USING IEEE FLOATS IN OPTO 22 PRODUCTS

About Floats

While computers, CPUs, and electronic devices (such as Opto 22 controllers) store numbers in binary format, most often they represent real numbers as floating point numbers, or *floats*. For example, in industrial automation applications, all analog values read from an I/O unit are floats. Floats represent real numbers in scientific notation: as a base number and an exponent.

The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the most widely used standard for floating-point computation. It defines how to store real numbers in binary format and how to convert between binary and float notations.

Opto 22's *groov* EPIC, *groov* RIO, and SNAP PAC products use IEEE single-precision floats, which have 32 binary digits (bits). The IEEE 754 32-bit float format is as follows:

1 bit	8 bits	23 bits
x	xxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxx
Sign	Exponent	Significand

Float calculation: $(-1)^{\text{Sign}} \times [1 + \text{Significand}/2^{23}] \times 2^{(\text{Exponent}-127)}$

While this is an excellent standard for the purpose, it has limitations that could cause issues if you're not aware of them. Squeezing infinitely many real numbers into a finite number of bits requires an approximate representation. Most floats cannot be exactly represented using this fixed number of bits in a 32-bit IEEE float. As a result, rounding error is inherent in floating-point computation.

In software (for example *groov* Manage, PAC Control, PAC Manager, and the OptoMMP protocol), a float is a 32-bit IEEE single-precision number ranging from $\pm 3.402824 \times 10^{-38}$ to $\pm 3.402824 \times 10^{+38}$. These single-precision floats give rounding errors of less than one part per million (1 PPM). You can determine the limit of the rounding error for a particular float value by dividing the value by 1,000,000.

This format guarantees about six and a half significant digits. Therefore, mathematical actions involving floats with seven or more significant digits may incur errors after the sixth significant digit. For example, if the integer 555444333 is converted to a float, the conversion yields $5.554444e^{+8}$ (note the error in the 7th digit). Also, converting $5.554444e^{+8}$ back to an integer yields 555444352 (note the error starting in the 7th digit).

Float Issues and Examples

Accumulation of Relatively Small Floating-point Values

When adding float values, the *relative size* of the two values is important. For example, if you add 1.0 to a float variable repeatedly, the value of the float variable will correctly increase in increments of 1.0 until it reaches $1.677722e^{+7}$ (16,777,220).

At that point you lose precision. The value will no longer change, because 1.0 is too small relative to $1.677722e^{+7}$ to make a difference in the significant digits. The same thing occurs if you add 0.0001 to 2,048.0, or add 1,000.0 to $1.717987e^{+10}$. The key is the relative size of the numbers.

Here's another way to think of it. Suppose your bank could keep track only of seven digits. If you were fortunate enough to have one million dollars ($1,000,000) in your account and tried to add 10 cents ($0.10) to it, you would not be able to, because the 10 cents is not big enough relative to the total to be significant. Since in this example the bank has only seven digits to keep track of your money, one digit has to fall off the end:

© 2022 Opto 22. All rights reserved. Dimensions and specifications are subject to change. Brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

either the 10 cents falls off the right side or the million digit falls off the left side. Which would you rather see in your bank account?

Seven digits available:	1	2	3	4	5	6	7	
Amount in account:	1	0	0	0	0	0	0	
Add 10 cents (0.10)? (digit falls off on right)	1	0	0	0	0	0	0	
Add 10 cents (0.10)? (digit falls off on left)	0	0	0	0	0	0.	1	Oops!

Note that moving the point indicator doesn't help, because the exponent is separate. If the seven digits for the account represent millions of dollars (1.000000) rather than dollars (1,000,000), the 10 cents would be 0.0000001—still too small to be represented by the seven digits:

Seven digits available:	1	2	3	4	5	6	7	
Amount in account	1.	0	0	0	0	0	0	
Add 10 cents (0.0000001)? (digit falls off on right)	1.	0	0	0	0	0	0	
Add 10 cents (0.0000001)? (digit falls off on left)	.0	0	0	0	0	0	1	Oops again!

The key is that it is not the size of the numbers that matter, but rather their *relative* size.

So if you are accumulating relatively small values in a float variable over a long period of time, at some point, the float value will stop increasing even though you continue to try to add to it.

Comparing Floating-point Values for Equality

Due to rounding errors and the way floating-point calculations are performed, comparing two floats for equality can yield inaccurate results. The precision of comparisons depends on the relative size of the float values as compared to the difference between them.

For example, if 2,097,15**1**.0 is compared for equality with 2,097,15**2**.0, the result will indicate that the two floats are equal, even though it's obvious they are not. The reason is that the difference between the two values is 1.0, and 1.0 compared to one of the compared values (2,097,151.0) is too small; it is less than one part per million.

In this case, 2,097,152.0 divided by 1,000,000 is 2.1. If the difference between the two values is at least 2.1, then the equality comparison is guaranteed to be correct. So if 2,097,152.0 and 2,097,149.0 were compared for equality, the result will indicate they are not equal, because the difference (3.0) is greater than one part per million (2.1). Any time the difference is *at least* one part per million, the result is guaranteed to be accurate. If the difference is less than 1 PPM, it may or may not be accurate.

One method that programmers use to work around this issue is to subtract one float from the other and then compare the absolute value of the result to a limit.

For example:

```
Float_Diff = Float2 - Float1;
If (AbsoluteValue(Float_Diff) < 1.0 ) then
  SetVariableTrue(EqualityFlag);
Else
  SetVariableFalse(EqualityFlag);
Endif
```

Helpful Links for More Information

From Wikipedia, the free encyclopedia:

- IEEE 754: https://en.wikipedia.org/wiki/IEEE_754
- Real numbers: https://en.wikipedia.org/wiki/Real_number
- Good example: https://en.wikipedia.org/wiki/Single_Precision

MADE IN THE USA

© 2022 Opto 22. All rights reserved. Dimensions and specifications are subject to change. Brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

Rounding error: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Comparing floating point numbers:
http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm

© 2022 Opto 22. All rights reserved. Dimensions and specifications are subject to change. Brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.