Round off accuracy problem
Wed, 8 Oct 2003
scott_bayes@agilent.com
"VEE vrf" vrf@it.lists.it.agilent.com

Don't feel silly Mark. Most everyone gets caught by this and analogous problems, because they're pretty subtle. I've written up the following explanation in various forms a few dozen times over my years at HP and Agilent, so I'll do it once more here, in the context of your example.

You may actually have 2 problems here, depending on the numbers you're really wanting to work with:

1. Real32 can only represent about 7 to 8 "significant digits" of a number. That's independent of whether the number is .1234567 or 123456.7 (i.e. the decade is unimportant, the dynamic range is pretty large). Your 26009.6 is right at 6 sig digits, perilously close to the edge. If your numbers typically have this much "significance" (6 digits or more) you may want to use real64 if that's possible. They can represent about 16 digits.

2. VEE works in binary (base 2), not decimal (base 10). This is at the heart of many problems like this, where we have unexpected "round-off errors".

Let's start with something we learned in school: 1/2 is exactly 0.5, 1/4 is exactly 0.25, etc. And 1/5 is exactly 0.2, 1/25 is exactly 0.04, etc. Finally 1/10 is exactly 0.01, 1/100 is exactly 0.01, etc.

But we also learned that 1/3 is 0.333... repeating, not exact; 1/7 is 0.142857142857... repeating, not exact; and 1/21 is 0.047619047619... repeating, not exact.

Why is this?

It's because 10 is composed of the factors 2 and 5 (2x5=10), and all the exact divisors above have only 2s and 5s in their factors(*). We use base 10, so without going through a proof, I think you'll see it makes intuitive sense that these fractions would be exact in base 10. And of course all the INexact divisors have at least one factor that is NOT a 2 or a 5, e.g. 3 or 7.

The other thing we learned is that if you multiply (1/5)*5 in decimal you'll get 1.0 exact, but if you multiply (1/3)*3, you'll get 0.999... repeating, inexact, and you have to remember that 0.999... is in some strange way equivalent to 1.0. Those inexact numbers always cause problems and special cases!

But computers don't generally do decimal math, they do binary math instead. Yes, some can do decimal math, though at a big performance penalty and typically only for financial applications (bean counters hate seeing $0.00999... for 1 penny!), but these days most all our scientific/engineering math is done in binary because it's fast and efficient. VEE is no exception.

But look out! Whereas base 10 has factors of 2 and 5, base 2 does not have a factor of 5, it's only got a 2. This means that any divisor that has factors other than 2 is inexact in base 2 and has to be approximated. 1/2 is OK, 1/4 is ok, etc. But 1/5 is inexact, repeating (I believe it's .00110011... repeating, if I did my binary division correctly by hand)

In your case, that 26009.6 is really 26009 + 6/10. 6/10 reduces to 3/5, and there's our factor of 5, leading to inexactness! So in binary, 26009.6 is not exact, it's a repeating binary fraction. Bummer!

Ultimately VEE has no way to keep an exact representation of 26009.6(**); it's forced to keep an approximation, because it doesn't have an infinite number of bits available. In other words, the round-off problem is built-in mathematically, just as in base 10 we know that 26009 + 1/3 can't be represented exactly.

By the way, this is all controlled in most computers by an IEEE standard, IEEE-754-1985 (don't know if it's been updated since 1985, I have the 1985 version), which is pretty well universally used. When properly implemented, this spec guarantees that the approximations generated are the closest possible in base 2, so it's a pretty good spec. By the way, this is pretty well equivalent to the way in which 0.999... is somehow equal to 1.0: there is no number larger than 0.999... that is closer to 1.0; 1.0 is the only thing closer! 0.999... is the best approximation available in decimal math.

But as you can see, C or C++ or C#--or even FORTRAN, the king of number-crunchers--will do no better than VEE with these fractional numbers. Misery loves company.

So we're stuck with the problem in this case. What do we do about it?

- for display or conversion to text, figure out how many significant digits you want to show the user, and set up the format accordingly. For example, set up your topmost To String's WRITE TEXT transaction to:

WRITE TEXT a REAL32 FIX:0 FW:7 RJ EOL

(you can use variations on this, like using default field width instead of FW:7 for example)

and you'll see the 26010 that you expect. For the bottom one, you can't fix the READ (it's forced to create an approximation), but in the Alphanumeric display, in Properties under the Number tab, turn off Global format, set Real:Fixed and ask for 1 Fractional digit. Do similar things with To Files, etc.

Note that you can't just do this blindly; you need to do a little analysis, try some examples, and make sure it always does what you expect. E.g. if your Real32 had 9 significant digits, maybe 26010.1234, the above WRITE would strip off the .1234 (and the Real32 wouldn't be able to keep track of all 9 digits very accurately anyway, losing the final "4", whereas a Real64 would)

- for comparison to another number, use VEE's ~= (approximately equal) operator. In Functiona & Object Browser choose Operators>Comparison, and there you'll see it. It's not perfect, but it will tell you that A(A/B)*B) ~= 1.0 where A= 1.0 and B=3.0, is true in a Formula box (doing it as ((1.0/3.0)*3.0)==1 without input pins returns true for some reason I don't yet understand), which is what we want. Read the F1 help for ~= before you start using it, because 0.0 is a special case for example.

VEE wins here! Many other languages don't have an ~= operator available, and it's a pain to build it for yourself, trust me.

Mark, I've attached a copy of your example program, so it stays together with this article.

NOTES

* Actually, decimal numbers will be exact as long as they contains only SUMS of fractions with only 2s and 5s in their divisor factors, e.g. 1/2 + 1/5 = 7/10 = 0.7, which is exact.

** There are languages that do "symbolic math", and they can keep track of things like (1/3)*3 and return 1.0 exact, but they do so at quite a performance penalty, not suitable for VEE.

I see that my old buddy Steve Asprelli has suggested using either Real64 or scaled integer math, both good suggestions. Note that even in Real64 your numbers won't be exact, but the approximations will be a lot closer and your calculations more trustworthy. I'd start with that for sure.

The technique of using scaled integers is also cool (also called fixed-point math, in contradistinction to floating-point math which is what Real32 and Real64 are), but you need to be sure your app is

amenible to the technique. The killer problem is often dynamic range, e.g. where adding large scaled numbers can overflow the integer. And multiply, divide and subtract can be, well, "interesting". You need to analyze your problem before you decide to use fixed-point. With floating-point the dynamic range of the numbers is so huge that this analysis is almost never needed (but you have the pesky inexact fractions to deal with instead, and you have to choose between Real32 and Real64 and know the limits if your calculations need to be very exact, e.g. matrix math).

(Shawn, I'm not trying to take over the "longest vrf essay" title. Honest. :-)

Best Regards,

Scott Bayes
Software Technical Support

Agilent Technologies, Inc.
815 14th Street S.W.
Loveland, CO, U.S.A. 80537

970 679 3799 Tel
970 635 6867 Fax