<BACK

*by Jack W. Crenshaw*

# Putting It Together

When we're writing software for real-time use or analysis, we often need to use numbers that represent real parameters such as position, velocity, angle, temperature, and so on. All of these parameters are inherently continuous values, which means they aren't integers. Computers, on the other hand, are inherently integer machines—they don't understand decimal points. We thus have a mismatch between the kinds of numbers we'd like to manipulate and the kinds the computer is comfortable with.

To handle this mismatch, most programming languages give support for floating-point arithmetic with built-in types defined as real, float, double, or equivalent type names. Most high-end processors include floating-point hardware, using either built-in commands or separate math coprocessors.

Unfortunately, many of the smaller CPUs that find their way into embedded systems don't include such hardware. Even if the CPU vendor offers a math coprocessor option, the coprocessor is a tempting target for product cost-cutting efforts and is often eliminated as a dubious economy measure. For this reason, we sometimes find ourselves trying to represent real numbers in a computer that doesn't support them. We can always use floating-point software—most modern compiler vendors include a floating point to accommodate CPUs that don't have the necessary software. But floating-point software is notoriously slow, often much too slow for real-time applications. What's more, in many cases (Microsoft Visual C++, for example), the floating-point software is not reentrant and cannot be used in real-time applications.

Real-time programmers have encountered this problem for years and found fixed-point arithmetic to be a workable if not exactly pleasant solution. Fixed point is sort of a do-it-yourself floating point where the programmer specifies the scaling, exponent, and so on, at design time rather than execution time. In essence, we're trading ease of programming for speed, and real-time programmers have long since accepted this trade as a way of life, at least until fast floating-point processors are universally available.

**We'll look at two ways to implement the sine function: one optimized for speed and one for size.**

Over the past several months, we've been dealing with the issues associated with fixed-point arithmetic and ways of implementing it on a pure-integer machine. This month, we're going to see the results of our labors in action in the form of a practical example.

As long as we're developing fixed-point software, we might as well make it something useful. The example I've chosen is about as practical as they come: It's a function to compute the sine of an angle. In fact, we're going to look at two ways to implement the sine function: one optimized for speed and one for size. I'll be showing you more than one approach, because each uses different aspects of fixed-point methods, so we get to exercise more of what we've learned. Furthermore, if we're going to do this at all, we might as well end up with functions that meet all your needs, that you can pass down to your grandchildren. One routine can't fill that bill, but perhaps with two, you may never need to revisit this subject again. Wouldn't that be nice?

## THE BASICS

We've already talked about how best to represent an angle in fixed point, and I've pointed out that expressing the angle in pirads (scale B0) is best, because the angle naturally wraps around from 359+° to zero, as the integer wraps from 0xffff to 0x0000. For this particular representation, and this one only, we can get away with representing a number that's seemingly larger than one in B0 format, because of the way the sign of angles behaves. In other words, +180° is the same thing as -180°, and 1.5 pirads is the same as -0.5. The absolute value of the angle never passes unity.

We also talked about the best scale for the sine of an angle and agreed that B0 would be nice, providing we agree never to let the value reach unity. This means we must fudge the values at angles such as zero and 90°, representing the actual value of 1.0 as something just less. This slight inaccuracy at four discrete angles is more than offset by the extra bit of accuracy we get at the other 65,532 possible angles. However, the decision means we must be very, very careful that the algorithm we use to compute the sine can never produce a value outside the acceptable range.

Based on these two choices of scalings, we can generate the table of sine function values shown in Table 1. Our next challenge is to invent the software that will generate this table, along with the values at all intermediate angles.

## THE LOOKUP METHOD

Methods to generate the sine function include two extremes, plus a whole continuum of methods in between. At one end of the spectrum, we can simply

**TABLE 1**
*Fixed-point sine function.*

| Angle (degrees) | Angle, B0 (pirads) | Sine, B0 |
|---|---|---|
| 0 | 0x0000 | 0x0000 |
| 22.5 | 0x1000 | 0x30fc |
| 45 | 0x2000 | 0x5a82 |
| 67.5 | 0x3000 | 0x7642 |
| 90 | 0x4000 | 0x7fff |
| 112.5 | 0x5000 | 0x7642 |
| 135 | 0x6000 | 0x5a82 |
| 157.5 | 0x7000 | 0x30fc |
| 180 | 0x8000 | 0x0000 |
| -157.5 | 0x9000 | 0xcf04 |
| -135 | 0xa000 | 0xa57e |
| -112.5 | 0xb000 | 0x89be |
| -90 | 0xc000 | 0x8001 |
| -67.5 | 0xd000 | 0x89be |
| -45 | 0xe000 | 0xa57e |
| -22.5 | 0xf000 | 0xcf04 |
| 0 | 0x0000 | 0x0000 |

store a table of numbers and optionally interpolate between them. A lookup without interpolation is, of course, the fastest possible way to compute the function, since it only involves accessing an array element.

Because the array access is clearly the easiest, we'll do it first. Assuming that we have an angle expressed as a 16-bit integer, the angle will have 64K possible values. This means that if we're to represent all the values exactly, we'll need to allocate 128K of RAM for the table. Such extravagance would have seemed out of the question a few years ago. Today, with desktop computer memories measured in megabytes, it's not so ridiculous. Still, we're far more likely to be willing to settle for an approximation that uses only, say, 1024 addresses or 2048 bytes.

First, we need to build the table. The code in Listing 1 defines the table and shows a function to fill the table with values. You wouldn't actually code this function into a real-time system, though, you'd use it offline to build the table and encode it as an array of constants in ROM. As you might guess, parameter `TABLE_SIZE` defines the size of the target array. Parameter `SHIFT`, which must always be changed with changes in `TABLE_SIZE`, tells the function how to compute the angle corresponding to a given index. Parameter `BITE` defines the number of counts between table entries. Stated another way, `BITE` is the angular step between entries, measured in pirads, scaled B0. Converting this number gives us:

**LISTING 1**
*Building a table.*

```
// number of entries in table
#define TABLE_SIZE 1024
// 16 - log2(TABLE_SIZE)
#define SHIFT 6
// step size
#define BITE (65536/TABLE_SIZE)

int sine_table[TABLE_SIZE];

void make_table(void){
  for(int i=0; i< TABLE_SIZE; i++){
    double ang = pi * (double)(i <<
      SHIFT)/32768.0;
    double sine = sin(ang);
    if(sine >= 1.0)
      sine = 0.99997;
    if(sine <= -1.0)
      sine = -0.99997;
    sine_table[i] = (int)(sine *
      32768.0);
  }
}
```

```
BITE    = 65536/1024
        = 64
        = 0x0040, scale B0
        = 0.001953125 pirads
        = 0.000621698 radians
        = 0.3515625°
```

Thus, we have about three steps per degree.

Several months ago, in my column entitled "Look It Up" (Jan. 95, pp. 13-24), we looked at ways of approximating a general function, using table lookups. I don't want to rehash that column here, but we must revisit the techniques learned because those techniques were developed for floating-point arithmetic. We must see how they should be modified to work with our fixed-point arithmetic.

We learned that a table lookup consists of two parts, which we might think of as the course lookup and the vernier correction. In the first part, we figured out which index in the table most closely corresponds to our input variable. In the second part, we interpolated between the two nearest tabular points. To get the index, we learned that we must divide the input value by the table size:

$$i = \frac{x}{\Delta x} \qquad (1)$$

where x is the input value, Δx is the step size in x between tabular points (the same as our `BITE`), and we are supposed to round i to the nearest integer. In C code, we used the line:

```
int i = max(0, min(MAX_INDEX, (int)(x/delta
   + 0.5)));
```

The purpose of the `min` and `max` functions was to limit the index to the legal range, even if the function were given a bogus value of x.

This process of finding i becomes much easier in our current case. In the first place, since trig functions repeat as the angle goes through a full circle, we don't need to bother limiting the index except, at most, to apply a modulo function. Secondly, I chose a power of two for the number of table entries. Since the full range of the angle is also a power of two, in our fixed-point representation, the division reduces to a simple right shift:

$i = x >> 6;$

If we want to round the value of x, we could add 32 before shifting. More generally:

$i = (x + \text{BITE}/2) >> \text{SHIFT};$

A simple table lookup for the sine function is shown in Listing 2. Small, isn't it? Fast, too. For the ultimate in speed, you can implement it in assembly language. Since the algorithm is basically an add, shift, and indexed

**LISTING 2**
*A simple sine function.*

```
int sine(unsigned int ang){
  ang += (unsigned int)(BITE/2);
  return sine_table[ang >> SHIFT];
}
```

load, it will be blazingly fast.

## INTERPOLATION

Because the function of Listing 2 doesn't use interpolation, we can expect an accuracy equivalent only to the maximum change in an increment of BITE/2. That difference is greatest at x=0 and is equal to 0.003. Even that error is not too bad; for some applications, such as computer graphics where things must be rounded to the nearest pixel anyway, it may be good enough. For most applications, though, it isn't, and we must use interpolation to trim up the result.

In my last column, we also learned that the interpolation formula is:

$$y(x) \approx y_0 + \left(\frac{y_1 - y_0}{x_1 - x_0}\right)(x - x_0) \qquad (2)$$

where y is the desired output value. The value $(x - x_0)$ is the fractional step from the beginning of the interval. In C, we had a simple formula for this as well:

```
double dx = mod(x, delta);
```

Again, in our present state, things are even simpler. Since `BITE` is a power of two, the modulo function becomes a simple masking process:

```
dx = x & MASK;
```

where, for our example:

```
MASK = 0x3f;
```

More generally:

```
MASK = BITE - 1
```

In Equation 1, the term:

$$M = \frac{y_1 - y_0}{x_1 - x_0} \qquad (3)$$

is the slope. We can compute this value from the known table entries. However, as I pointed out in "Look It Up," that's a waste of computing power. The value of M is never going to change, for a given table entry. Thus, we could build a second table of slopes, and use it rather than computing the slope at run time.

We could build one, but we won't, because there happens to be an even better approach. As we look at this approach, it's vitally important that you totally understand what we're doing and how all the shifting and scaling gets involved, so I'll explain it in considerable detail.

First, consider the numerator of Equation 3. This is a simple difference of two successive table entries. It can be scaled the same as the entries themselves, which in our example is B0. Because we're subtracting two nearly equal numbers, we can expect to get a lot of zeros in the upper bits of the words. We may be tempted to rescale the number in the hope of gaining more accuracy, but that would be a vain hope, because we can't get more accuracy than the subtraction gives us. Shifting the number left to fill up the word would only shift zeros into the lower order bits, and there's no advantage to that. We should leave the number in its original scaling.

Next, consider the denominator of M. This is merely the step in x between table entries. In other words, it's our old friend, `BITE`. Not only is this number a constant for all table entries, it's a power of two. So we can form the ratio by shifting again. But should we? We shouldn't, because shifting to the right can only cost us precision. There's no advantage to that, either.

A better way to look at the math is to rewrite Equation 1 like this:

$$y(x) \approx y_0 + (y_1 - y_0)\left(\frac{x - x_0}{x_1 - x_0}\right) \qquad (4)$$

Here, instead of computing the slope, I've lumped the step size into a ratio with x. This ratio can vary only between zero and one, since if x were larger than $x_1$, we'd move to a different index. Obviously, B0 is the right scaling for such a number.

If you remember how to multiply and divide fixed-point numbers, you'll recall that we should multiply them as integers into a long result, then shift the result right. For the product of two B0 numbers, we should shift right 15 bits. Similarly, to divide two numbers, we should shift left 15 bits before dividing. Finally, to divide by `BITE`, we must shift right six bits.

Now pay attention, because this is very important: If we perform the operations in Equation 4 in the right sequence, we can greatly simplify the algorithm. That sequence is neither our first thought, to compute M, nor our second, to compute the ratio of x steps, but a third sequence, shown like this:

$$y(x) \approx y_0 + \frac{(y_1 - y_0)(x - x_0)}{x_1 - x_0} \qquad (5)$$

In other words, we first multiply two numbers, then divide. In this situation, the shifts offset each other. The last thing we do after a multiply is to shift right; the first thing for a divide is to shift left. If the scales are the same, the shifts cancel, and we can omit them.

(As an aside, the language Forth supports this very operation, using the operator `*/`. To my knowledge, no other language does this. They all should, though I can't imagine what the syntax would look like in such a three-argument operation.)

Because all the numbers in our example are scaled B0, no shifting is necessary except for the division by `BITE`, which is itself a shift. The end result of all this discussion is that we need only one shift, a right shift after the multiplication. The use of a long integer for temporary storage is not needed, since both the differences and the x offset will be small.

## THE IMPLEMENTATION

To use this technique, we need the difference table. I've written a general function to take a table of numbers and

generate a difference table. The function is shown in Listing 3. I wrote the function to be general-purpose because it's a handy thing to have in your toolbox, and we can use it in many other applications.

Armed with this difference table, we can write a sine function that includes interpolation. The result is shown in Listing 4. Using only three lines of executable code, it's almost as fast as the simpler version and much more accurate.

How accurate? Well, I have a confession to make. In "Look It Up," I gave you a formula for computing the error, but it was wildly optimistic. The correct formula is:

$$\varepsilon_{\max} \approx \frac{\Delta x^2}{8} \qquad (6)$$

where Δx is measured in radians. For our table of 1,024 points, this gives us a maximum error of 0.000001, which is much better than the 16-bit accuracy we can expect from our fixed-point representation. As a matter of fact, we could and should reduce the table size to 512 points, since that number still gives us 16-bit accuracy.

We could take this approach even further by adding second-order differences and even third-order differences, but I see no particular need to. A higher order algorithm lets us reduce the table size even further, at the expense of slower execution time. But the whole point of the tabular approach is to get fast execution, and a total storage of two kbytes seems a small price to pay.

For the record, I've settled on the algorithm in Listing 4 as the best implementation of the sine function for embedded systems. I hope you'll agree.

You may well be wondering what to do about the cosine function. Most applications require both functions. The following is the cosine function, in its entirety:

```
int cosine(unsigned int x){
  return sine(x + 0x4000);
}
```

That's it. We use an identity to convert the cosine to a sine, thus avoiding the need for a second table. If the idea of nested function calls for a high-speed implementation bothers you, you can always duplicate the code of sine(), but add the offset to x.

## THE SERIES APPROACH

Having told you I've decided the table lookup is a better solution than the power series one, we'll now look at the power series. Because the lookup works so well, this may seem rather pointless, but the series approach gives us a chance, as well as a reason, to exercise our skills with scaling, which is the purpose of this month's column. Also, the business of selecting the right quadrant, difficult enough in floating point, can be really tricky in fixed point, so I wouldn't feel I had given you the whole story without covering both approaches.

The sine function can be computed from its Taylor series:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots \qquad (7)$$

The corresponding series for the cosine is:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots \qquad (8)$$

As you can see, the two series have the same form, except that the sine function contains only odd terms, and the cosine function contains even ones. Though we're seeking a function for the sine here, we'll find that we need both series. That's because we must truncate the series to a few terms each, to get reasonable performance.

Theoretically, the terms in the series continue out to infinity. In that theoretical world, the series always converge to exactly the correct answer for any angle, even huge ones. In the real world, however, we can't afford to wait while our computer evaluates and sums an infinite number of terms. The minute we agree to chop the series off at some number of terms less than infinity, we're accepting the fact that we now have an approximation. From the form of the series, it's clear that the quality of this approximation depends on the size of x. Thus, to keep the number of terms small, we must limit the range of x.

We can calculate how many terms we'll need for a given range and accuracy. To a good (and conservative) approximation, the error caused by truncating the series is equal to the size of the first omitted term. This term is always of the form:

$$\frac{x^n}{n!}$$

In the scaling we've chosen, we'd like the error to be less than one in the lowest order bit. For the B0 scaling we used, the low-order bit has a weight of $1 \times 2^{-15}$, so we can write the inequality requirement:

$$\frac{x^n}{n!} \le 2^{-15} \qquad (9)$$

Using the equality, we can solve this equation for x. The results are shown in Table 2. The even numbers correspond to the sine series, and the odd numbers

**LISTING 3**
*Building a difference table.*

```
void make_diff(int table[ ], int
diff_table[ ], int n)
{
  for(int i=0; i<n; i++){
    int j=(i+1)%n;
    diff_table[i] = table[j] -
      table[i];
  }
}
```

**LISTING 4**
*First-order interpolation.*

```
int sine(unsigned int ang){
  int i = ang >> SHIFT;
  int result = sine_table[i];
  result += (diff_table[i] * (ang &
    MASK)) >> SHIFT;
  return result;
}
```

correspond to the cosine series. From those error values, we can decide how many terms are needed for a given range. Those numbers are given in Table 3. As you can see, we have quite a bit of incentive for limiting the range; three terms are a lot easier to handle than eight.

## LIMITING THE RANGE

The question remains, how do we limit that range? After all, we can't control what angle someone gives us. We must somehow reduce a potentially large input angle to one that's small enough so the series converges well. We do it by breaking the angle up into two parts:

$$x = n\theta + \delta \tag{10}$$

where $\theta$ is some nice, simple angle like 45˚ or 90˚, and $\delta$ is whatever's left. This process perfectly parallels what we did for the table lookup case; we let the high bits of x define the index i (now we're using n), and the low bits represent the "vernier correction." In the lookup case, that fractional angle went into the interpolation formula. This time, it will go into the series approximation. The double-angle formulas give us the relationships we need:

$$\sin(n\theta + \delta) = \sin n\theta \cos\delta + \cos n\theta \sin\delta \tag{11}$$

$$\cos(n\theta + \delta) = \cos n\theta \cos\delta - \sin n\theta \sin\delta \tag{12}$$

Because $\theta$ is a constant, we can tabulate (or otherwise represent) the functions of $n\theta$, so we only need to evaluate the series for $\delta$.

The values of the constant parts are particularly easy if we let $\theta = 90˚$. For this case, the functions are:

| n | angle | sine | cosine |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 1 | 90 | 1 | 0 |
| 2 | 180 | 0 | -1 |
| 3 | 270 | -1 | 0 |

The corresponding equations are:

$$\sin(0 + \delta) = \sin\delta$$
$$\cos(0 + \delta) = \cos\delta \tag{13}$$
$$\sin(90 + \delta) = \cos\delta$$
$$\cos(90 + \delta) = -\sin\delta$$

The nice part about this choice for $\theta$ is that we only need to evaluate one series; we only must decide which one and what sign to attach to the result. You can see, though, why we need both sine and cosine series.

I'll just mention in passing another possibility, which is to let $\theta = 45˚$. For n = 1, we find:

$$\sin(45 + \delta) = 0.70711(\cos\delta + \sin\delta)$$
$$\cos(45 + \delta) = 0.70711(\cos\delta - \sin\delta) \tag{14}$$

**TABLE 2**
*Max range of argument.*

| n | x, degrees | No. terms |
|---|---|---|
| 1 | 0.00175 | 1 |
| 2 | 0.448 | 2 |
| 3 | 3.253 | 2 |
| 4 | 9.426 | 3 |
| 5 | 18.66 | 3 |
| 6 | 30.32 | 4 |
| 7 | 43.85 | 4 |
| 8 | 58.80 | 5 |
| 9 | 74.85 | 5 |
| 10 | 91.74 | 6 |
| 11 | 109.3 | 6 |
| 12 | 127.4 | 7 |
| 13 | 146.0 | 7 |
| 14 | 164.8 | 8 |
| 15 | 184.0 | 8 |
| 16 | 203.5 | 9 |

**TABLE 3**
*Terms needed.*

| angle range (deg) | sine | cosine |
|---|---|---|
| 180 | 8 | 8 |
| 90 | 5 | 6 |
| 45 | 4 | 4 |
| 22.5 | 3 | 3 |
| 11.25 | 2 | 3 |

For this case, we must evaluate both series, and the logic is quite a bit more complicated, since Equation 13 still holds when n is even. We won't use this option here, but you might want to file it away for future reference. It can be useful when very high accuracy is needed, because it limits the range of $\delta$ to only 22.5°.

To achieve the smallest magnitude for $\theta$, we'd like it to vary positive and negative around zero. This means we should split up the angular range into quadrants as shown in Figure 1a. Unfortunately, simply masking the binary word representing x doesn't do that; it gives us the divisions of Figure 1b. That's easily fixed, though. We add 45° to the angle, split it up, and then subtract the 45° again. The following code does the job:

```
x += 0x2000;
int n = (x >> 14) & 0x3;
x &= 0x3fff;
x -= 0x2000;
```

The resulting values of n will be 0, 1, 2, or 3, and x will be in the range ±0x2000, which is 45° in pirads. A simple case statement on n gives us the rest.

## EVALUATING THE SERIES

We've now reduced the problem to that of evaluating the series. We've already shown the number of terms in the series in Table 3. We'll need three terms for the sine and four for the cosine. Here are the series, properly truncated:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \qquad (15)$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} \qquad (16)$$

One caution: Never, ever evaluate the series in this form! We don't want to be computing fifth powers of anything, especially in fixed-point arithmetic. The correct approach is to factor them into Horner's form:

$$\sin x = x\left(1 - \frac{x^2}{6}\left(1 - \frac{x^2}{20}\left(1 - \frac{x^2}{42}\right)\right)\right) \qquad (17)$$

$$\cos x = 1 - \frac{x^2}{2}\left(1 - \frac{x^2}{12}\left(1 - \frac{x^2}{30}\right)\right) \qquad (18)$$

For the record, the need for that fourth term in sin x is debatable. In practice, omitting the fourth term changes the result at most by one in the low-order bit. You may well decide that it's not worth the bother to get that extra bit. I'm including it here, because it's easier to take things out than to put them in.

We're almost ready to commit to code. However, one small bit of business still remains. The series of Equation 17 and Equation 18 are only valid when x is measured in radians, but our angle is measured in pirads. To fix this, we must multiply the argument by $\pi$. We have two choices. We can perform the conversion going in, which is certainly the more straightforward and easy-to-understand approach. Alternatively, we can combine the conversion factor, which would be $\pi^2$, into the coefficients for each term. I've tried this approach, hoping to save a multiply or two. I see nothing to recommend it; we end up saving no computations, and the coefficients and scaling are much messier.

More interesting is the issue of how to handle the ones. Remember, we can't represent a "1" in B0 scale, we can only express it as the next smaller number, 0x7fff. We can avoid this problem by representing the one as 0x4000 B-1, but in doing so, we lose a bit of precision. I've chosen here to represent a one by 0x7fff, the same way we do for the limiting value of the functions. This doesn't seem to cause any problems in practice; we still get good accuracy throughout the range.

## CHOOSING SCALINGS

The conversion from pirads to radians typifies the kind of things we must deal with in fixed-point computations, so let's look at it in detail. The input angle, in pirads, is scaled B0. The value of $\pi$ is greater than two, so we must scale it B2. In this scale:

$$\pi = 0x6488,\ B2 \qquad (19)$$

Normally, we'd expect that if we multiply two numbers scaled B0 and B2, the result must also be B2. But remember, we've already reduced x to a range of ±45° or ±0x2000. This number could be scaled B-1 and just misses fitting in B-2. In fact, the product $\pi/4$ radians is less than one and fits nicely into a B0 scale, so that's the way we'll do it. After the multiplication, we shift right 15 bits, less the number that the scale is being changed. Since we're switching from B2 to B0 scale, the proper number of shifts is 13. The corresponding line of code is:

```
int xr = (int)(((long)x * 0x6488) >> 13);
```

I must apologize for the seemingly gratuitous complexity of this line. You

**FIGURE 1**
*Sectors.*

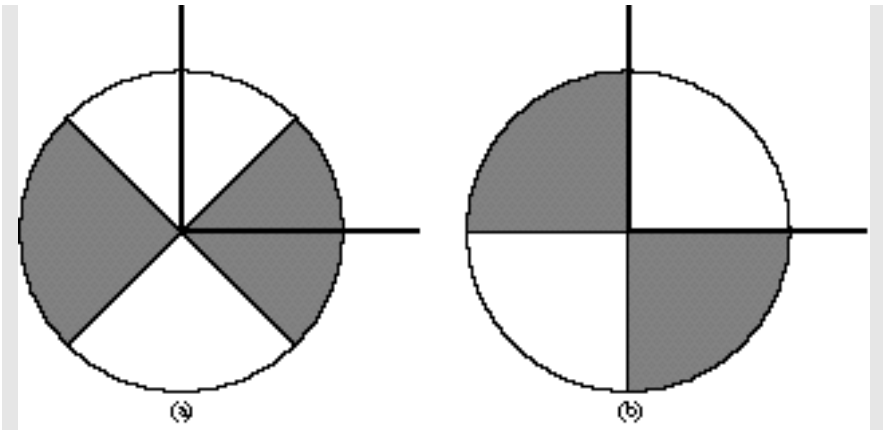

**TABLE 4**
*Coefficients.*

| Sine | | |
|---|---|---|
| **term** | **value** | **scale** |
| 1/42 | 0x6186 | B-5 |
| 1/20 | 0x6666 | B-4 |
| 1/6 | 0x5555 | B-2 |
| **Cosine** | | |
| **term** | **value** | **scale** |
| 1/30 | 0x4444 | B-4 |
| 1/12 | 0x5555 | B-3 |

can thank the C syntax for it. With a language that supported fixed point, we should be able to write:

```
xr = x * pi;
```

Sadly, we don't have such a language, and that's that.

To complete the computation of the series, we need the coefficients at each step. They're shown in Table 4. Armed with the coefficients and the rules for fixed-point multiplication, we can generate the code shown in Listing 5. I've also included the driver functions that reduce the angle to the proper range and deal with the different quadrants.

You now have a complete set of functions for generating sines and cosines in fixed point. Tuck them into your toolbox and enjoy them. Before you do, however, please take the time to read and understand them, because that was the main reason for this exercise. In particular, notice the subtraction terms. Since our ones are all scaled B0, we must make sure that the result of the previous multiplications has the same scale. This is accomplished by shifting right more places than would be needed just to hold the product.

## AN EASIER WAY

Before we part, I'd like to show you an alternative implementation that's quite a bit simpler. In Listing 5, I deliberately showed you the classical implementation in which we multiply by coefficients representing 1/42, 1/20, and so on. This is the way these functions have always been done, mainly because for most computers, division is a more expensive operation. Many of the small CPUs we're likely to find in an embedded system don't even support division. On the other hand, if we use the division operation, we get two nice advantages. First, since we're dividing by pure integers and not by fixed-point numbers, we can use the ordinary integer division with no rescaling. This fact alone probably makes the division more efficient than an equivalent multiplication, followed by shift. Also, we don't have to typecast in two directions. Secondly, and this is important, all scaling turns out to be B0, so we don't need to do strange shifts to get things right. I think you'll agree that the alternative code of Listing 6 is quite a bit cleaner and clearer than the earlier version that uses multiplication.

This approach has the disadvantage that we can't tweak the coefficients using Chebyshev polynomials (to be covered in an upcoming column). Until we're ready to do that, I'm sticking with the division approach. ESP

**LISTING 5**
*Sine/cosine series.*

```
int sin(int x)
{
  x += 0x2000;
  int n = (x >> 14) & 0x3;
  x &= 0x3fff;
  x -= 0x2000;
  switch(n){
  case 0: return _sin(x);
  case 1: return _cos(x);
  case 2: return - _sin(x);
  case 3: return - _cos(x);
  }
  return 0;
}

int cos(int x){
  return sin(x + 0x4000);
}

int _sin(int x){
  int fcn;
  int xr=(int)(((long)x*0x6488) >> 13);
  int x2=(int)(((long)xr * xr) >> 15);
  fcn=(int)(((long)x2 * 0x6186) >> 20);
  fcn=0x7fff - fcn;
  fcn=(int)(((long)x2 * fcn) >> 15);
  fcn=(int)(((long)fcn*0x6666) >> 19);
  fcn = 0x7fff - fcn;
  fcn = (int)(((long)x2 * fcn) >> 15);
  fcn=(int)(((long)fcn*0x5555) >> 17);
  fcn = 0x7fff - fcn;
  return (int)(((long)xr*fcn) >> 15);
}

int _cos(int x){
  int fcn, xr, x2;
  xr=(int)(((long)x * 0x6488) >> 13);
  x2 = (int)(((long)xr * xr) >> 15);
  fcn=(int)(((long)x2 * 0x4444) >> 19);
  fcn = 0x7fff - fcn;
  fcn = (int)(((long)x2 * fcn) >> 15);
  fcn=(int)(((long)fcn*0x5555) >> 18);
  fcn = 0x7fff - fcn;
  fcn = (int)(((long)x2 * fcn) >> 16);
  return 0x7fff - fcn;
}
```

**LISTING 6**
*Using division.*

```
int _sin(int x){
  int fcn;
  int xr=(int)(((long)x*0x6488) >> 13);
  int x2=(int)(((long)xr * xr) >> 15);
  fcn = 0x7fff - x2/42;
  fcn = (int)(((long)x2 * fcn) >> 15);
  fcn = 0x7fff - fcn/20;
  fcn = (int)(((long)x2 * fcn) >> 15);
  fcn = 0x7fff - fcn/6;
  return (int)(((long)xr*fcn) >> 15);
}

int _cos(int x){
  int fcn;
  int xr=(int)(((long)x*0x6488) >> 13);
  int x2=(int)(((long)xr * xr) >> 15);
  fcn = 0x7fff - x2/30;
  fcn = (int)(((long)x2 * fcn) >> 15);
  fcn = 0x7fff - fcn/12;
  fcn = (int)(((long)x2 * fcn) >> 15);
  return 0x7fff - fcn/2;
}
```

*Jack W. Crenshaw is a staff scientist at Invivo Research in Orlando, FL. He did much early work in the space program and has developed numerous analysis and real-time programs. He holds a PhD in physics from Auburn University. Crenshaw can be reached at 72325.1327@compuserve.com.*