Use fixed-point math for embedded applications

Steve Hageman - June 04, 2013

Why is my floating point math so inaccurate?
If you lurk on embedded C forums for much time at all you will run into a question like this: "Why is my floating point math so inaccurate? I do a few operations on the number 10 and I end up with 9.99989 as the result?"

One of the other common forum complaints, especially for the small memory Microprocessors is: "Why does my Floating Point take so much room?" Just adding a single floating point multiply will necessitate the inclusion of at least the basic floating-point math routines, which on a small device can consume a lot of memory. Add a single printf and even more floating-point conversion support needs to be included.

These floating-point operations can take a lot of time also, but interestingly enough that seems to get rarely mentioned on the forums. That would suggest that our microprocessors are fast enough for most human interaction operations anyway, so people don't have as much reason to notice this.

To those of us who grew up on something similar to Apple ][ Basic this is no surprise – we are well familiar with the effects of limited precision floating point. Those who grew up on PCs, with their double precision math and nearly unlimited resources, have never experienced this problem.

The solution to most of these problems is to use fixed-point math instead of floating point [1].

This would seem to be a big constraint; but if you are clever enough anything can be represented in fixed-point math. My most basic C compiler for microprocessors has support for up to 32-bit integers. Using an unsigned int32 allows values of between 0 and 4,294,967,296 - that's 192 dB of dynamic range and should be plenty for nearly any application.

Some Implementation Examples
One thing I always like to do when possible is to use 12-bit data converters, as that level of resolution works out well for many, many analog control tasks. Even with no oversampling and processing gain applied, a 12-bit data converter yields 72 dB dynamic range. This is way more than most human interaction applications require.

Why 12 bits? The cost is right and if you use a 4.096-volt reference you get a bit scaling of 1 millivolt per bit! How convenient is that? 1 mV is just naturally easy to remember - full scale is then 4095 counts and 4.095 volts you can easily convert to a voltage by placing a decimal point by yourself. This in fact is how fixed point begins – the proper selection of your analog reference and data converter (or just your data) to get a basic unit of your measurement to fit nicely into an integer value.
Sometimes the implementation is a little more difficult. In a VHF receiver that I designed [2], the tuning was from 144 to 148 MHz in 5-kHz steps. This design used a simple low-cost PLL with a reference frequency or channel spacing of 5 kHz. This works out to be 800 channels, and internally the microprocessor kept track of the frequencies as a channel represented as an unsigned int16. So by simple math I could program the PLL by starting at 0 and then just incrementing each tuning frequency by one channel or count of the PLL feedback divider.

Naturally users don't want to see channel numbers, they want a readout in megahertz with decimals representing the kilohertz tuning portion – like: "144.525 MHz."

With the simple brute force C code segment below I was able to display on the design's internal LCD the frequency the way that the user wanted to see it.

```c
// Calculate actual frequency for the LCD display
uint16 freq;
freq = Channel*5 + 4000;

// Write frequency to LCD display
write_lcd("14");
write_lcd((uint8)(freq / 1000) + '0');
write_lcd('.
write_lcd((uint8)((freq / 100) % 10) + '0');
write_lcd((uint8)((freq / 10) % 10) + '0');
write_lcd((uint8)(freq % 10) + '0');
write_lcd(" MHz");
```

As you can see – in this design I didn't even have to keep track of the "14" portion of the MHz values as that never changed, it is hard coded. Only the MHz and kHz values changed. No floating point overhead and no math round-off errors occur in this implementation. This is low memory usage, fast and sweet.

With higher frequencies, like the 800-MHz cell phone range, you will probably need to use a dual modulus PLL or perhaps even a fractional N type. Even though the programming algorithm is more complex with these types of PLLs the same basic principles apply.

I always sit down with a notepad or spreadsheet and make a programming table with the frequencies required as an input and the programming bits required as an output.

There is always a repeating pattern that can be found; then this pattern can be implemented in fixed-point code because the programming bits to the IC are always integers themselves at their most basic level.

**Sensing dBs and Correcting the Results**

An 800-MHz radio receiver design I worked on needed to measure the received channel power over a 90-dB range. For this I used an Analog Devices AD8310 demodulating log amplifier [3]. The basic receiver design used a single-conversion architecture where the filtered 45-MHz IF was directly detected by the AD8310. The DC output of this amplifier was then digitized by my microprocessor's 10-bit internal ADC.
The AD8310 has an output slope (or gain) of 24 mV/db; by using a 2.5-volt reference on the microprocessor's ADC the LSB value was: 2.44 mV / LSB. That is blissfully close to the 24-mV/db slope of the AD8310!

Uncorrected that scale factor results in 9.8 LSB / db. That means we would get nearly 10 LSB counts per dB input change. Since the final result needed was only to the nearest db we would place our decimal one digit to the left in our ADC result to get a fixed-point value in dB.

The code for this conversion is as follows:

```c
uint16 adc_result;
uint8 db_value;

adc_result = read_adc();
db_value = (uint8)(adc_result / 10);

return(db_value);
```

If we just used that conversion the result would be only a 1.5-dB error over a 90-dB range.

If we offset the conversion by 0.8 dB (or 8 ADC counts) in the middle of the receiver's range we could halve that value to be less than +/-0.8 dB. Figure 1 shows the unadjusted and corrected error of this conversion scheme.

![Plot: Error](image)

Figure 1: Using a 2.5 volt reference nearly matched the output of the AD8310 Log Amp, but there is still a 10.8% gain error that adds up to a total error of slightly over 1.5 dB over a 90 dB range.
The receiver specifications called out for a 1-dB typical (+1 digit) error, and since the AD8310 has a linearity error of +/-0.4 dB we should keep the math conversion errors to below 0.5 dB to meet the specifications goal.

To get the error even lower, a single resistor can be added to the AD8310 to tweak the gain up by 10.8% to make the AD8310 output match the scale factor of the ADC. That results in a zero gain error, and the conversion error if shown on Figure 1 would be a flat line.

A purely code way to make the correction is to multiply the dB calculated value by 56/55. This corrects the slope error to be exactly zero as shown in Figure 2.

![Figure 2: By adding a compensating slope the error may be made to be zero over the full 90- dB range. This plot adds the quantization error of the ADC (The jagged lines).](image)

Code wise this error correction looks like the following:

```c
uint16 adc_result;
uint8 db_value;

adc_result = read_adc();
db_value = (uint8)(((adc_result * 56) / 55) / 10);
return(db_value);
```

You need to be careful that the intermediate result does not overflow the unsigned `int 16` variable size and you should make sure that your compiler is casting the way you think it is - i.e., that the intermediate result is not getting cast to an `int8` and overflowing or something silly like that.
Another more brute force way to reduce the error is to have a multi-part offset using if-then-else statements. Code wise, this conversion function looks like this:

```c
uint16 adc_result;
uint8 db_value;

adc_result = read_adc();

if(adc_result > 255)
    db_value = (uint8)((adc_result + 12) / 10);
else if(adc_result > 511)
    db_value = (uint8)((adc_result + 16) / 10);
else if(adc_result > 767)
    db_value = (uint8)((adc_result + 21) / 10);
else
    db_value = (uint8)((adc_result +8) / 10);

return(db_value);
```

Figure 3 shows the results of the multi-part offset correction. This method adds very little code space overhead and execution time as the if-then-else construct is usually quite compact and efficient and it can keep the error in our fixed point correction to reasonable levels.

![Figure 3: The more brute force multi-part error correction simply breaks the total range up](image-url)
and adds a compensating offset to keep the total error in check. This is more than adequate for most applications and is usually very computationally and space efficient.

**Speed and Size**

**Speed and Size**

Speed and size mean a lot in small embedded systems. Memory is still at a premium and, frankly, many of the compilers we have choke when the code space fills up, as they don't have the ability to really pack every last bit into the available memory.

To find out the speed and size comparison of fixed-point versus floating-point math, I made a simple test program for a Microchip PIC18F6722 with two test cases: One was an integer math problem, the other was floating point. The math itself was a simple linear curve fit of the \( Y = m \times X + b \) variety. I also added one `printf` statement that printed out the integer or floating point result to simulate a human interaction. After all, the results need to be displayed somewhere or why do them at all?

Figure 4 shows the results of that test. The floating point is 33 times slower and took nearly 1800 more bytes of program memory to perform the same task as the fixed-point case.

<table>
<thead>
<tr>
<th>Case</th>
<th>Code Size Delta (bytes)</th>
<th>Execution Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Point</td>
<td>0</td>
<td>1.2 ( \mu )S</td>
</tr>
<tr>
<td>Floating Point</td>
<td>1780 (+6 bytes RAM)</td>
<td>39 ( \mu )S</td>
</tr>
</tbody>
</table>

**Figure 4:** The results of a comparison of a fixed point and floating point math operation of the type \( Y = m \times X + b \) and a single `printf` of the proper type to a small embedded processor. The floating point adds a lot of code space consumption and executes nearly 33 times slower than the fixed-point equivalent.

**The Bottom Line**

To some this seems like a lot of work when a PC can dispense with all of this, and I agree, these methods would probably never be used in a PC-based application, but this is an extremely useful concept when we step into the world of the small and resource-limited single-chip microprocessors, and in many instances it is the only way to avoid the dreaded limited precision floating-point problems that result in such environments.

**References**

