.

Keeping track of position and direction through encoders is known as "wheel odometry", and this is just a fancy way of saying "dead reckoning". Under dead reckoning, navigation is done without external references. And any errors that slip into the system have a way of accumulating. Without some external reference from which to obtain periodic corrections, dead reckoning will, over time, exhibit larger and larger position and orientation errors. Does that mean it's not useful? No, of course not! The Mars rovers employ wheel odometry, along with rate gyros (another aid to dead reckoning) – to keep track of their short-term positions on the Martian surface. But they also employ their onboard cameras to maintain visual references for correcting the odometry errors. A properly-implemented and calibrated odometry system can easily provide good short-term navigation by itself. When corrected by periodic external feedback, it can perform well in the long-term as well.

### *Calibration*

The key to success in any odometry system is calibration. We could examine and analyze the sources of deterministic error and come up with theoretical values for the effective wheel diameter and separation in a given environment. *Or*, we could just make some good assumptions, plop the Boe-Bot down and see how it does. If it's off, refine the assumptions and try again. What, then, would be a good test? For distance, it's easy. Just move forward by 100 eps, and measure to see how far that is in real-world units. Divide by 100, and we have our inches-per-ep, millimeters-per-ep, or furlongs-per-ep figure. If we wanted, we could then back-calculate to see what the *effective* wheel diameter is. The formula is:

**Effective_diameter = 16 · Distance_per_ep / π**

The same applies to angular calibration. If we could determine the effective diameter and separation of the wheels, we could plug these figures into the control program, *et voilá*, a calibrated robot. An easier approach is to guess at a figure that combines these two measurements, *i.e.* the *eps-per-full-turn* figure, or the number of encoder pulses needed from one wheel, while the other remains still, to effect a full 256-brad turn. Then, plug this figure into the control program and send the Boe-Bot over a square course that returns to its starting point. If the Boe-Bot comes up short, we know our eps-per-full-turn figure was too low, *i.e.,* it didn't really turn as far as we thought it would. If it overshoots the starting point, we know the figure was too high. Either way, we can adjust it until the Boe-Bot returns exactly to its starting point. Again, if we wanted to, we could then use the effective wheel diameter we calculated before, combine it with the eps-per-full-turn figure, and come up with the effective wheel separation. Here's the formula:

**Effective_separation = Eps_per_full_turn · Distance_per_ep / (2π)**

To take advantage of the calibration, of course, we really don't need to calculate either of these two values. But it might just be interesting to see how close they are to measurements taken with calipers or a ruler!

Another, more accurate and less time-consuming method than trial and error for determining the eps-per-full-turn figure, involves an optical sensor. Let's say we outfit the Boe-Bot with such a sensor, point the sensor forward, and place the bot on a smooth, level surface in a visually "interesting"

environment. We now rotate the Boe-Bot one ep at a time. While doing so, we can record the amount of light the Boe-Bot "sees" at each angle that it points. When it returns to the start position, after making one full revolution, it should see the same light level it saw when it started. By recording several full rotations, we should be able to obtain a periodic waveform that repeats every **t** readings, where **t** is the (possibly fractional) number of encoder pulses in a full rotation.

To determine the number **t** accurately, we use a statistical method called *autocorrelation*. This is just a special application of the normal correlation coefficient. That coefficient is a calculation performed on a set of number pairs to see if the first numbers of the pairs rise or fall in concert with the second numbers. It is given by the formula:

$$\rho(x, y) = (n \Sigma x_i y_i - \Sigma x_i \Sigma y_i) / \sqrt{(n \Sigma x_i^2 - (\Sigma x_i)^2)(n \Sigma y_i^2 - (\Sigma y_i)^2)}$$

where each $(x_i, y_i)$ represents one data pair from the set of **n** pairs, and the symbol $\Sigma$ denotes a summation of the term that follows over the range **i = 1 to n**. The number $\rho$ can range from **–1** (perfect inverse correlation) to **1** (perfect correlation), with **0** representing no correlation whatsoever.

In autocorrelation, we compare a sequence of numbers to itself, offset by some amount **m**. Actually, we pick two equal-length subsequences from the main sequence and perform the above computation upon them as if they were separate. If we were to pick the *same* subsequences (*i.e.* **m = 0**, or no offset), $\rho$ would certainly equal **1**, since any given sequence correlates perfectly with itself. If the sequence repeats itself with period **t**, we can pick one subsequence starting at **0**, and the other starting at **m = t**, and we should again expect to find that $\rho = 1$ or close to it, depending on how accurately the sequence repeats.

So to find **t**, we could select several sequential values for **m** and see which one yields the highest value of $\rho$. If $\rho$ is very close to **1**, we can assume we've found the period of the waveform.

But the waveform may not have an integer period. In the Boe-Bot's case, there's no reason to assume that after any whole number of encoder pulses we're going to return *exactly* to our starting angle. To account for this, we need to resort to linear interpolation. Suppose we wanted to select an offset value of **50.7** for the second subsequence (*i.e.* **y**) in the calculation of $\rho$. To do so, we would use the following formula for each element of **y**:

$$y_i = (1 - p)\, x_{i+m} + p\, x_{i+m+1}$$

where **m** is the integer part of the offset (in this case, **50**), and **p** is the fractional part (in this case, **0.7**). This effectively yields a new subsequence that's offset from the original by a fractional amount.

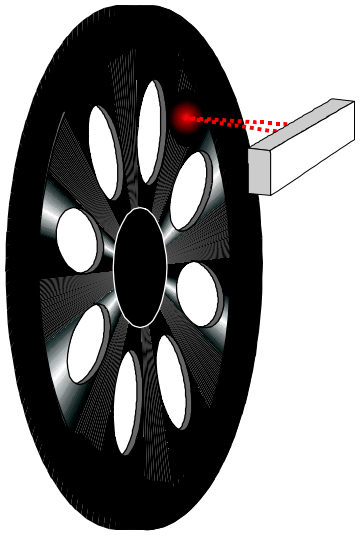
To find the period **t**, then, we first find the value of **m** that maximizes $\rho$. This is not complicated to do. We just try all of them in the neighborhood we're interested in (*i.e.* around **50**; so from **30** to **70**, say) and pick the one that gives the highest $\rho$. We know from this that the optimum fractional value will be between **m – 1** and **m + 1**. Next we try however many fractional values in between these two that our precision requirements dictate and pick the one yielding the highest $\rho$ for our value of **t**.

In the section titled "BASIC Stamp Programs", one part of the calibration program presented there records 256 light intensity values while incrementing the angle by one ep each time. Because it's not practical to calculate $\rho$ in the Stamp itself, and thus to find **t**, this is done in a separate, PC-based program.

## Encoder Hardware

The encoder hardware consists of a photoreflective sensor aimed at the inside of the Boe-Bot wheel. When the wheel is positioned so that one of the holes is in front of the sensor, no light is reflected back, and the sensor's output floats. When a portion of the wheel is present to reflect light, the sensor detects its shiny black surface and pulls its output down.

The sensor used is *modulated*. That means that it emits IR light in a high-frequency pulse stream and expects to see pulsed light reflected back. If all it sees is a constant brightness, it knows it's coming from somewhere else and won't respond. Such a sensor can be used in a wide range of ambient lighting conditions without the use of special shrouds to shield it. The sensor arrangement (pulled back from the wheel for illustrative purposes) is shown below:



Because the output floats under no-detect conditions, a 10K pullup resistor to Vdd is necessary to provide a full 5-volt swing. In all other regards, the output is completely digital, complete with hysteresis for stability.

## BASIC Stamp Programs

### *Calibration*

By combining all the theory, the practical considerations, and the hardware, we can come up with some BASIC Stamp code to drive the Boe-Bot using the encoders for feedback. The first step, though, is to calibrate the servos and the encoders. This consists of three separate operations:

1. Null the servos, so that they do not rotate when fed 1.5ms pulses.
2. Establish the relationship between pulse width and servo speed.
3. Determine the exact number of encoder pulses in a full 256-brad turn on the Boe-Bot's axis.

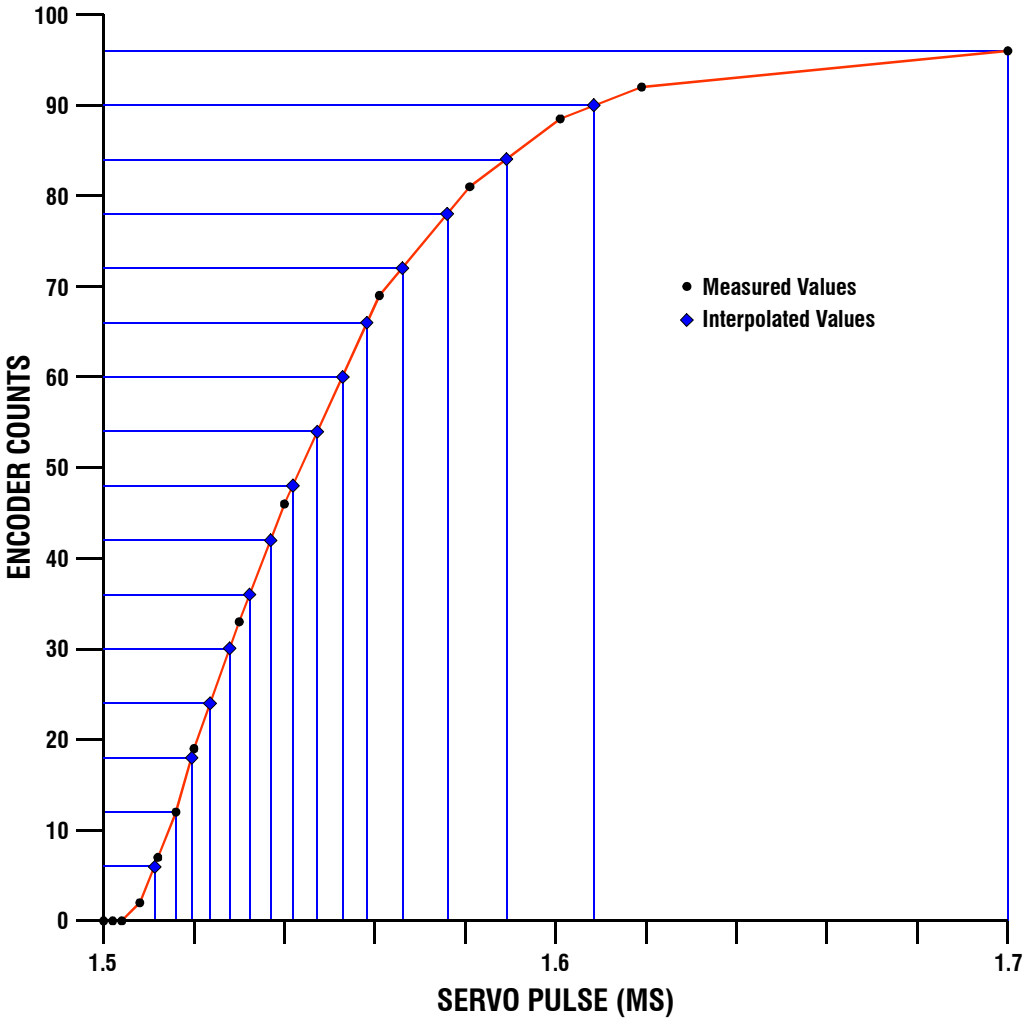
Step one is described in ***Robotics with the Boe-Bot***. The calibration program shown here has a section which generates a repeated sequence of 1.5ms pulses to both servos so they can be adjusted. However, it's rather difficult to get an adjusting tool into the holes provided once the servos are installed on the Boe-Bot, so it's best to perform this step prior to final assembly.

In step two, we have to determine for each servo – because they may differ – the correspondences between the various pulse widths and the actual rotation speeds. But we don't have to do this by hand: we've got encoders to help! The idea is first to check the servos to make sure they're properly nulled. This is done by sending each a stream of 1.5ms pulses to see if the wheels move. If they do, the encoder output will change, and the program can detect it an flag the error.

The next step is to determine a single maximum velocity that both wheels can sustain in both directions. To do this we send both servos a stream of 256 pulses of the same width, at one extreme of their pulse range. This will cause the Boe-Bot to spin around. While it's doing this, the program counts the transitions on each encoder output. Next we do the same at the other extreme., and the Boe-Bot will spin the other direction. Finally, we take the lowest of the four counts measured, and this becomes the maximum common sustainable speed.

Next, we cycle the servos through a series of pulse streams, each with 256 pulses, but each series with a different pulse width. Again, we count edges for each servo. From this data one could construct a graph for each servo of its velocity at each of the tested pulse widths. But this is not what we want.

What we really want is a graph of the pulse width for each of several equally-spaced velocities. To get these, the program uses *linear interpolation* between the points on the first graph to approximate the pulse values we need. The principle is illustrated below for one wheel in one direction:



Once the interpolated pulse widths for sixteen equally-spaced velocities are obtained in each direction for each servo, they are written to the BASIC Stamp's EEPROM in the first 64 bytes. The byte values represent offsets from the null value of 1.5ms and are scaled appropriately for the BASIC stamp version being used.

Finally, in step three, we bring one of the photoresistors included with the Boe-Bot to bear on the angle calibration. The data thus obtained can later be subjected to an autocorrelation analysis, as discussed in the theory section. To perform the angle calibration, one photoresistor should be wired as shown in ***Robotics with the Boe-Bot***, page 194, to pin 6. But substitute a 0.1µF capacitor (marked "104") for the .01µF capacitor shown. We can do this because we don't need to monitor the photoresistor while the servos are running and can therefore measure longer discharge times. The photoresistor's leads should be bent so it's aiming forward. When recording the angle data, be sure to place the Boe-Bot on the same flat, smooth surface you'll be running it on. For these measurements, the surroundings should have visual contrast. For example, an all white room with no windows would be bad, but a room with daylight coming through a single window or one with a lamp on would be ideal. Also, while running this portion of the program, there should be no movement in the room. The photoresistor should see exactly the same scene for every revolution of the Boe-Bot. So be sure to sit stock still or else dive out of sight before this segment starts.

When this section of the program is running, the Boe-Bot will rotate one ep at a time, alternating movements with its left and right wheels. After each increment, it will read the photoresistor and save the word value thus obtained to EEPROM. It will do this 256 times, completing about five full revolutions.

The Stamp program that does all this, Calibrate_All.bs2, is sensitive to the number of times the reset button is pressed in rapid succession before it begins. Different numbers of presses result in different sections of code being run. These are summarized below:

1 press: Dump all the data obtained.

2 presses: Send the servos 1.5ms pulses for nulling.

3 presses: Calibrate the servo velocities.

4 presses: Obtain photoresistor data.

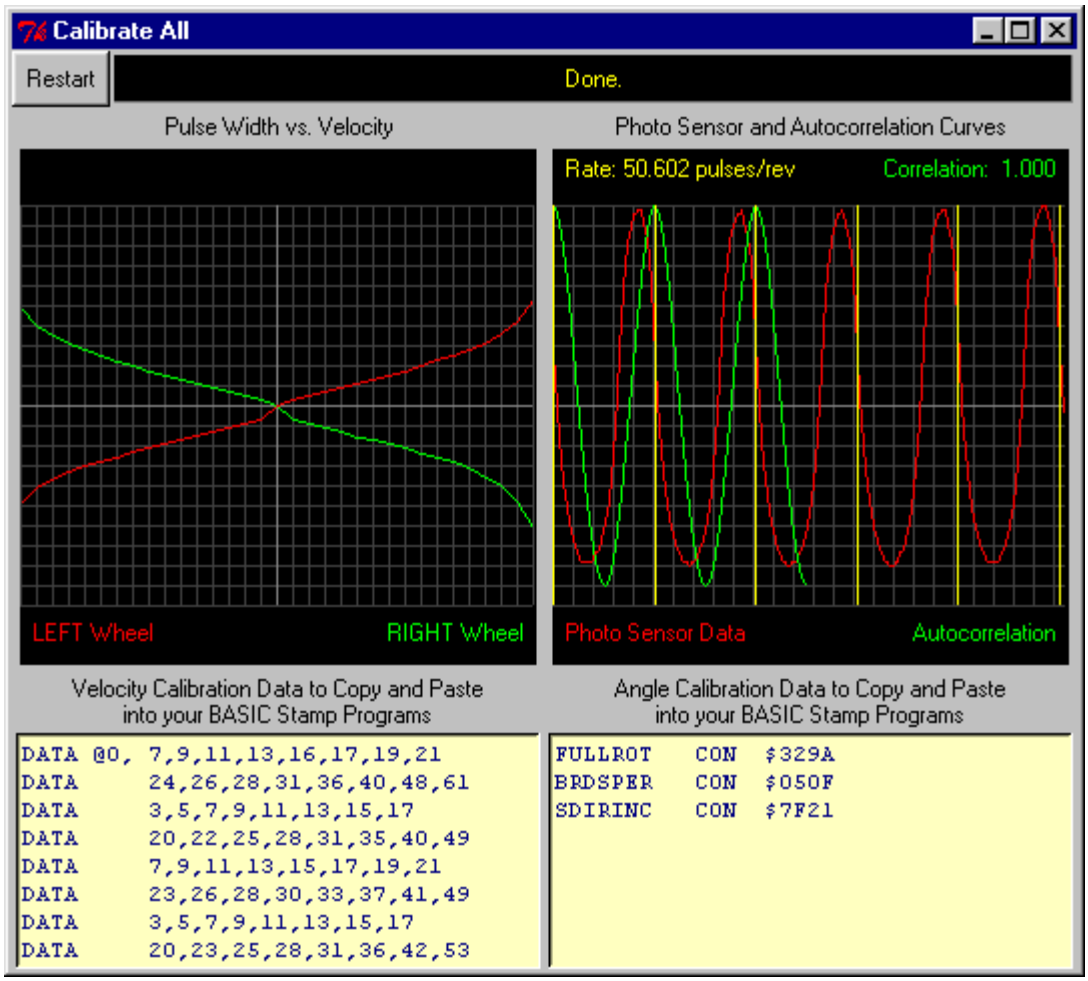
The format of the data dumped by this program is shown below:

```
Copy and paste these DATA statements into your BASIC Stamp programs:

DATA @0, 7,9,11,13,16,17,19,21
DATA     24,26,28,31,36,40,48,61
DATA     3,5,7,9,11,13,15,17
DATA     20,22,25,28,31,35,40,49
DATA     7,9,11,13,15,17,19,21
DATA     23,26,28,30,33,37,41,49
DATA     3,5,7,9,11,13,15,17
DATA     20,23,25,28,31,36,42,53

Autocorrelation data for Calibrate_All.exe:

58,51,45,38,33,28,25,22
19,17,15,14,12,11,10,10
10,10,10,10,11,12,12,13
```

```
15,17,19,21,24,29,34,40
48,57,65,71,77,81,84,85
88,88,89,89,89,86,85,82
79,72,65,55,47,40,35,30
26,23,20,18,16,14,13,12
11,10,10,10,9,10,10,11
11,13,14,15,17,20,22,26
31,37,44,51,61,69,73,77
82,84,86,87,88,88,89,87
85,83,81,77,70,59,53,45
38,32,29,25,21,19,17,15
13,12,11,10,10,9,9,10
10,10,11,12,13,14,16,18
21,24,28,32,39,46,54,62
70,75,79,82,85,86,88,88
89,88,86,84,82,79,74,66
57,47,42,36,31,27,24,21
18,16,15,13,12,11,10,10
9,9,9,10,10,12,12,13
15,17,19,22,26,31,36,42
51,59,67,72,77,82,85,86
88,88,89,89,89,86,84,82
79,72,64,55,46,39,35,30
26,22,20,18,16,14,13,12
11,10,10,10,10,10,10,11
12,13,14,16,18,21,23,27
33,39,45,53,63,71,76,80
85,86,88,89,90,90,90,88
86,83,81,76,68,58,51,43

END
```

The first 64 bytes of data saved in EEPROM are required by some of the other programs listed here. You can either leave the data in EEPROM or copy and paste the DATA statements output by the DEBUG port (your own, *not* the ones listed here) into your program directly. That way if the values in EEPROM ever get overwritten, they will be restored when you reload your program.

The remaining data are from the photoresistor. They may be analyzed by the program **Calibrate_All.exe** available for download from **www.parallax.com**. To use this program, just make sure you've run all sections of **Calibrate_All.bs2**. While it's still loaded in the Boe-Bot, connect the Boe-Bot to your PC, and run **Calibrate_All.exe**. It will download all the data from the Boe-Bot and perform the necessary calibration computations, including the autocorrelation optimization. A typical output screen looks like the one on the following page. On the left-hand side, the velocity data are plotted and redisplayed for copying and pasting. (Select all the text, and use ctrl-C to copy.) On the right-hand side, the photoresistor data are shown in red with the autocorrelation data in green. Vertical yellow bars denote the optimum detected period. The top bar presents this value numerically along with the computed correlation coefficient. This value has to be at least 0.975 to be acceptable, but the value of 1.000 shown here is highly unusual. If the correlation value is below 0.975 or if there's not enough contrast in the photoresistor data, an error message will appear, and you will have to rerun **Calibrate_All.bs2** under better lighting conditions. Assuming good data, calibration constants will appear in the text box below the graphs. These include an optimized value for **SDIRINC**, the **sin δ** factor used by the odometry routines to track orientation. The value is chosen not only to follow the pulses-per-revolution value correctly, but also to minimize divergence of the sine and cosine components from a true circle. You can copy and paste these **CON**stant statements into the BASIC Stamp programs that need them.




A listing for the calibration program follows. Some lines are too long to display in the available page width. Those are indicated by a continuation character ➤. This means that the next line of text should actually be keyed as a continuation of the current line.

## Demo Program (CALIBRATE_ALL.bs2)

```
'{$STAMP BS2}
'{$PBASIC 2.5}

'========[Calibrate_All.BS2]=========================================================

'This program performs several calibration functions, depending on how it's
'started:

'  1. A single (normal) reset causes calibration data to be dumped to the debug port.

'  2. Pressing reset twice in rapid succession causes the servos to be continuously
'     pulsed with 1.5ms pulses, so they can be nulled.

'  3. Pressing reset three times in rapid succession calibrates the wheel servos.
'     For each servo AND FOR each direction it calculates the correct pulse widths
'     for sixteen equally-spaced velocities. It stores the 64 byte values thus
'     obtained in the first 64 bytes of the Stamp's EEPROM. The Boe-Bot needs to be
'     sitting on a flat, smooth surface for this operation. (See documentation.)

'  4. Pressing reset four times in rapid succession obtains the data necessary for
'     calibrating the angular rate constants. It reads the output from a photoresistor
'     at each of 256 minimally-spaced angles and stores the word values obtained
'     in locations 64 - 575 of the Stamp's EEPROM. This data is later downloaded by the
'     PC program Calibrate_Angle.exe, which performs an autocorrelation to determine
'     how many pulses (to the nearest 1/256th pulse) in a complete revolution. The
'     Boe-Bot needs To be sitting on a flat, smooth surface for this operation.
'     (See documentation.)
```

```
'Written by Philip C. Pilgrim    9 April 2004

'---------[Time constants for various BASIC Stamps]---------------------------

#IF ($Stamp = BS2) OR ($Stamp = BS2E) #THEN

  NULL      CON 750       'Pulse width for 1.5ms pulse.
  SCALE     CON $100      '256 * amount to scale pulses by.

#ELSE

  NULL      CON 1875      'Pulse width for 1.5ms pulse.
  SCALE     CON $280      '256 * amount to scale pulses by.

#ENDIF

'---------[Constants for all Stamps]-----------------------------------------

MAXPULS   CON 100       'Adder/subtractor (before scaling) to NULL for max speed.

RIGHT     CON 0         'Constants used as subscripts into bit arrays.
LEFT      CON 1
CCW       CON 0
CW        CON 1
FWD       CON 0
BAK       CON 1

Photo     PIN 6         'Photo resistor input.

SenseR    PIN 10        'Lefthand encoder input.
SenseL    PIN 11        'Righthand encoder input. (MUST be SenseL + 1.)

MotorR    PIN 12        'Lefthand motor output.
MotorL    PIN 13        'Righthand motor output. (MUST be MotorL + 1.)

Sense     CON SenseR    'Base address for encoders.
Motor     CON MotorR    'Base address for motors.

Pulse     VAR Byte      'Current unscaled or nulled pulse value.
pPulse    VAR Byte      'Previous pulse value (used in interpolation).

i         VAR Word      'General FOR/NEXT index.
Value     VAR Word
p         VAR Byte      'Index into EEPROM table.
n         VAR Byte(2)   'Encoder counts for both sides.
nPrev     VAR Byte(2)   'Previous encoder counts for both sides.
v         VAR Byte(2)   'Next velocity index we're looking for (both sides).
nt        VAR Byte      'Next encoder count to find, corresponding to v.
pt        VAR Byte      'Interpolated pulse width to get the desired count.
nMax      VAR Byte      'Maximum sustainable velocity (encoder count).

Prev      VAR Bit(2)    'Previous readings from encoders.
New       VAR Bit(2)    'Current readings from encoders.
Side      VAR Bit       'Side index (RIGHT or LEFT).
Dir       VAR Bit       'Direction index (FWD or BAK).
Opp       VAR Bit

Veloc     VAR Nib

Dist      VAR Byte(2)   'Distance for each wheel to travel.
Counts    VAR Byte(2)   'Encoder pulse countdown for each wheel.

'---------[EEPROM table of pulse sample points]-------------------------------

STEPS     CON 13        'Pulse points at which to sample servo speeds.
          DATA @576, 0, 1, 2, 4, 6, 8, 10, 15, 20, 30, 40, 50, 60, 100

'=========[Program starts here]=================================================

READ 576, i
WRITE 576, i + 1
PAUSE 1000
WRITE 576, 0
SELECT i

  CASE 0

    PAUSE 1000
    DEBUG "Copy and paste these DATA statements into your BASIC Stamp programs:", CR, CR
    FOR i = 0 TO $3f
      IF (i = 0) THEN
        DEBUG "DATA @0, "
      ELSEIF (i & 7 = 0) THEN
        DEBUG "DATA     "
      ENDIF
      READ i, p
      DEBUG DEC p
      IF (i & 7 = 7) THEN DEBUG CR ELSE DEBUG ","
    NEXT
    DEBUG CR, "Autocorrelation data for Calibrate_All.exe:", CR, CR
    FOR i = $40 TO $23f STEP 2
      READ i, n(0)
```

```
      READ i+1, n(1)
      DEBUG DEC n(0) << 8 + n(1)
      IF (i & 15 = 14) THEN DEBUG CR ELSE DEBUG ","
    NEXT
    DEBUG CR, "END", CR
    END

  CASE 1

    DO
      PULSOUT MotorR, 750
      PULSOUT MotorL, 750
      PAUSE 50
    LOOP

  CASE 2

    GOTO Calibrate_Veloc

  CASE 3

    PAUSE 5000
    FOR i = 0 TO 512 STEP 2
      HIGH Photo
      PAUSE 6
      RCTIME Photo, 1, Value
      WRITE i + 64, Value.HIGHBYTE
      WRITE i + 65, Value.LOWBYTE
      Dist(LEFT) = i.BIT1
      Dist(RIGHT) = 1 - Dist(RIGHT)
      Dir(LEFT) = FWD
      Dir(RIGHT) = BAK
      Veloc = 1
      GOSUB DoMove
    NEXT
    END

  ENDSELECT
  END


Calibrate_Veloc:

'---------[Make sure servos are nulled]-----------------------------------------

Pulse = 0                                   'Pulse width is null value.
GOSUB Counter                               'Count encoders for 256 servo pulses.
IF (n(RIGHT) > 1 OR n(LEFT)> 1) THEN Error  'More than one pulse is error.

'---------[Find maximum sustainable rotation rate]-----------------------------

Pulse = MAXPULS */ SCALE                    'Pulse width for fastest speed.
Dir = CCW                                   'Direction is counter-clockwise.
GOSUB Counter                               'Count encoders for 256 servo pulses.
nMax = n(RIGHT) MAX n(LEFT)                 'Pick the smallest encoder count.
Dir = CW                                    'Now go the other way.
GOSUB Counter                               'Count encoders for 256 servo pulses.
nMax = nMax MAX n(RIGHT) MAX n(LEFT) */ $F8 'Pick the smallest encoder count * 31/32.
IF (nMax <= 1) THEN Error                   'If an encoder didn't respond, then error.

'---------[Get interpolated pulse widths in each direction]--------------------

FOR Dir = CCW TO CW                         'Once for one direction; once for the other.
  pPulse = 0                                'Previous pulse width deemed zero.
  FOR Side = RIGHT TO LEFT                  'Clear the count and velocity index arrays.
    nPrev(Side) = 0
    v(Side) = 0
  NEXT
  FOR p = 0 TO STEPS - 1                    'Index over the sample points.
    READ p + 577, Pulse                     'Get the next sample point.
    Pulse = Pulse */ Scale                  'Scale it for this Stamp.
    GOSUB Counter                           'Count encoders for 256 servo pulses.
    DEBUG DEC Pulse, ": ", DEC n(RIGHT), " ", DEC n(LEFT), CR
    FOR Side = RIGHT TO LEFT                'For both sides...
      DO                                    'Do until all interpolations in this section are done.
        nt = nMax * (v(Side) + 1) >> 4      'Scale nt to nMax.
        DEBUG "       (", DEC nt, ")", CR
        IF (v(Side) <= 15) THEN             'If v were 15, that side would be finished.
          IF (nt >= nPrev(Side) AND n(Side) >= nt) THEN  'If nt is between observed values...
                                            'Use linear interpolation to get pt.
            pt = (Pulse * (nt - nPrev(Side)) + (pPulse * (n(Side) - nt))) / (n(Side) - nPrev(Side)) + 1
            WRITE Side << 1 + Dir << 4 + v(Side), pt  'Save pt in EEPROM.
            DEBUG REP " "\(Side * 10 + 12), DEC Side, " ", DEC v(Side), ": ", DEC pt, " "
            v(Side) = v(Side) + 1           'Increment to next desired velocity.
          ELSE
            EXIT                            'Done in this section.
          ENDIF
        ELSE
          EXIT                              'Done with this side altogether.
        ENDIF
      LOOP
    NEXT
```

```
    pPulse = Pulse                          'Previous pulse width is current pulse width.
    FOR Side = RIGHT TO LEFT                'Previous counts are current counts.
      nPrev(Side) = n(Side)
    NEXT
  NEXT
NEXT

END

'---------[Error routine]-------------------------------------------------------

Error:                                      'Error condition detected. Waggle from side to side.
  FOR n = 1 TO 3
    FOR Dir = CCW TO CW
      FOR i = 1 TO 20
        FOR Side = RIGHT TO LEFT
          PULSOUT Motor + Side, NULL + ((Dir << 1 - 1) * (MAXPULS */ SCALE))
        NEXT
        PAUSE 20
      NEXT
    NEXT
  NEXT
  END

'---------[Encoder pulse counter]-----------------------------------------------

Counter:
  FOR Side = RIGHT TO LEFT                  'Get initial encoder states & clear counts.
    Prev(Side) = INS.LOWBIT(Sense + Side)
    n(Side) = 0
  NEXT
  FOR i = 0 TO 255                          '256 servo pulses.
    FOR Side = RIGHT TO LEFT
      New(Side) = INS.LOWBIT(Sense + Side)  'Get new servo states.
      IF (New(Side) <> Prev(Side)) THEN     'Different from previous state?
        Prev(Side) = New(Side)              '  Yes: Save new state.
        n(Side) = n(Side) + 1               '       Increment edge count.
      ENDIF
    NEXT
    FOR Side = RIGHT TO LEFT                'Pulse both motors.
      PULSOUT Motor + Side, NULL + ((Dir << 1 - 1) * Pulse)
    NEXT
    PAUSE 20                                'Delay between servo pulses.
  NEXT
  RETURN

'--------[DoMove]---------------------------------------------------------------

'Move RIGHT wheel by Dist(RIGHT) in direction Dir(RIGHT) and
'LEFT wheel by Dist(LEFT) in direction Dir(LEFT) at peak velocity Veloc,
'using ramping and RIGHT/LEFT coordination.

DoMove:

  'Initialize Counts TO Dist.
  'Save current encoder status.

  FOR Side = RIGHT TO LEFT
    Counts(Side) = Dist(Side)
    Prev(Side) = INS.LOWBIT(Sense + Side)
  NEXT

  'Do for as long as there are encoder counts remaining...

  DO WHILE (Counts(RIGHT) OR Counts(LEFT))

    'Get new encoder state for each wheel.
    'If it's changed, decrement that wheel's Count.

    FOR Side = RIGHT TO LEFT
      New(Side) = INS.LOWBIT(Sense + Side)
      IF (New(Side) <> Prev(Side) AND Counts(Side)) THEN
        Prev(Side) = New(Side)
        Counts(Side) = Counts(Side) - 1
      ENDIF
    NEXT

    'For each wheel decide whether and how much to pulse its servo.

    FOR Side = RIGHT TO LEFT
      Opp = ~ Side

      IF (Counts(Side) AND Counts(Side) * Dist(Opp) + (Dist(Side)) >= Counts(Opp) * Dist(Side) + ▶
(Dist(Opp) >> 1)) THEN

        Pulse = (Veloc MIN 3) MAX ((Counts(Side) MIN Counts(Opp)) MAX ((Dist(Side) - Counts(Side)) ▶
MIN (Dist(Opp) - Counts(Opp))) << 1 MIN 3)

        READ Side << 1 + (Dir(Side) ^ Side) << 4 + (Pulse * Dist(Side) / (Dist(Side) MIN Dist(Opp)) + ▶
1 MAX 15), Pulse
        PULSOUT Motor + Side, NULL - ((Dir(Side) ^ Opp << 1 - 1) * Pulse)
      ENDIF
    NEXT
```

```
  'Pause between pulses.

  PAUSE 5
LOOP
RETURN
```

### *Coordinated Motion*

The BASIC Stamp program described here provides coordinated, encoder-guided motion that can best be described as *proactive*. This means that you can tell the Boe-Bot what turns to make and how far to travel, and it'll go there. This stands in contrast with a *reactive* driver program which will simply monitor the encoders to keep track of where the Boe-Bot is. Each has its place in the grand scheme of navigation. And this is not to say that a proactive program cannot be made somewhat reactive as well. Even while the motion subroutine is showing off its coordination skills, it can still monitor the external environment for unexpected situations like obstacles and alert the main program accordingly. The astute programmer should have no difficulty modifying these routines for his/her own use.

The program presented here builds upon all the theory we've covered up to this point, as well as upon the calibration parameters obtained by the previous program. There are two main user-callable subroutines, **Turn** and **Move**. They do what their names suggest, and each uses the same byte variable, **Arg**, and nibble argument, **Veloc**, as its parameters.

**Turn** causes the Boe-Bot to rotate in place by the amount of **Arg**, a signed value ranging from –128 to 128, given in whole brads. **Veloc** can range from 0 to 15 and determines the peak velocity at which the wheel turning the furthest can attain. The following sequence, for example, will cause the Boe-Bot to turn left by 90 degrees (64 brads) at half speed:

```
Arg = 64
Veloc = 8
GOSUB Turn
```

That's all there is to it? Well, yes and no. As we saw previously, it may not be possible for the Boe-Bot to turn precisely to the angle we request, due to the coarseness of the encoders. But **Turn** will get as close as it can, *plus* keep track of the difference between the angle requested and the angle obtained. This difference is constantly maintained by the signed word variable **DirErr,** the directional error. **DirErr.HIGHBYTE** is the integer part of the error, and **DirErr.LOWBYTE** is the fractional part. So, as a whole, it is accurate to $1/256^{th}$ of a brad – that is, assuming the Boe-Bot's angle calibration is accurate.

The calibration constants are two: **FULLROT** and **BRDSPER**. **FULLROT** expresses the number of encoder pulses in one 256-brad turn, again as a whole number in the high byte, and a fraction in the low byte. It will be close to 50 for the Boe-Bot, *i.e.* $3200 in hexadecimal. **BRDSPER** is the number of brads rotation per encoder pulse – again a whole number and a fraction. It can be calculated on a hand calculator as $100000 / **FULLROT**, or 1048676 / **FULLROT** in decimal. Because the Version 2.5 tokenizer doesn't yet support 32-bit arithmetic in its constant expressions, it's always necessary to carry out this computation externally and enter the result in the program by hand. **FULLROT** itself needs to be determined by experiment. More on this later.

When **Turn** is called, it first increments **DirErr** by the amount of the requested turn. It then calculates the amount that each wheel should rotate – in integer encoder counts, remember – to reduce this error as nearly as possible to zero. This way, if a previous turn or movement left the Boe-Bot a

little askew, a subsequent turn won't compound the error by adding a new one, but rather attempt to correct the first one.

In every case, **Turn** will come up with a whole number of encoder counts by which to turn. This count has to be split between the two wheels, one moving forward, the other backward. If the total count is an odd number, one wheel will have to turn one count farther than the other one. Once this determination is made, **Turn** jumps to **DoMove**, which not only completes the requested motion, but decrements **DirErr** by the actual amount of rotation, thus returning it to the *difference* between the requested and actual direction the Boe-Bot is pointing.

**Move** commandeers the Boe-Bot's forward and backward motion. It's the subroutine that gets the Boe-Bot where we want to go in the direction we *think* we're pointing. It implements the mid-course correction introduced in the theory section to land the Boe-Bot on the desired destination, even if it wasn't pointed at it precisely to begin with.

When **Move** is called, **Arg** should be set to a number between –128 and 127. Positive values will move the Boe-Bot forward; negative values, backward. So the total move possible in one call to **Move** is 127 or 128 eps (about 64 inches, or 162 cm). As with **Turn**, **Veloc** can be set to any value from 0 to 15. (These values are offset by one, by the way: 0 represents 1/16 of maximum speed; 15 stands for 16/16 of maximum speed, *i.e.* actual maximum speed.) But because the minimum ramp speed is fixed at three, ¼ of full speed is effectively the minimum speed at which the Boe-Bot can travel.

The first thing **Move** must do when called is determine whether to make a beeline for its destination or break the path into two tacks to attain the destination more accurately. This is easy to do:

1. Just assume it's a two-legged journey, and calculate the length of the first leg.
2. If that length is zero, just move the entire distance.
3. If it's non-zero, move that amount, and calculate the length of the second leg.
4. If that length is zero, we're done.
5. If it's non-zero, turn by one encoder count, then move by the amount of the second leg.

Both **Turn** and **Move** use **DoMove** to effect their desired movements. **DoMove** is the heart of the whole shebang. It keeps track of angular error, performs the velocity ramping, monitors the encoders, and makes sure the wheels stay coordinated during their movements. **DoMove** requires four arguments: **Dist(LEFT)** and **Dist(RIGHT)**, the two elements of an *unsigned* byte array, and **Dir(LEFT)** and **Dir(RIGHT)**, the two elements of a bit array. The **Dist** variables tell **DoMove** how far to move each wheel. The **Dir** variables say which direction. Because **Dist** is unsigned, the directions being indicated in separate bits, each wheel can move by an amount between -255 to 255 eps.

In operation, **DoMove** makes presumptive corrections to **DirErr**, then initializes an encoder countdown array, **Counts**, to equal **Dist**, both **LEFT** and **RIGHT**. It then peeks at the encoders and records their current states.

Next it begins a **DO** loop, which terminates only when both elements of the **Counts** array are zero. In this loop, the first order of business is to see if the encoders have changed state. For each one that has, its **Counts** element is decremented by one. Then an **IF** statement checks to see, for each wheel, if it's significantly ahead of the other one. If so, it's skipped entirely; if not, it can be pulsed. The width of

that pulse, **Pulse**, is determined by the maximum speed, given by **Veloc**, as well as the current ramp velocity – up or down. The result is then multiplied by the fraction **Dist(Side) / Dist(Other side)** if **Side** is the slow side. The result of this is used as an index into one 16-byte section of the 64-byte EEPROM calibration table to arrive at a pulse width. That pulse width is then applied to the motor. When both **Counts** are zero, we're done!

For those hackers who want to tinker with **DoMove** to make it more reactive to external stimulae – possibly even aborting it early should some obstacle appear – there are some things to consider:

1. **DirErr** should either be updated at each step, or back-corrected by the amounts remaining in the **Counts** array.
2. If a premature stop is necessary, try to ramp the motion down if possible. If this isn't practical, don't exit the subroutine immediately, but wait until you're sure all motion has ceased, all the while keeping track of residual encoder counts.

Following is an annotated listing of the locomotion routines, with a main program designed to help calibrate angular motion. All it does is make the Boe-Bot traverse a square path, 50 eps on a side, in a counterclockwise fashion. A properly calibrated Boe-Bot running on a hard, flat surface will always return to its starting point when executing this program. This point can be marked with a piece of tape on the floor. If it overshoots, decrease the constant **FULLROT** a little, and recalculate **BRDSPER**. If it comes up short, adjust the other way.

**IMPORTANT:** A dead-reckoned journey is no better than its first step. To ensure the utmost success, remember the following:

1. Before placing the Boe-Bot on the floor, make sure its wheels are aligned such that both photosensors are centered in their respective wheel holes. This will help to guarantee that one encoder won't pulse prematurely and throw the initial direction off.
2. Orient the Boe-Bot carefully when placing it on the floor.
3. Program a long pause at the beginning of the main program to provide a moment's rest once your hand moves away from the power switch or reset button.

Here's the program. Some lines are too long to display in the available page width. Those are indicated by a continuation character ➤. This means that the next line of text should actually be keyed as a continuation of the current line.

## Demo Program (WHEEL_MOTION.bs2)

```
'{$STAMP BS2}
'{$PBASIC 2.5}

'This program consists of a set of subroutines for producing coordinated motion in the
'Boe-Bot wheels, through the use of encoder feedback.

'Addresses 0 - 63 of EEPROM are assumed to have calibration coefficients put
'there by the program Wheel_Calibrate.bs2.

'Written by Philip C. Pilgrim    30 March 2004

'Modified 6 April 2004:
'  Corrected bug in move routine resulting from switching LEFT/RIGHT port assignments.
'Modified 12 April 2004:
'  Added compile-time conditionals for NULL and SCALE
'  Changed calibration constants based on results from Calibrate All.exe.

'---------[Time constants for various BASIC Stamps]----------------------------
```

```
#IF ($Stamp = BS2) OR ($Stamp = BS2E) #THEN

  NULL      CON 750       'Pulse width for 1.5ms pulse.
  SCALE     CON $100      '256 * amount to scale pulses by.

#ELSE

  NULL      CON 1875      'Pulse width for 1.5ms pulse.
  SCALE     CON $280      '256 * amount to scale pulses by.

#ENDIF

'--------[Calibration Constants]-------------------------------------------------
'
' Adjust FULLROT for best precision on a closed course.
' Recompute BRDSPER as shown.
'
FULLROT     CON  $326D
BRDSPER     CON  $0514
SDIRINC     CON  $7F92

'---------[Other Global Constants]-----------------------------------------------

RIGHT     CON 0         'Constants used as subscripts into bit arrays.
LEFT      CON 1
FWD       CON 0
BAK       CON 1

SenseR    PIN 10        'Lefthand encoder input.
SenseL    PIN 11        'Righthand encoder input. (MUST be SenseL + 1.)

MotorR    PIN 12        'Lefthand motor output.
MotorL    PIN 13        'Righthand motor output. (MUST be MotorL + 1.)

Sense     CON SenseR    'Base address for encoders.
Motor     CON MotorR    'Base address for motors.

Arg       VAR Byte      'Requested travel or turn amount.
Veloc     VAR Nib       'Requested maximum velocity.

DirErr    VAR Word      'Current directional error in brads and 1/256 brads.

Prev      VAR Bit(2)    'Previous readings from encoders.
New       VAR Bit(2)    'Current readings from encoders.
Dir       VAR Bit(2)    'Wheel directions (FWD or BAK).
Side      VAR Bit       'Side index (RIGHT or LEFT).
Opp       VAR Bit       'Index to the other side (saves code).

Pulse     VAR Byte      'Current unscaled, unnulled servo pulse value.

Dist      VAR Byte(2)   'Distance for each wheel to travel.
Counts    VAR Byte(2)   'Encoder pulse countdown for each wheel.
i         VAR Byte      'General FOR loop index.

'=========[MAIN PROGRAM]=============================================================

PAUSE 2000
GOSUB Square
END

PAUSE 2000
GOSUB Octagon
PAUSE 2000
GOSUB Circle
PAUSE 2000
GOSUB Turn256
END

'=========[SUBROUTINES]==============================================================

'---------[Octagon]-------------------------------------------------------------

' Move Boe-bot along a counterclockwise, octagonal path, each leg of length 25.

Octagon:
  Veloc = 15                'Set velocity to maximum.
  FOR i = 1 TO 8            'Do for eight sides.
    Arg = 25: GOSUB Move    'Go straight for 25 eps.
    PAUSE 500
    Arg = 32: GOSUB Turn    'Turn left by 32 brads.
    PAUSE 500
  NEXT
  RETURN

'---------[Square]--------------------------------------------------------------

' Move Bot-Bot counterclockwise along a closed, square path, each leg of length 50.

Square:
  Veloc = 15                'Set velocity to maximum.
  FOR i = 1 TO 4            'Do for four sides.
```

```
    Arg = 50: GOSUB Move     'Move ahead by 50 eps.
    PAUSE 500
    Arg = 64: GOSUB Turn     'Turn left by 64 brads.
    PAUSE 500
  NEXT
  RETURN

'---------[Circle]------------------------------------------------------------

' Move Boe-Bot counterclockwise in a complete circle about its left wheel.

Circle:
  Veloc = 15                 'Set velocity to maximum.
  Dist(LEFT) = 0             'Left wheel stays still.
                             'Right wheel goes forward by number of encoder counts
                             '(rounded) in one full, 256 brad turn.
  Dist(RIGHT) = Dist(LEFT) + (FULLROT + $80 >> 8)
  Dir(RIGHT) = FWD            'Both directions are forward.
  Dir(LEFT) = FWD
  GOSUB DoMove               'Execute the move.
  RETURN

'---------[Turn256]-----------------------------------------------------------

'Spin the Boe-Bot on its axis one complete revolution.

Turn256:
  Veloc = 15                 'Set velocity to maximum.
  FOR i = 1 TO 2             'Do in two, 128-brad segments.
    Arg = -128               'Turn 128 brads to the LEFT.
    GOSUB Turn               'Execute turn.
  NEXT
  RETURN

'---------[Move]--------------------------------------------------------------

'Move the Boe-Bot forward/backward by signed byte Arg at unsigned speed Veloc.

Move:
  Dir(LEFT) = Arg.BIT7      'Direction of motion given by sign bit of Arg.
  Dir(RIGHT) = Dir(LEFT)     'Both directions are the same.
  Arg = ABS(Arg << 8) >> 8   'Take the absolute value of Arg to get distance.
                             'Compute length of first leg of tack.
  Dist(LEFT) = Arg * (16 - (ABS(DirErr) / (BRDSPER >> 4))) / 16
  IF (Dist(LEFT)) THEN       'Is it greater than zero?
    Dist(RIGHT) = Dist(LEFT) '  Yes: Right wheel goes the same distance.
    GOSUB DoMove             '       Execute the move
    Arg = Arg - Dist(LEFT)   '       Subtract the distance moved from Arg.
    IF (Arg) THEN            '       Anything left?
      Dist(LEFT) = DirErr.BIT15     '  Yes: Correct direction by the sign of DirErr.
      Dist(RIGHT) = Dist(LEFT) ^ 1  '       Only one wheel moves forward.
      GOSUB DoMove                  '       Execute the move.
    ENDIF
  ENDIF
  IF (Arg) THEN              'Now back to Arg. Is it greater than zero?
    Dist(RIGHT) = Arg         '  Yes: Set distances from it.
    Dist(LEFT) = Arg
    GOSUB DoMove             '       Execute this leg of the move.
  ENDIF
  RETURN

'---------[Turn]--------------------------------------------------------------

'Turn the Boe-Bot on its axis by the amount in signed byte Arg at speed Veloc.
'Arg > 0 turns left; Arg < 0 turns right.

Turn:
  DirErr = DirErr + (Arg << 8)          'Adjust direction error by amount of turn.
  Arg = ABS(DirErr) ** FULLROT >> 7 + 1 >> 1 'Compute new Arg from new DirErr.
  Dir(RIGHT) = DirErr.BIT15             'Direction of turn is sign bit of DirrErr.
  Arg = ABS(Arg << 8) >> 8              'Amount of turn is absolute value of Arg.
  Dist(LEFT) = Arg >> 1                 'Split turn amount evenly between both wheels.
  Dist(RIGHT) = Arg >> 1 + (Arg.BIT0)   'Round right wheel up if Arg is odd.
  Dir(LEFT) = ~ Dir(RIGHT)              'Left wheel goes opposite direction.
  GOTO DoMove                           'Execute the motion.

'--------[DoMove]-------------------------------------------------------------

'Move right wheel by Dist(RIGHT) in direction Dir(RIGHT) and
'left wheel by Dist(LEFT) in direction Dir(LEFT) at peak velocity Veloc,
'using ramping and RIGHT/LEFT coordination.

DoMove:

  'Correct DirErr by effects of presumptive motion.
  'Initialize Counts TO Dist.
  'Save current encoder status.

  FOR Side = RIGHT TO LEFT
    DirErr = DirErr + ((Side ^ Dir(Side) << 1 - 1) * Dist(Side) * BRDSPER)
    Counts(Side) = Dist(Side)
    Prev(Side) = INS.LOWBIT(Sense + Side)
```

```
  NEXT

  'Do for as long as there are encoder counts remaining...

  DO WHILE (Counts(RIGHT) OR Counts(LEFT))

    'Get new encoder state for each wheel.
    'If it's changed, decrement that wheel's Count.

    FOR Side = RIGHT TO LEFT
      New(Side) = INS.LOWBIT(Sense + Side)
      IF (New(Side) <> Prev(Side) AND Counts(Side)) THEN
        Prev(Side) = New(Side)
        Counts(Side) = Counts(Side) - 1
      ENDIF
    NEXT

    'For each wheel decide whether and how much to pulse its servo.

    FOR Side = RIGHT TO LEFT
      Opp = ~ Side

      IF (Counts(Side) AND Counts(Side) * Dist(Opp) + (Dist(Side)) >= Counts(Opp) * Dist(Side) +
(Dist(Opp) >> 1)) THEN

        Pulse = (Veloc MIN 3) MAX ((Counts(Side) MIN Counts(Opp)) MAX ((Dist(Side) - Counts(Side)) MIN
(Dist(Opp) - Counts(Opp))) << 1 MIN 3)

        READ Side << 1 + (Dir(Side) ^ Side) << 4 + (Pulse * Dist(Side) / (Dist(Side) MIN Dist(Opp)) +
1 MAX 15), Pulse
        PULSOUT Motor + Side, NULL - ((Dir(Side) ^ Opp << 1 - 1) * Pulse)
      ENDIF
    NEXT

    'Pause between pulses.

    PAUSE 5
  LOOP
  RETURN
```

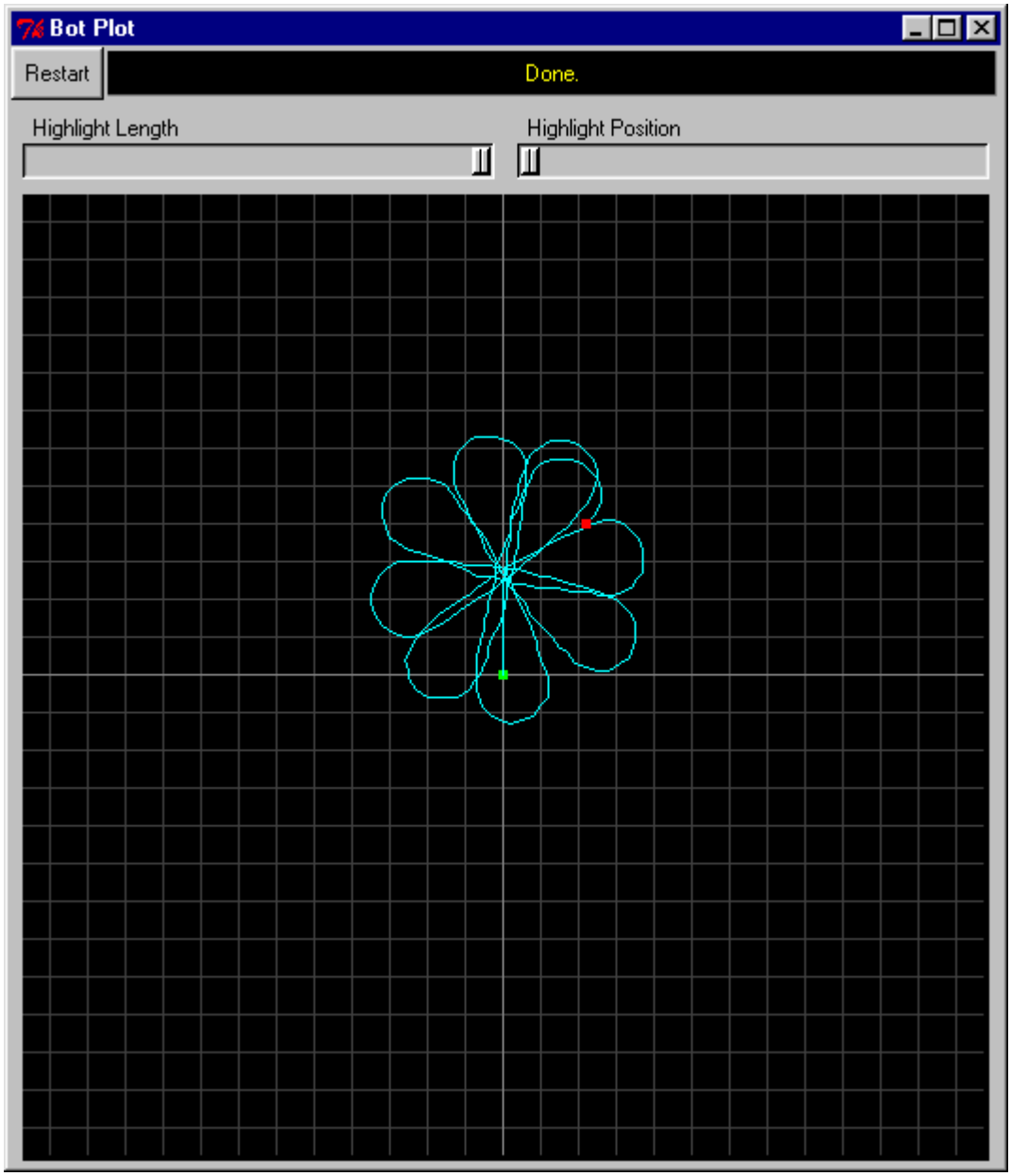
### *Wheel Odometry*

The BASIC Stamp program presented here uses the encoders to keep track of the Boe-Bot's position and direction. It also records the high bytes of the **x** and **y** position coordinates in the Stamp's EEPROM for later retrieval.

What the program does when it starts depends on how many times in rapid succession the reset button was pressed. If just once, it dumps the positions recorded in EEPROM to the DEBUG port. If twice, it begins a sequence of moves roughly describing a clover- or petal-shaped trajectory. You can view the recorded trajectory graphically using the PC host program **botplot.exe** downloadable from **www.parallax.com**. A typical display from **botplot** will look something like the one on the following page. The center position is (128, 128), and the graph intervals are every ten units. The start is marked with a green box; the end, with a red one. The sliders at the top control how much of the total path is highlighted and where the highlighting begins. This is useful for unraveling complicated paths with lots of overlaps.

The constants **SDIRINC** and **SPOSINC** are calibration constants. **SDIRINC** can be obtained from **Calibrate_All.exe** and tells the program how much the Boe-Bot turns after one encoder pulse from one wheel. It is the value **sin(δ)** discussed in the theory section, scaled up by $2^{18}$ (262144). **SPOSINC** tells the program how much the center of the Boe-Bot moves forward after one encoder pulse from each wheel. This can be in inches, centimeters, or some other unit with a similar order of magnitude. A value of 256 corresponds to ½ inch of travel. 256 * 2.54 cm/inch = 650 corresponds to the same distance in centimeters.

**XINIT** and **YINIT** are the starting **x** and **y** coordinates. The starting direction is always assumed to be north (towards +**y**).




**RECINT** is the recording interval. It determines whether and how often the **x** and **y** coordinates are written to the Stamp's EEPROM. Only the HIGHBYTE is written, which usually corresponds to whole units with the fractional part lopped off (depending on the value of **SPOSINC**). **RECINT** can range from 0 (don't record) to 15 (record on every 15$^{th}$ encoder pulse).

This program, like the other examples, assumes that the right-hand motor and encoder are on the lower-numbered port of each port pair. The constants **SenseR**, **SenseL**, **MotorR**, and **MotorL** determine which ports are used.

All of the Boe-Bot's nav state info is contained in nine bytes of variable space. Four **Word** variables, **Xpos**, **Ypos**, **Xdir**, and **Ydir** track the bot's position and direction. Eight **Bit** variables keep track of previous encoder readings (so changes can be detected), the direction each wheel is turning (set prior to each movement), bookkeeping bits for grouping movements into pairs (the anti-crabbing stuff), a bit for telling the program whether the bot moved in the last interval, and an index bit for selecting which wheel to process.

A call to the subroutine **Initialize** at the program's beginning gets everything set up. The meat of the odometry is the subroutine **ChkEnc**. It requires that **Dir(LEFT)** and **Dir(RIGHT)** each be set to **FWD** or **BAK** so it knows which direction each wheel is turning. **ChkEnc** just needs to be called often enough while driving the motors to ensure that all encoder edges are caught. Calling it between servo pulses will more than suffice for this purpose. It checks the encoders, updates the nav state info for each detected pulse, and records the bot's position in EEPROM at appropriate intervals.

**WaitEnc** should always be called at the end of a motion sequence to account for any stray encoder pulses that occur once the motors cease being driven. It will return when all motion has stopped.

## Demo Program (WHEEL_ODOMETRY.bs2)

```
'{$STAMP BS2}
'{$PBASIC 2.5}

'-----------------------------------------------------------------------------
'  Sample program to keep track of position and direction using wheel encoders.
'-----------------------------------------------------------------------------

'Written by Philip C. Pilgrim      30 March 2004

'Modified 12 April 2004:
'  Changed calibration constants based on results from Calibrate_All.exe.
'Modified 13 April 2004:
'  Changed from -128 to 127 coordinate system centered at (0,0) to a 0 to 255
'  system centered at (128, 128).
'  Changed startup to depend on number of reset button presses instead of a
'  debug port input.

'---------[Calibration Constants]---------------------------------------------

SDIRINC   CON $7F92     'Sin(delta) factor by which to modify the angle vectors for each
                        'encoder pulse. Increase this number if Boe-Bot turns
                        'more than odometer says it does. Decrease, if less.
                        'Sin(delta) = SDIRINC / 262144.)

SPOSINC   CON 650       'Sin(delta) factor to compute distance for each encoder pulse.
                        '256 corresponds to 0.5" travel per two-wheel pulse.
                        'For centimeters, start with 650 to get 1.27cm of travel per
                        'two-wheel pulse. (SPOSINC = W * SDIRINC / 512, where W
                        'is the effective distance between wheels in the desired units.)

'---------[Initialization Constants]------------------------------------------

XINIT     CON 128       'Initial X position (byte) in whole units.
YINIT     CON 128       'Initial Y position (byte) in whole units.
                        'Initial direction is always assumed to be towards +Y (north).

RECINT    CON 4         'Position record interval:
                        '  0 = don't record.
                        '  1 - 15 = record every 1 - 15 encoder pulses.

'---------[Other Constants]---------------------------------------------------

MAXADDR   CON $3FF      'Maximum writable EEPROM address.

RIGHT     CON 0         'Constants used as subscripts into bit arrays.
LEFT      CON 1
FWD       CON 0
BAK       CON 1

SenseR    PIN 10        'Righthand encoder input.
SenseL    PIN 11        'Lefthand encoder input. (MUST be SenseR + 1.)

MotorR    PIN 12        'Righthand motor output.
MotorL    PIN 13        'Lefthand motor output. (MUST be MotorR + 1.)

Sense     CON SenseR    'Base address for encoders.
Motor     CON MotorR    'Base address for motors.
```

```
'---------[Variables]-----------------------------------------------------------

Prev      VAR Bit(2)    'Previous readings from encoders.
Dir       VAR Bit(2)    'Wheel directions (FWD or BAK).
LastSide  VAR Bit       'Last side to get a pulse (LEFT or RIGHT).
NewSeq    VAR Bit       'Set if last movement started a new LR or RL sequence.
Side      VAR Bit       'Side index (LEFT or RIGHT).
Moved     VAR Bit       'Indicates motion since last checking.

Xpos      VAR Word      'X location of the Boe-Bot's center.
X         VAR Xpos.HIGHBYTE 'Integer portion of Xpos. Low byte is fraction.
Ypos      VAR Word      'Y location of the Boe-Bot's center.
Y         VAR Ypos.HIGHBYTE 'Integer portion of Ypos. Low byte is fraction.
Xdir      VAR Word      'X component of the Boe-Bot's direction.
Ydir      VAR Word      'Y component of the Boe-Bot's direction.
MemPtr    VAR Word      'Pointer into EEPROM for position recording.
MemCtr    VAR MemPtr.HIGHNIB 'Countdown for record interval.

i         VAR Word      'Scratch variables...
n         VAR Byte
pulse     VAR Word

'---------[Program begins here.]-------------------------------------------------

DATA @64, 0                 'Initialize reset button count on upload.

PAUSE 10                    'Debounce reset button.
READ 64, i                  'Read reset button count - 1.
WRITE 64, i + 1             'Increment count and save back to EEPROM.
PAUSE 1000                  'Wait one second.
WRITE 64, 0                 'If not reset again during wait, reinitialize count.
SELECT i                    'Act based on reset button count - 1.

'---------[Start here on one reset button press.]------------------------------

  CASE 0:

    DumpData:               'Dump stored data to DEBUG port.

      PAUSE 1000            'Wait one second for host program to get ready.
      DEBUG "NEW", CR       'Send start message.
      MemPtr = 65           'Start reading from address 64.
      DO                    'Main reading loop...
        READ MemPtr, X      'Read 2's complement X and Y byte values.
        READ MemPtr + 1, Y
        DEBUG "X:", DEC X, ",", "Y:", DEC Y, CR 'Output one line of data.
        MemPtr = MemPtr + 2 'Increment memory pointer.
        Xpos.LOWBYTE = X
        READ MemPtr, X      'Peek at next X value.
      LOOP UNTIL (X ^ Xpos.LOWBYTE = $80) 'When next X differs from previous by
                                          '$80, we're done.
      DEBUG "END", CR       'Send end message.
      END                   'That's all.

'---------[Start here on two reset button presses.]----------------------------

  CASE 1:

    MainProg:                         'Begin sample motion profile.

      GOSUB Initialize                'Do encoder initialization.

      FOR n = 1 TO 9                   'This will be a nine-leaf clover.
        FOR pulse = 700 TO 735 STEP 35 '700 for straighter areas, 730 for turns.
          FOR i = 1 TO 100             'Proceed with 100 servo pulses.
            Dir(LEFT) = FWD            'Must set direction values so encoder
            Dir(RIGHT) = FWD           'routine will know.
            PULSOUT MotorL, 850        'Pulse the servos.
            PULSOUT MotorR, pulse
            GOSUB ChkEnc               'Update the encoders.
            PAUSE 20                   'Wait 20 ms.
          NEXT                         'Next servo pulse.
        NEXT                           'Next pulse value.
      NEXT                             'Next leaf in clover.

      GOSUB WaitEnc                    'Wait for all motion to cease.
      END                              'We're done.

ENDSELECT
end

'---------[Subroutines]----------------------------------------------------------

Initialize:                       'Do encoder initialization.

  Xdir = 0                        'Set direction to +Y.
  Ydir = $4000
  Xpos = XINIT << 8               'Set positions from constants.
  Ypos = YINIT << 8
  IF (RECINT) THEN                'Are we going to record positions?
    WRITE 65, X                   '  Yes: Record initial positions now.
    WRITE 66, Y
```

```
    MemPtr = RECINT << 12 + 67     '        Initialize MemPtr and MemCtr.
  ENDIF
  FOR Side = RIGHT TO LEFT         'Read initial encoder values.
    Prev(Side) = INS.LOWBIT(Sense + Side)
  NEXT
  RETURN                           'Done.

'-------------------------------------------------------------------------------

WaitEnc:                           'Track encoders until motion ceases.

  FOR i = 1 TO 30                  'Need 30 motionless intervals to be sure.
    GOSUB ChkEnc                   'Check the encoder.
    IF (Moved) THEN i = 0          'If the bot moved, reset counter and start over.
  NEXT
  RETURN                           'No motion for 30 steps. It's stopped.

'-------------------------------------------------------------------------------

ChkEnc:                            'Update and record position from encoders.
                                   'Just call it often enough to catch all the
                                   'changes.
  Moved = 0                        'Initialize to no detected movement.
  FOR Side = RIGHT TO LEFT                 'For both encoders...
    IF (INS.LOWBIT(Sense + Side) ^ Prev(Side)) THEN  'Encoder different from prior value?

        Prev(Side) = ~ Prev(Side)          '  Yes: Update with new value.
        Moved = 1                          '       Indicate that we've moved.
        NewSeq = ~ (Side ^ Dir(Side) ^ LastSide & NewSeq)   'Start a new LR or RL sequence unless
                                           '       different side moved last and that was
                                           '       the start of a new sequence.
        LastSide = Side ^ Dir(Side)                  '       Update last side to move.
        IF (NewSeq) THEN DoPos             '       New sequence starts with position.
                                           '       2nd half of same sequence undoes angle first.

      DoAng:                               '       Angle change.

        Ydir = Ydir - ((Dir(Side) ^ Side ^ Xdir.BIT15 << 1 - 1) * (ABS(Xdir) ** SDIRINC >> 3))
        Xdir = Xdir + ((Dir(Side) ^ Side ^ Ydir.BIT15 << 1 - 1) * (ABS(Ydir) ** SDIRINC >> 2))
        Ydir = Ydir - ((Dir(Side) ^ Side ^ Xdir.BIT15 << 1 - 1) * (ABS(Xdir) ** SDIRINC >> 3))
        IF (NewSeq) THEN DoNext

      DoPos:                               '       Position change.

        Xpos = Xpos - ((Dir(Side) ^ Xdir.BIT15 << 1 - 1) * (ABS(Xdir) ** SPOSINC))
        Ypos = Ypos - ((Dir(Side) ^ Ydir.BIT15 << 1 - 1) * (ABS(Ydir) ** SPOSINC))
        IF (NewSeq) THEN DoAng

      DoNext:

    ENDIF
  NEXT

  IF (Moved AND RECINT > 0) THEN           'If bot moved and we're recording it...
    MemCtr = MemCtr - 1                    'Decrement counter (i.e. MemPtr.HIGHBYTE).
    IF (MemPtr < MAXADDR - 2) THEN         'If counter is zero AND address is low enough...
      WRITE MemPtr, X                        'Write X and Y to EEPROM.
      WRITE MemPtr + 1, Y
      WRITE MemPtr + 2, X ^ $80              'Add an impossible next X for end-of-file.
      MemPtr = RECINT << 12 + MemPtr + 2     'Reset counter and update pointer.
    ENDIF
  ENDIF
  RETURN                                   'Over and out.
```