Floating Point/Fixed-Point Numbers

Contents

Fixed-Point

Fixed point numbers are a simple and easy way to express fractional numbers, using a fixed number of bits. Systems without floating-point hardware support frequently use fixed-point numbers to represent fractional numbers. ("Systems without floating-point hardware support" includes a wide range of hardware—from high-end fixed-point DSPs, FPGAs, and expensive custom ASICs that process streaming media faster than any floating-point unit ever built; to extremely low-end microcontrollers).

The Binary Point

The term "Fixed-Point" refers to the position of the *binary point*. The *binary point* is analogous to the *decimal point* of a base-ten number, but since this is binary rather than decimal, a different term is used. In binary, bits can be either 0 or 1 and there is no separate symbol to designate where the binary point lies. However, we imagine, or assume, that the binary point resides at a fixed location between designated bits in the number. For instance, in a 32-bit number, we can assume that the binary point exists directly between bits 15 (15 because the first bit is numbered 0, not 1) and 16, giving 16 bits for the whole number part and 16 bits for the fractional part. Note that the most significant bit in the whole number field is generally designated as the sign bit leaving 15 bits for the whole number's magnitude.

Width and Precision

The width of a fixed-point number is the total number of bits assigned for storage for the fixed-point number. If we are storing the whole part and the fractional part in different storage locations, the width would be the total amount of storage for the number. The **range** of a fixed-point number is the difference between the minimum number possible, and the maximum number possible. The precision of a fixed-point number is the total number of bits for the fractional part of the number. Because we can define where we want the fixed binary point to be located, the precision can be any number up to and including the width of the number. Note however, that the more precision we have, the less total range we have.

There are a number of standards, but in this book we will use **n** for the width of a fixed-point number, **p** for the precision, and **R** for the total range.

Examples

Not all numbers can be represented exactly by a fixed-point number, and so the closest approximation is used.

The formula for calculating the integer representation (X) in a Qm.n format of a float number (x) is:

$X = \text{round}(x*2^n)$

To convert it back the following formula is used:

$x = X * 2^{-n}$

Some examples in Q3.4 format:

```
After conversion:                         After converting them back:
0100.0110 = 4 + 3/8                         4 + 3/8
0001.0000 = 1                             1
0000.1000 =  1/2                           1/2
0000.0101 =  5/16                           0.3125
0000.0100 =  1/4                           1/4
0000.0010 =  1/8                           1/8
0000.0001 =  1/16                           1/16
0000.0000 = 0                             0
1111.1111 = -1/16                           -1/16
1111.0000 = -1                            -1
1100.0110 = -4 + 3/8 = -( 3 + 5/8 )           -4 + 3/8
```

Randomly chosen floats:

```
0000.1011 = 0.673                    0.6875
0110.0100 = 6.234                    6.25
```

Some examples in the (extremely common)[1] Q7.8 format:

```
0000_0001.0000_0000 = +1
1000_0001.0000_0000 = -127
0000_0000.0100_0000 = 1/4
```

Arithmetic

Because the position of the binary point is entirely conceptual, the logic for adding and subtracting fixed-point numbers is identical to the logic required for adding and subtracting integers. Thus, when adding one half plus one half in Q3.4 format, we would expect to see:

```
 0000.1000
+0000.1000
_________
=0001.0000
```

Which is equal to one as we would expect. This applies equally to subtraction. In other words, when we add or subtract fixed-point numbers, the binary point in the sum (or difference) will be located in exactly the same place as in the two numbers upon which we are operating.

When multiplying two 8-bit fixed-point numbers we will need 16 bits to hold the product. Clearly, since there are a different number of bits in the result as compared to the inputs, the binary point should be expected to move. However, it works exactly the same way in binary as it does in decimal.

When we multiply two numbers in decimal, the location of the decimal point is N digits to the left of the product's rightmost digit, where N is sum of the number of digits located to the right side of the decimal point in the multiplier and the multiplicand. Thus, in decimal when we multiply 0.2 times 0.02, we get:

```
 0.2
x0.02
_____
0.004
```

The multiplier has one digit to the right of the decimal point, and the multiplicand has two digits to the right of the decimal point. Thus, the product has three digits to the right of the decimal point (which is to say, the decimal point is located three digits to the left).

It works the same in binary.

From the addition example above, we know that the number one half in Q3.4 format is equal to 0x8 in hexadecimal. Since 0x8 times 0x8 in hex is 0x0040 (also in hex), the fixed-point result can also be expected to be 0x0040 - as long as we know where the binary point is located. Let's write the product out in binary:

```
0000000001000000
```

Since both the multiplier and multiplicand have four bits to the right of the binary point, the location of the binary point in the product is eight bits to the left. Thus, our answer is 00000000.01000000, which is, as we would expect, equal to one quarter.

If we want the format of the output to be the same as the format of the input, we must restrict the range of the inputs to prevent overflow. To convert from Q7.8 back to Q3.4 is a simple matter of shifting the product right by 4 bits.

FIR filter

Fixed-point numbers are often used internally in digital filters including FIR and IIR filters.

There are a number of practical considerations for implementing FIR and IIR algorithms using fixed-point numbers.[2][3]

Sine table

Many embedded systems that produce sine waves, such as DTMF generators, store a "sine table" in program memory. (It's used for approximating the mathematical sine() and cosine() functions). Since such systems often have very limited amounts of program memory, often fixed-point numbers are used two different ways when such tables are used: the values stored in the tables, and the "brads" used to index into these tables.

Values stored in sine table

Typically one quadrant of the sine and cosine functions are stored in that table. Typically it is a quadrant where those functions produce output values in the range of 0 to +1. The values in such tables are usually stored as fixed point numbers—often 16-bit numbers in unsigned Q0.16 format or 8-bit numbers in unsigned Q0.8 values. There seems to be two popular ways to handle the fact that Q0.16 can't exactly handle 1.0, it only handles numbers from 0 to (1.0-2^-16): (a) Scale by exactly a power of two (in this case 2^16), like most other fixed-point systems, and replace (clip) values too large to store as that largest value that can be stored:

so 0 is represented as 0, 0.5 represented as 0x8000, (1.0-2^-16) represented as 0xFFFF, and 1.0 truncated and also represented as 0xFFFF.[4] (b) Scale by the largest possible value (in this case 0xFFFF), so both the maximum and minimum values can be represented exactly: so 0 is represented as 0, (1.0-2^-16) represented as 0xFFFE, and 1.0 is represented as exactly 0xFFFF.[5]

A few people draw fairly accurate circles and calculate fairly accurate sine and cosine with a Bezier spline. The "table" becomes 8 values representing a single Bezier curve approximating 1/8 of a circle to an accuracy of about 4 parts per million, or 1/4 of a circle to an accuracy of about 1 part in a thousand.[6][7]

Turns

Many people prefer to represent rotation (such as angles) in terms of "turns". The integer part of the "turns" tells how many whole revolutions have happened. The fractional part of the "turns", when multiplied by 360 (or $1\tau = 2\pi$[8]) using standard signed fixed-point arithmetic, gives a valid angle in the range of -180 degrees ($-\pi$ radians) to +180 degrees ($+\pi$ radians). In some cases, it is convenient to use unsigned multiplication (rather than signed multiplication) on a binary angle, which gives the correct angle in the range of 0 to +360 degrees ($+2\pi$ radians).

The main advantage to storing angles as a fixed-point fraction of a turn is speed. Combining some "current position" angle with some positive or negative "incremental angle" to get the "new position" is very fast, even on slow 8-bit microcontrollers: it takes a single "integer addition", ignoring overflow. Other formats for storing angles require the same addition, plus special cases to handle the edge cases of overflowing 360 degrees or underflowing 0 degrees.

Compared to storing angles in a binary angle format, storing angles in any other format—such as 360 degrees to give one complete revolution, or 2π radians to give one complete revolution—inevitably results in some bit patterns giving "angles" outside that range, requiring extra steps to range-reduce the value to the desired range, or results in some bit patterns that are not valid angles at all (NaN), or both.

Using a binary angle format in units of "turns" allows us to quickly (using shift-and-mask, avoiding multiply) separate the bits into:

- bits that represent integer turns (ignored when looking up the sine of the angle; some systems never bother storing these bits in the first place)
- 2 bits that represent the quadrant
- bits that are directly used to index into the lookup table
- low-order bits less than one "step" into the index table (phase accumulator bits, ignored when looking up the sine of the angle without interpolation)

[9][10][11]

The low-order phase bits give improved frequency resolution, even without interpolation.

Some systems use the low-order bits to linearly interpolate between values in the table.[12] This allows you to get more accuracy using a smaller table (saving program space), by sacrificing a few cycles on this "extra" interpolation calculation. A few systems get even more accuracy using an even smaller table by sacrificing a few more cycles to use those low-order bits to calculate cubic interpolation.[4]

Perhaps the most common binary angle format is "brads".

Brads

Many embedded systems store the angle, the fractional part of the "turns", in a single byte binary angle format.[13] There are several ways of interpreting the value in that byte, all of which mean (more or less) the same angle:

- an angle in units of brads (binary radians) stored as an 8 bit unsigned integer, from 0 to 255 brads
- an angle in units of brads stored as an 8 bit signed integer, from -128 to +127 brads
- an angle in units of "turns", stored as a fractional turn in unsigned Q0.8 format, from 0 to just under 1 full turn
- an angle in units of "turns", stored as a fractional turn in signed Q0.7 (?) format, from -1/2 to just under +1/2 full turn

One full turn[14] is 256 brads[15] is 360 degrees.

If a single byte doesn't give enough precision, the brad system can easily be extended with more fractional bits—65,536 counts per turn can be represented in 16 bits.[16]

For further reading

1. Bruce R. Land. "Fixed Point mathematical functions in GCC and assembler" (https://courses.cit.cornell.edu/ee476/Math/).
2. Randy Yates. "Practical Considerations in Fixed-Point FIR Filter Implementations" (http://www.digitalsignallabs.com/fir.pdf). 2010.
3. Randy Yates. "Fixed-Point Arithmetic: An Introduction" (http://www.digitalsignallabs.com/fp.pdf). 2013.
4. Brad Eckert: approximating the sine with 16 short, equally-spaced-in-x cubic splines per quarter-cycle "gives better than 16-bit precision" for sin(x). [1] (http://web.archive.org/web/20060714225905/http://tinyboot.com/cubic.txt)
5. "Renormalizing the Sine Table" (http://forums.parallax.com/archive/index.php/t-116047.html)
6. Don Lancaster. "Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines." (http://www.tinaja.com/cubic01.asp) "Tutorial derives the approximation math for various spline fits from two through eight."
7. Wikipedia: Bezier spline#Approximating circular arcs
8. Michael Hartl. The Tau Manifesto: π is wrong. (http://tauday.com/) 2010.
9. Sound Synthesis Theory/Oscillators and Wavetables
10. description of a sine wave generator using a wave table (http://electronics.stackexchange.com/questions/16512/is-this-a-suitable-sine-wave-osc-how-would-i-control-the-frequency/16516#16516)
11. Eric Smith. "Implementing the Sine function on the PIC". [2] (http://www.brouhaha.com/~eric/pic/sine.html) (no interpolation -- fast but rough)
12. Scott Dattalo. "Sine waves with a PIC". [3] (http://www.dattalo.com/technical/software/pic/picsine.html) (uses interpolation -- much more accurate but slower)
13. Futurebasic/Language/Reference/circle
14. Wikipedia: turn (geometry)
15. Wikipedia: binary radian
16. Garth Wilson. "Large Look-up Tables for Hyperfast, Accurate, 16-Bit Fixed-Point/Scaled-Integer Math: Angles in trigonometry". [4] (http://wilsonminesco.com/16bitMathTables/index.html#TRIG_ANG). 2012.

- Embedded Systems/Floating Point Unit
- Ada Programming/Types/delta describes the excellent support for fixed-point numbers in the Ada programming language.
- The Philosophy of Fixed Point (http://forth.com/starting-forth/sf5/sf5.html) (from Starting FORTH by Leo Brodie)