<BACK

By Jack W. Crenshaw

Fixed-Point Arithmetic

A funny thing happened on the way to this column. Those of you who were here last month will recall that I had just begun a discussion of how to implement fixed-point arithmetic using integers. Last week, I got my March issue of *Embedded Systems Programming*, and discovered, to my horror, a nice article by Jean Labrosse on how to implement fixed-point arithmetic using integers. This is the ultimate writer's nightmare, somewhat akin to my all-time favorite of remembering on the last day of finals that I was signed up for a class I had never attended. Or looking at the conference program to discover that the person speaking before me is covering the same topic.

What do we do now? I don't want to bore you with repetition. But this is the second half of a two-part article, and aborting the discussion halfway through would leave a whole train of thought dangling. Neither solution really satisfies.

Fortunately, after rereading Labrosse's article, I see that the way he presents the subject is different enough that we don't have that much overlap. I'm going to press on with the discussion and give you my take on the subject. It might help if you think of us as having the foresight and coordination to provide you with saturation coverage of the subject from two carefully synchronized points of view.

SNEAKING INTO IT

Last month, I sneaked up on the subject of fixed-point arithmetic through the problem of expressing an angle using an integer representation. I showed you that if we measure the angle in pirads (π radians) scaled properly, we get a representation that naturally wraps around as the angle gets larger and passes through zero.

Next, I introduced the B-notation (similar to Labrosse's S-notation.) This notation describes the location of the imaginary binary point, with B0 corresponding to a number with the binary point at bit 15, just below the sign bit. (We're assuming 16-bit integers here, but the notation is easily extended to longer word lengths.) Thus, the hex integer 0x1256, scaled B0, is equivalent to the binary number:

In general, we want to use the scaling that will give us the best accuracy that we can get without overflowing.

0.001001001010110 = 0.143249511

Every increase in the scale factor n moves the binary point to the right one bit. A decrease moves it to the left. Any 16-bit number scaled B15 is a pure integer, since the binary point lies just to the right of the least significant bit. Last month, I showed how to convert a number from its integer to decimal values and back, using the factor:

$$m = 2^{15-n} \qquad (1)$$

I also showed you how to compute the range and resolution for any scaling, using the relationships:

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

$$\text{range} = 2^{15-n} \qquad (3)$$

To handle fixed-point notation, it's very important that you feel comfortable with these relationships and can toss them around with dexterity. Each of us must decide how best to do this, but I tend to do the conversion in two steps. Equation 1 is fine for computer programming, but I can remember the conversion better if I separate out the two parts of the exponent. 2^{15} is equal to 32,768. To effect the conversion from integer to decimal, divide by 32,768, then multiply by n powers of two. To go the other way, reverse the process. A couple of examples will help. To convert 0x5678, B4, to decimal, do the following:

- Convert hex to decimal 22136
- Divide by 32,768 0.675537109
- Multiply by 2^4 10.80859375.

To convert 1,000.0 to B10:

- Divide by 2^{10} = 1,024 0.9765625
- Multiply by 32,768 32,000
- Convert to hex 0x7d00.

To get the feel of the process, practice on the examples in Exercise 1. The answers are on page 16.

EXERCISE 1

Conversion practice.

From Hex to Decimal:

1. 0x4000, B1 = ?
2. 0x0001, B15 = ?
3. 0x6667, B4 = ?
4. 0xaaab, B1 = ?
   (hint: the number is negative)
5. 0x7fff, B4 =?

From Decimal to Hex:

1. 3.14, B2 = ?
2. 5280.0, B13 = ?
3. 0.015625, B-6 = ?
4. 3.00 x 108, B29 = ?
5. -3 B4 = ?

If you didn't do the exercises, stop right now and do them. You will notice that when we convert from decimal to hex, the answer always fits nicely into four hex digits. Was this a lucky accident? Hardly. As a matter of fact, making the number fit is the trickiest and most important part of the scaling process. And, as usual, there's a trick that can make the process less painful.

Last month, I included a table giving the maximum range and best resolution we can expect from a given scaling. For those who like tables, I've reproduced it in Table 1. In general, we want to use the scaling that will give us the best accuracy, meaning the most bits, that we can get without overflowing. Suppose we know that a given distance will never exceed one mile, or 5,280 feet. Looking at the table, we can see that the range just large enough to hold this value is 8,192, or rather 8,191.75, which corresponds to a scaling of B13. This tells us what scaling must be used for this number.

The only problem is, most of us aren't likely to carry this table around with us all the time. How do we arrive at a scaling without the table? The trick is to recognize that the scaling Bn allows us to represent a number with n bits to the left of the binary point. If we know how many bits we need here, we immediately get the needed scaling. We can get this number by taking the logarithm, base two, of the number. For our example:

$$\log_2(5280) = 12.366 \qquad (4)$$

We must take the next larger integer, which is 13, giving us the needed scaling of B13. The same process works for small numbers:

$$\log_2(0.001) = -9.965 \qquad (5)$$

In this case, note carefully that the next larger integer is -9, not -10. So we can represent the number scaled B-9. (Its scaled value is 0x4189.)

Unfortunately, most calculators don't offer the logarithm, base two. But that's no problem, because we can get it in two steps:

$$\log_2 x = \frac{\ln x}{\ln 2} = \frac{\log x}{\log 2} \qquad (6)$$

To figure out how many bits to allow to the left of the binary point, first take the logarithm of the number (in either base e or base 10), then divide the result by the log of two (using the same base, of course). The result, rounded up, is your B-number.

It goes without saying that to do these computations, you need a calculator that can handle both decimal and hex numbers, as well as scientific functions like ln x. You can spend a lot of money for such a calculator and still not get the ability to flip between decimal and hex representations. My expensive Hewlett-Packard can't do it. Or you can buy something like the Sharp EL-506D, a neat little pocket calculator priced under $30. I can't get along without my Sharp. And no, I don't own any Sharp stock, nor have they paid me to endorse their product. I just know a good thing when I see it.

ANSWERS

For Exercise 1.

From Hex to Decimal:

1. 0x4000, B1 = 1.000
2. 0x0001, B15 = 1.000
3. 0x6667, B4 = 12.800
4. 0xaaab, B1 = -1.3333
5. 0x7fff, B4 = 15.9995

From Decimal to Hex:

1. 3.14, B2 = 0x647a
2. 5280.0, B13 = 0x5280
3. 0.015625, B-6 = 0x8000
4. 3.00 x 108, B29 = 0x4786
5. -3 B4 = 0xe800

TABLE 1
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	0.12
B13	8191.75	0.25
B14	16383.5	05
B15	32768	1
B16	65536	2
B17	131072	4
B18	262144	8
B19	524288	16
B20	1048576	32

WHAT'S YOUR SINE?

Now that we know how to represent numbers, including angles in pirads, it's only natural that we try to use them for something. An obvious and very, very practical use is to represent the trig functions, sine and cosine. Right away, we run into a brick wall, because these functions can vary from -1.0000 to +1.0000. Of all possible ranges, this is the most awkward, because for every angle except four, these functions are safely less than unity. We're tempted to use the B0 representation. Unfortunately, we can express 0.99997 in this notation, but not the limiting value of 1.0000. In B0 notation, 1.0000 is 0x8000, which looks like a negative number, not a positive one. Our representation can't hold the value.

We really have only two choices: We can scale the number to B1, recognizing that we lose a bit of precision in the process, or we can limit the trig functions so they never overflow. The first choice is the safe and simple one, and one I use when I am in a hurry. Most people, however, prefer to keep the resolution and limit the value of the function. Thus, we represent the 1.0000 value by 0x7fff, scale B0 and accept that three part in 100,000 error, which is all we can expect of the bit resolution we have, anyway. Table 2 gives a short table of the sine function of an angle in pirads. Later, we'll see how to compute this function.

In case you haven't noticed, the process of choosing a scaling for a given variable is tricky and time-consuming. It's also dangerous, since any overflow of the number is likely to be catastrophic. You must perform a balancing act between getting the most precision you can and having enough headroom to handle unexpectedly large values. When you're designing a new program, this process must be repeated for each parameter in the program, including the results of all intermediate computations.

In practice, a certain amount of artistry is required to balance safety against accuracy. For certain rare cases like the angle and its sine function of Table 2, the range is naturally limited, and you're safe. Or perhaps you're dealing with an input from, say, an analog-to-digital converter. Again, the range is predictable. But for most of the program parameters, particularly the internal ones, we can't be so certain what the range is going to be, so we must estimate it and provide a reasonable safety factor. This need is among the main reasons why embedded systems programming can be so tricky and why we get the big bucks.

ADDING AND SUBTRACTING

Two fixed-point numbers can be added only if their scales are the same. When you add two decimal numbers by hand, you've learned that you must write them down so their decimal points line up:

```
1,234.0
+  12.345
1,246.345                    (7)
```

We must do the same thing when we add fixed-point numbers in a computer. But since the computer doesn't know anything about scaling, we must line them up ourselves by shifting. We can't shift the larger number to the left (it would overflow), we can only shift the smaller number to the right. Suppose, as in the previous example, we have:

```
1,234.000 B11
+ 12.345  B4                 (8)
```

In hex, the numbers are:

```
  0x4d20 B11
+ 0x62c2 B4
```

Shifting the smaller number right gives:

```
    0x4d20 B11
  + 0x00c5 B11
    0x4de5 B11
```

This is equivalent to 1,246.3125, which is close enough.

When adding, you must also watch out for overflows. Consider the simple sum:

```
1 + 1:
0x4000 B1 + 0x4000 B1 = 0x8000 B1
                             (9)
```

Oops! In this case, the summation overflows. This possibility is something we must always expect and provide for when adding two numbers. Sometimes, you can predict in advance that a summation won't overflow. For example, if you're adding a small increment to a larger number that has already been properly scaled for the worst-case condition, you're safe. The biggest danger comes when the two numbers are approximately equal in size.

If this is the case and the numbers have been scaled close to their limits, you must provide one guard bit to catch the overflow. This is one area where assembly language is useful, because you can play with the carry bit to sense the overflow—that's what the bit is there for. C and C++ don't know about carry bits, so in these languages, you must make arrangements to be sure the overflow doesn't happen.

Let's consider some coding examples. To code the summation of Equation 8, we can write:

TABLE 2
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	0x764
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

```
int x = 0x4d20;             // scale B11
int y = 0x62c2;             // scale B4
int z = x + (y >> 7);       // scale B11
```

Always attach the scaling as a comment to every computation. It's the only way you'll be able to make sense of things later.

Where an overflow is possible, we have two choices: either shift both numbers right before adding or accumulate them in a `long` result. The first choice gives us the following code:

```
int x = 0x4000;             // scale B1
int y = 0x4000;             // scale B1
int z = (x >> 1) + (y >> 1); // scale B2
```

This choice avoids the use of a `long`, but we pay a price in precision. If the low-order bits of `x` and `y` are both ones, they would normally ripple into the next higher bit. By shifting them out before adding, we lose this bit. Using a `long` gives us more accuracy:

```
long temp = x + y;
int z = temp >> 1;          // scale B2
```

Again, if you're programming in assembly language, you can shift the carry bit back into the number and avoid the need for a `long temporary`.

Are you beginning to get the idea that even a simple addition is a lot of bother? Now you've got it! It also requires CPU clock cycles. This is the price we pay for wanting to do fast arithmetic in fixed point with as much accuracy as possible. Though the shifts and data movements take time, they still take a lot less time than floating point. As a matter of fact, the same shifts and data movements are done in floating point but are hidden from us.

You may think that subtraction doesn't suffer from the overflow problem, which is true if you can be assured that both numbers have the same sign. If they don't, however, you can get overflow in a subtraction by subtracting a negative number. So the same techniques must be used to handle the overflow.

The adjustment of the numbers before and after an operation is where the artistry in program design comes in. Although for best accuracy, we want to maintain the greatest number of bits in a number, a slavish devotion to this goal can cause the CPU to do more work. Sometimes it's better to relax the design requirements a hair, sacrificing a bit of accuracy in an intermediate result to cut down the number of shifts. Alternatively, you can sometimes prove via analysis that the overflow can't occur and skip the otherwise necessary shifting. All of this requires careful planning and attention to detail in every step of an algorithm. It also requires someone with the experience to know when the speed/accuracy trade is a good one and when it's not. That's why we get the big bucks.

MULTIPLYING

When multiplying two fixed-point numbers, we don't have to adjust them beforehand. You can multiply two

numbers of any scaling. You need only record what the scale of the result is. Even so, you'll find that shifting is almost always required. To see why, let's look at the simplest case I can think of, multiplying two numbers scaled B0. The largest number we can represent in this format is one, minus a bit:

0.9999695 = 0x7fff B0

Let's multiply two of these hex numbers together:

0x7fff * 0x7fff = 0x3fff0001

The first thing you'll notice is that we get a 32-bit result. That may not be a surprise to you or the CPU (the 80x86, like virtually every other computer ever built, is designed to accommodate that fact), but it is surprising to C and C++. If you need some more convincing, write yourself the following short C++ program:

```
void main(){
  int x = 0x7fff;            // scale B0
  int y = 0x7fff;            // scale B0
  cout << hex << x * y << endl;
}
```

What result did you get? Correct—the 0001 is the low-order half of the actual result. The high-order half, which is the part that contains most of the information, is quietly discarded with no error message. That's because the C and C++ rules of type conversion and promotion don't extend to understanding how multiplication (or addition, for that matter) can add bits to a result. According to their rules, any operation involving two `int`s gives an `int` result. Period. End of discussion.

Since that's not the kind of result we need, we must somehow coerce the compiler to promote the result to a long, at least temporarily. Unfortunately, in C, that's not possible without promoting the two operands as well. To fix our little test program, we must promote at least one of the operands to a `long` (if one is `long`, the other will be promoted automatically):

Anybody who says that C is an ideal language for programming real-time applications has obviously never had to do it.

```
void main(){
  int x = 0x7fff;
  int y = 0x7fff;
  cout << hex << (long)x * y << endl;
}
```

That change gives us the answer. But is it really correct? Actually, no, because we need the result back in a B0 integer. More importantly, even the value of the high word is wrong. Multiplying two values that are almost one, we should surely get a result that's also almost one. In other words, our final result should still be about 0x7fff. But as you can see, it's actually half that. The act of multiplication has shifted our result to the right one too many bits. Or, more accurately, it shifted it to the left 15 bits.

GETTING RESULTS

This is a very important result, particularly if you're writing your software in assembly language. As I mentioned earlier, virtually all CPUs are designed to multiply two 16-bit numbers and give a 32-bit result. In assembly language, it's a very simple matter to store the high half as the desired 16-bit result. Unfortunately, as you can see, this doesn't give us the result we need. We must either shift the result one bit to the left before storing the high half or shift the whole result 15 bits (not 16, which would be easier and quicker) to the right. Which solution you choose depends on the kind of hardware you're working with. In older or smaller computers, the time required to perform a shift depends on how many bit positions you're shifting, so a one-bit shift is much preferred. You can accomplish this if you create structures to hold the result:

```
struct doubleword{
  unsigned int lo;
  unsigned int hi;
};

union longword{
  long l;
  struct doubleword d;
};
```

With these structures in hand, the multiplication can be written:

```
longword temp = (long)x * y;
int z = temp.d.hi << 1;       // scale B0
```

The notation is certainly busy, though. We also have that nagging problem that the structure `doubleword` depends on the CPU rules for byte- and word-order (big endian or little endian). This makes our code nonportable.

Fortunately, most modern CPUs use barrel shifters, so the time required to perform a shift is independent of the number of bits. In the latter case, shifting right 15 bits is as good a solution as any. The corresponding code is:

```
int z = (int)(((long)x * y) >> 15);
```

Does it strike you that this is a torturous bit of code, just to multiply two numbers? It does me, too. Things get exponentially worse when we must cascade such operations through several steps.

This is my biggest beef with C and C++. Anybody who says that C is an ideal language for programming real-time applications has obviously never had to do it. Having to write obscure, write-only code like the line just shown for every multiplication operation in a program is a heck of a way to run a railroad or a software project.

The plain fact is this: We're trying to write fixed-point arithmetic in a lan-

guage (and on a CPU) that was designed to deal with numbers as pure integers.

We'll talk more about this and about languages that support true fixed-point arithmetic, next month. For now, recognize that when we're programming real-time systems using C and C++, we're really using the wrong tool for the job.

We can make the multiplications a bit more palatable by using the following macro:

```
#define mult(x, y, l, m, n) ((int)((long)x
  * y >> (15+l+m-n)))
```

Here, `x` and `y` are the two operands, `l` and `m` the integers representing their scaling, and `n` the desired scaling for the result. The product of `x` and `y` can then be written:

```
int z = mult(x, y, 0, 0, 0);
```

Using this technique, you won't be likely to write complex expressions, but then you're not likely to do that anyway, given the complexity of a single multiplication.

Once we've dealt with the horrors of multiplying even the simplest numbers, extending the process to less simple ones is a piece of cake. To multiply two numbers, add their scalings for the product. Thus, for example, the product of a number scaled B4 and one scaled B3 has a scale of B7. This assumes the one-bit adjustment that we must always do to retain precision. The following example will give you the picture:

```
  12.00 B4
* 6.00 B3
  72.00 B7
```

In hex, this is:

0x6000 * 0x6000 = 0x24000000

Shifting right 15 bits gives:

0x4800 = 72.00 B7

(To shift a number 15 bits on your calculator, divide by 0x8000, equalling 32,768.)

MORE TO COME

If you thought multiplication was fun, wait until you see division! Unfortunately, we're out of space for this month, so we'll have to postpone that pleasure until next time. We'll discuss division and the story of programming languages next month. We'll also wrap up the discussion with a practical example of programming in fixed point. See you then. ESP

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