<BACK

by Jack W. Crenshaw

Integer Arithmetic

Over the last few months, we've discussed rational fractions as ways to represent irrational numbers such as π and the golden ratio. This topic leads to one of representing variables that are naturally floating-point numbers, such as position and velocity, using integers. The motivation is simple enough: Floating-point arithmetic tends to be slower than integer arithmetic, sometimes much slower. Often, it's far slower than we can tolerate in an embedded system.

The problem is not as severe as it used to be. Unless you've been living on Mars, you're well aware that the hotter chips, such as Intel's 80486 and Pentium and Motorola's 68040 and higher, have built-in floating-point processors. Aerospace processors such as the MIL-STD-1750A also use them, and in some cases, the processors are blazingly fast. The Fairchild (now Intergraph) Clipper can perform a floating-point multiply and add in 10 nanoseconds! Even the old numeric processors in the 1750As were so fast that it was often faster to convert an integer to floating point, operate on it, and convert it back rather than to try to use integer arithmetic.

Back in the real world, not all of us are using Pentiums, 486s, or 68040s. Some of us are still programming 68HC16s or, heaven forbid, 8051s. Heck, some of us are still programming Z80s and 6502s! Even if you're lucky enough to get a hot chip, when the systems engineers begin looking at their budgets for cost, power, and board real estate, the floating-point unit is always an easy target. Heck, we don't need that, do we? The software folks can always compensate using software.

Someday, we will all program real-time systems that have lightning-fast numeric processors so entangled with the CPU that designers couldn't leave them out even if they wanted to. But for now, we have to live with the realities all real-time programmers must deal with. The only alternative is to represent real numbers using integer arithmetic.

Unless you've been living on Mars, you're well aware that the hotter chips have built-in floating-point processors.

AN EASY START

Regular readers know that I like to sneak up on new problems, starting from very simple and familiar examples and working my way from there. In this case, we have an excellent start on a problem we touched on last month: how to represent an angle and its trigonometric functions. An angle begins at zero and increases smoothly through positive values until it gets to 180° or 2π radians. At that value, we can think of the angle as being either +180° or -180°. The angle is the same either way. From there, we can either think of the angle as increasing through ever more positive values, such as 270°, or decreasing through negative values, such as -90°. Eventually, the angle reaches its origin again, wrapping around to 360°, which is the same as zero.

Fortunately, the same behavior is displayed by the two's-complement integers that computers use. The 16-bit value 0x8000 can be thought of as either the unsigned integer 32,768 or the signed integer -32,768. Similarly, 0xc000 can be thought of as either 49,152 or -16,384. Continuing, we finally reach 0xffff, which is either -1 or 65,535, depending on your point of view. Add one to this number and you're back at zero.

This behavior leads us quite naturally to represent an angle as a scaled integer, with the scale factor being whatever it takes to make the angle wrap to zero at the same place the integer does. In this particular case, we have absolutely no choice in the matter; only one scale factor is possible. For a 16-bit integer, the correspondence is:

360° <=> 65536 units

So the scale factor is:

$$\frac{65536}{360} = 182.044444 \text{ units/degree} \quad (1$$

From radians, the conversion is:

$$\frac{65536}{2\pi} = 10430.378 \text{ units/radian} \quad (2$$

If we must, we can work out similar scale factors for other word lengths. It's almost never necessary, though. An eight-bit integer is usually too course, giving us worse than one degree slop in resolution. At the other extreme, a 32-bit integer is massive overkill. With 16 bits, we have over 180 counts per degree, so our resolution, which is the angle change caused by a single bit change, is less than 20 seconds of arc. That's the angle subtended by a quarter held up at a distance of three football fields. Unless you're doing precision navigation or pointing a telescope, that resolution is good enough. Even if you'd like to have more, it's very doubtful that you'll find a sensor that can give you more bits. With 32 bits, the resolution is equivalent to 0.0003 seconds of arc, which is the angle subtended by a human hair seen end-on at a distance of 11 miles. That's overkill.

Assuming we'll use a 16-bit number to represent angles, we have the equivalence shown in Table 1, which some of you have seen before. The unit of scale is sometimes called a pirad (for pi radians), since that's the value corresponding to the full range of a signed integer.

SCALING PIRADS

Once we have the angle in its internal representation, we don't need to convert it. Any arithmetic we must do, such as adding or subtracting two angles, computing changes or rates, and so on, can be done just as though the angle were an ordinary signed integer. Or unsigned, for that matter. Unfortunately, we humans like to see angles displayed in degrees, or sometimes degrees, minutes, and seconds.

Suppose we only need to express the angles to the nearest degree. You've already seen the conversion factor in Equation 1, but that factor is not an integer. How do we do the conversion if we can't use floating point?

TABLE 1
Angles in pirads.

Angle, degrees	Hex equivalent
0	0000
22.5	1000
45	2000
67.5	3000
90	4000
112.5	5000
135	6000
157.5	7000
180 = -180	8000
202.5 = -157.5	9000
225 = -135	a000
247.5 = -112.5	b000
270 = -90	c000
292.5 = -67.5	d000
315 = -45	e000
337.5 = -22.5	f000
360 = 0	0000

Resolution:
1 count = 19.77 arc-sec
1 degree = 182.04444 counts

If you were paying attention last month, you already know the answer. We originally wrote the conversion as a rational fraction. The solution is to leave it in that form. Reduced to its simplest form, the ratio of Equation 1 is:

$$k = 8192/45 \qquad (3)$$

To convert a number from pirads to degrees, multiply it by 45, then divide by 8,192 (which you can do by shifting 13 bits). The following two lines of code show you how. We must use a long integer to hold the intermediate product, but the end result fits back into an integer.

```
temp = (long)angle * 45;
angle = temp >> 13;
```

To convert the other way, simply reverse the process:

```
temp = (long)angle << 13;
angle = temp / 45;
```

If you want to display the angle in degrees and hundredths, just multiply by 4,500 instead of 45. The long integer format has plenty of room for that.

What you do with the angle from there depends on your display capabilities and your needs. Most often, you'll want to be able to display the angle. If your compiler supports floating point and you can afford the size overhead associated with loading the floating-point library, the most straightforward approach is simply to convert the integer value to an equivalent floating-point one:

```
float f_angle = (float)temp /100;
```

You needn't worry about processor performance for this application. Since the number is only needed for display, we don't really care how long it takes to compute it. The computation can be done at a low rate or in the background.

The worst-case situation is one in which you must display the angle without the help of the floating-point routines. Perhaps the integer and fractional parts must be sent to different display devices. We can't get too specific here because so much depends on the hardware configuration, but I can at least show you how to separate out the two parts:

```
temp = ((long)angle * 4500) >> 13;
if(temp < 0)
  temp += 36000;
angle = temp / 100;
fraction = temp % 100;
```

In this case, it's best to convert negative angles to positive ones to avoid trouble with the modulo function.

Displaying the number in the range -180°..180° is much trickier, because we must handle both positive and negative values for both parts of the integer. Listing 1 shows a function that does the job. The function returns three values: the whole part, the fractional part, and a separate sign flag.

What's that you say? You need a display in degrees, minutes, and seconds? No problem. Just use a different conversion factor:

$$\begin{aligned} k &= \left(\frac{45}{8192}\right) * 3600 \\ &= \frac{45 * 225}{512} \\ &= \frac{10125}{512} \qquad (4) \end{aligned}$$

The reduction by the greatest common divisor is absolutely necessary or the intermediate product won't fit in 32 bits. Listing 2 gives the resulting code.

SCALED INTEGERS

Now we know how to convert angles from degrees to pirads and back and how to display them in the degrees, minutes, seconds format if we need to. It's unlikely, but marginally possible, that you may want to convert to radians and back, as well. (Some mathematical operations, such as Kepler's equation for orbital mechanics, require the angle to be in radians.) That's no problem, either, because by definition, a pirad is equal to π radians. To convert from pirads to radians, simply multiply by π. You know by now that π can be written as the rational fraction:

$$\pi = \frac{355}{113} \tag{5}$$

There is the small matter of scaling. After we've multiplied by π, the result will no longer fit in 16 bits. To make it do so, we must shift right two bits. We can do so by making the denominator 4 * 113, or 452. Thus, our conversion for this case shoud be:

$$k = \frac{355}{452} \tag{6}$$

Keep in mind that we're talking about scaled integers. We've effectively moved the decimal, or rather binary point, to afford the best accuracy we can get. Consider the representation of 180° in its pirad form. In hexadecimal arithmetic, this number is coded as 0x8000. But 0x8000 is equal to 32,768. Does this mean that there are 32,768 pirads in 180°? Hardly. There's only one in 180°, and only two in the whole circle. We've scaled the number to get the maximum resolution for the range needed. We can think of this number as a fixed-point number, with the binary point located between bits 14 and 15, as shown in Figure 1. Any bits above this binary point are integer bits (in this case, there can be only one). All other bits represent successively larger negative powers of two as we move to the right. The least significant bit represents 2^{-15} pirads, which is equivalent to 0.0055° or about 20 seconds of arc, as we saw earlier.

LISTING 1
Angle formatting.

```
void to_degrees(int angle, int & sign,
  int & whole, int & fraction){
  long temp = (long)angle * 4500;
  temp >>= 13;
  whole = temp /100;
  fraction = temp % 100;
  if(temp < 0)
  {
    sign = -1;
    whole = -whole;
    fraction = (100 - fraction) % 100;
  }
  else
    sign = 1;
}
```

LISTING 2
Conversion to degrees, minutes, and seconds.

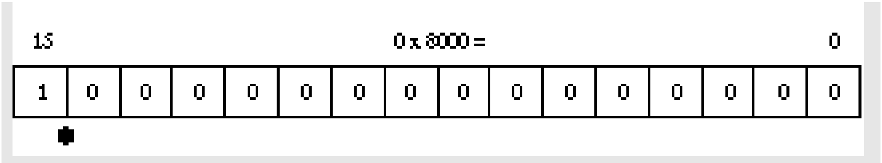
```
void dms(int angle, int & deg, int &
  min, int & sec){
  long temp = (long)angle * 10125;
  temp >>= 9;
  if(temp < 0)
    temp += (long)360 * 3600;
  deg = temp / 3600;
  temp %= 3600;
  min = temp / 60;
  sec = temp % 60;
}
```

It's worth noting that the one non-zero bit in Figure 1 is in bit 15, which is normally reserved for the sign bit. 0x8000 is the numerically largest negative integer we can represent, corresponding to -32,768. Again, we can think of the angle as either a signed or unsigned integer. As long as the angle is measured in pirads, the distinction doesn't really matter, because we deliberately chose the scaling to make the integers wrap around at the same point the angles do. In general, however, this won't happen. For example, consider the same number expressed in radians, which we get by using the conversion factor from Equation 6. When we multiply 0x8000 by this number, we get 0x6487. Because we had to shift the binary point two places, it now rests between bits 12 and 13, as shown in Figure 2. We have three integer bits, which are now displaying 3, the integral part of π. Doubling this number should give us an angle of 2π radians. In hex, the number is 0xc90e. This is again correct—the high three bits represent a six and the converted value is indeed 6.28318. That sure isn't zero, is it? We didn't automatically wrap to zero—that convenience is reserved for the case where the angle is expressed in (scaled) pirads, which of course is why we chose to use that representation in the first place. For the same reason, the equivalence of signed and unsigned representations only works for angles in pirads. In other cases, we must either make the numbers unsigned or leave room for a sign bit.

FIXED POINT

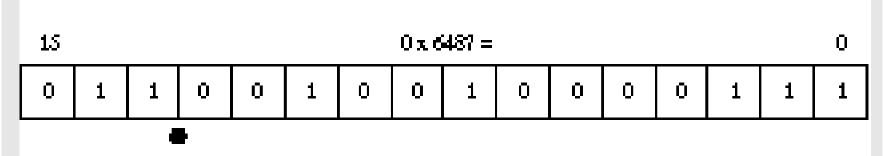
It's important that you understand how we got here. We began, you'll recall, with a very special kind of number, the angle measured in pirads. We didn't really discuss where the binary point was, we just chose the scaling to give us a wraparound at 360°. But we see now that this representation is equivalent to a number that has both integer and fractional parts, with the binary point located between bits 14 and 15. For convenience, I'll say that the binary point is located at bit 15 and leave it up to you to understand that I mean at

FIGURE 1
One pirad.



1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

FIGURE 2
One radian.



0	1	1	0	0	1	0	0	1	0	0	0	0	1	1	1

the right-hand side of that bit.

We then talked about the conversion to radians and realized that the converted number would no longer fit into a 16-bit integer. We got around that by shifting the binary point two more bits to the right.

These two cases are examples of numbers using what is called fixed-point notation. Internally, the computer's arithmetic unit sees them as integers, but we agree to interpret them as fractional numbers with the binary point wherever it must be. Usually, we try to place this point as far to the left as we can, to get the maximum resolution by maximizing the number of bits to the right of the binary point.

This concept is the keystone to any real-time processing that involves measuring and dealing with parameters that, in the real world, are not integers. In an ideal situation, we'd use floating-point notation and let the computer take care of the details. When we can't afford to because of performance constraints, we do the next best thing, which is to use fixed-point notation.

The notation is both the boon and the bane of every real-time programmer's existence. We could hardly get anything done without the notation, but its use adds immeasurably to the difficulty of programming. It's one of the major reasons real-time programming costs so much. We can't just stuff a value into a variable and go compute with it. As you'll see, the outcome of arithmetic operations depends very much on how we interpret the numbers.

Fixed-point notation is almost certainly the most common source of errors in real-time programs. Some of these errors can be quite subtle and difficult to track down. When we choose the representation for a given variable, we must set bounds to the values we expect to see. If, in operation, the value exceeds those bounds, you won't get a nice pop-up dialog box telling you that you just went out of range—dialog boxes rarely show up in missile launchers or factory control systems. Instead, a large positive number will silently go negative, and you'll start to get goofy results. If you're lucky, the results will be insanely goofy, and it will be apparent to everyone that something is terribly wrong. Hopefully, this revelation will occur while the system is undergoing integration testing and not while it's steering your family to Boise, Idaho.

If you're not lucky, the error will affect the final result only slightly, and you'll be left wondering where the gremlins are hiding.

If you're not lucky, the error will affect the final result only slightly, and you'll be left wondering where the gremlins are hiding. Or, you'll be ducking for cover.

Things get even worse if requirements change during development. The range you expect for a given variable may not be the same at the end of the project as it was at the beginning. If and when the range changes, so should the representation used for that variable change. And if it changes, so must every other variable computed from it. Thus, one change tends to ripple through the whole system, sometimes to the extent that it changes the design. You may find yourself rewriting the code several times during the course of a single project, just to keep the scaling correct. This is one area where you can't anticipate changes or allow for growth. The need to get as much accuracy as possible dictates that you don't waste any bits or leave padding in the representation. You must use the format that best fits the problem, and if the problem changes, so be it.

THE B-NOTATION

If we're going to use fixed-point representations, as we almost certainly must, we need every aid we can think of to make the job easier. One such aid is convenient and compact notation. Over the years, the following notation has emerged. We'll begin with the format of Figure 1, where the binary point is at bit 15. This is the natural format when we can be sure that the number is less than (but just less than) one. In other words, it's a pure fraction. Bit 15 is used strictly as a sign bit. (I know it's used differently, or at least can be viewed differently, for the special case of an angle in pirads. But this case is not the norm.)

This particular format, where the number is treated as a pure fraction, is the most common. We'll call this format B0 (B for binary point). The format of Figure 2 has the binary point shifted right two bits. We'll call this format B2. In general, the number following the B tells us how many bits the binary point is offset from its reference position at bit 15. According to this notation, a pure integer, which has its binary point at bit 0, has a format of B15.

Every increase in the B value shifts the binary point to the right, and a decrease shifts it to the left. The range is not limited in either direction—we can go both larger than 15 and less than zero (and thus negative). A format of B-5 has an imaginary binary point at bit 19. Though this bit doesn't physically exist, we can still treat the number as though it did. Negative offsets are useful when the numbers are small, and we can be sure they will never approach one. Positive offsets beyond 15 are used for very large numbers.

In practice, you should choose the scaling that can just contain the number, given the expected range. The resolution is then given by the weight given the least significant bit. For a 16-bit word with scaling Bn, this resolution is equal to:

$$\text{resolution} = 2^{-(15-n)} \qquad (7)$$

For your convenience, I've shown

the range and resolution for various scalings in Table 2. If your number is unsigned, you can double the range but not the resolution. As you can see from the table, the maximum positive value is never quite equal to a power of two. Because the positive and negative limits for integers are not quite symmetrical, neither are those of the scaled numbers. In most cases, it's so close that you can fix the range at equal values. In this particular case, the range is approximately:

$$\text{range} = 2^{15-n} \qquad (8)$$

There are a few cases where the difference between the approximate and exact range is critical. We'll look at one example next month.

For a given word length, once you've chosen the scaling by fixing a range, you've also set the resolution. It's up to you to decide if this resolution is adequate for your application. If it's not, you must go to 32-bit integers. The formulas for range and resolution for these integers are just like those for 16-bit integers except you must substitute 31 for 15.

ALMOST FLOATING POINT

The observant reader will note that fixed-point notation has a lot in common with floating-point. Internally, the latter format breaks a number up into a numeric part, the mantissa, which is always less than one, and an exponent to tell us where the binary point must go. A number in fixed-point notation is very much like the mantissa of that floating-point number. The difference is that the exponent is not stored with the number. In fixed-point notation, the binary point is, well, fixed. Since it doesn't change, we don't need to record it, except in our notebook and program comments. By not carrying around the exponent, we avoid the need to compute it anew after each computation, and this is where we gain performance. You can see, however, that we gain performance at the expense of flexibility and robustness in the presence of overflow conditions. Floating-point arithmetic always adjusts the result to give us maximum accuracy while avoiding overflow. In fixed point, this doesn't happen. We must choose a fixed scaling in our design process and live with it. That's why we're better off using floating point if we have the computing power to spare. It's also why computer manufacturers are working hard to make chips that will give us that power.

We'll continue this discussion next month. Before I go, I'll leave you with a convenient rule of thumb to help you deal with fixed-point scaling. To convert a number in its Bn notation to floating point, divide by:

$$2^{15-n} \qquad (9)$$

For example, our 180˚ angle was represented as 0x8000, format B0. Divide by 2^{15}, which is 32,768, to get 1.0000000 pirads, which is correct.

Similarly, the same angle in radians was 0x6487, format B2. Divide by 2^{13}, which is 8192, to get 3.14148, or roughly π radians. This is also correct, within the limits of 16-bit accuracy.

In the same way, we can convert any number from its floating-point form to its Bn form by multiplying by the factor of Equation 9. After a little practice, you'll find that you can do these conversions very rapidly.

Next month, we'll continue with this topic and look at what we must do to perform arithmetic with fixed-point numbers. See you then.

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 enjoys contact and can be reached via e-mail at 72325.1327@compuserve.com.

TABLE 2
Range and resolution.

Scaling	Range, +/-	Resolution
B-5	0.031249046	0.000000953
B-4	0.062498092	0.000001907
B-3	0.124996185	0.000003814
B-2	0.24999237	0.000007629
B-1	0.499984741	0.000015258
B0	0.999969482	0.000030517
B1	1.999938965	0.000061035
B2	3.99987793	0.000122070
B3	7.999755859	0.000244140
B4	15.99951172	0.000488281
B5	31.99902344	0.000976562
B6	63.99804687	0.001953125
B7	127.9960937	0.00390625
B8	255.9921875	0.0078125
B9	511.984375	0.015625
B10	1023.96875	0.03125
B11	2047.9375	0.0625
B12	4095.875	0.125
B13	8191.75	0.25
B14	16383.5	0.5
B15	32768	1
B16	65536	2
B17	131072	4
B18	262144	8
B19	524288	16
B20	1048576	32