| **Index** | **Comment** | **Evan’s Response** | **Jim’s Response** | **For Clint** |
| --- | --- | --- | --- | --- |
| 5 | For SingleToString, we need some kind of tolerance, or SingleToString needs to be updated to round like the rest of the world.  Examples:  Input string:  .005  PIC output:    0.00499999    Input string:  .249  Pic output:    0.24899999 | See #7 also.    Recommend.  Only send decimal values from the PC to the PIC that are based on a HEX number.    So,    0.005 is encoded as 0x 3ba3d70a  so, send from PC 0.00499999988824129 and expect 0x 3ba3d70a and same decimal number    Then match.      0.249 is encoded as 0x3ba3d70a so, send from PC 0.248999997973442 expect back the same  **It is ok to disagree. I am learning more each day. Mostly from you!**  **See this page. They are fighting the same issue. https://www.arduino.cc/reference/en/language/variables/data-types/float/**  **Our issue to be assured the functions we have provide the correct results with the constraints of the testing methods.**  **We need to pragmatic and give a range of what was tested.**  **So, here is an idea. Test from 1000000.0001526 to -1000000.0001526 in increment of 0.000152587890625**  **So, you will never test 0.005 but 0.005035400390625 and 0.005035400390625.** | I respectfully disagree on this one.  The whole purpose is to see how the system handles numbers that the normal user would use.  He's going to put in .005, not 0.004999999888, and even if he did, SingleToString should present the output to the 7 significant digits possible which would be .005, just like every version of basic I've ever used.  If you want to see extra digits, that's what formated print is for. | No response needed |
| 7 | Or, round to 3 decimal figures and compare | This would test more numbers.  Therefore, send next number to PIC and then round PC and PIC to 3 decs.  **See #6.** | This is type of thing I am stumbling on.  It would have to 3 significant digits, not 3 decimal places.  Or was that what you meant by "3 decimal figures". | Clint response please. |
| 9 | **StringToSingle**    1. trailing blanks       eg.  "1.2     " yields 1.28471994 | What is adding the trailing blanks? What is yielding 1.28… ?    **The users will have to ensure the string is trimmed. Why ? the trim routines uses a lot of RAM and I think we should further burden these two routines with TRIM() as the default.**  **We could ask Clint to add TRIM() as a conditional compiler option then the default would be no trim and with the conditional compile the TRIM() could be included.**  **Your thougths?** | Me.  I put those blanks there for testing.  It should be legal.  (or if not, don't return bogus numbers.) StringToSingle is yielding 1.28471994 when I send it "1.2   ".  However, I have yet to check that error byte Evan was talking about.   Perhaps it was set, and the return was just random garbage in memory, don't know which. | No response needed |
| 11 | **StringToSingle**    Does not take exponential notation e.g.  1.2e4 | This is rangeconstraint.    No exponents are supported.    Recommendation    This limits the range of tests to HEX numbers.  **OK** | Appologies for the blanket statement Clint.  My bad.  Very bad.  It totally accepts exponential notation.  The error I was getting was from SingleToString, not StringToSingle.  For "4e-008",  StringToSingle produced the correct value and hex.  It was the SingleToString that returned the string "Error".  However, for the string  "4.0e-008" StringToSingle produced the number 4.70630073, so not good.    I definately need to do more testing to pin down what works and what doesn't. | No response needed |
| 12 | **StringToSingle**      Numbers between and including -.00000005 to .00000005 return "Error " | Is this range constraint.?    We should check with Clint.      However, now that you raise this… then you may not have scanned the Help I wrote based on Clint’s code.    I should have thought about this.    May be you can returned the Error byte. It could have useful information.    The user can check for ‘Error’ and the can check the returned Error byte `SysByte\_STS\_Err`    This is the help page.  <https://gcbasic.sourceforge.io/help/_stringtosingle.html>    **OK** | Clint- my appologies again, it was SingleToString that was producing the error.  I will have to do more testing, but I think StringToSingle is good in this area.  Way too many trees in this forest. | Clint response please. |
| 14 | **SIngleToString**    Routine does not round properly.    This only appears to be a problem for numbers between -.25 and .25 all other numbers seem to work properly. | Is this range constraint.?    We should check with Clint.    **See #6. The rounding error, for our testing could be resolved.** | Again, I wasn't precise in my statement.  starting at -.249, the next few errors were at -.242,-.235,-.227,-.229,-.213  Later on there were areas with 1 out 3 errors, later still the error rate got less.  No obvious overall pattern I could see.  And again, these weren't flat out errors, it was only the rounding in question. | Clint response please. |

https://forum.arduino.cc/t/a-basic-question-about-float-and-double/580350/3OP

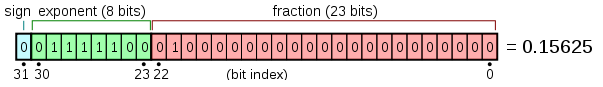
**1.** In human vocabulary, the **integer number** refers to a number with no decimal point and fractional part; it has only integer part. For example: 1234.

**2.** In human vocabulary, the **decimal number** refers to a number with integer part, decimal point, and fractional part. For example: 12.37. In programming language, it is known as **floating point number** or simply **float**.

**3.** How to store a integer number like 1234 into computer memory?  
We simply execute the following instruction; as a result, the given number is automatically saved into two consecutive memory locations. The given integer number is always saved as binary bits in the computer memory. This is to say that the given integer number of base 10 is converted into binary number of base 2 (123410 ---->00000100110100102 ---->04D216) and then it is saved into computer memory. As one memory location can hold 8-bit data, we need two memory locations to hold the given data.

unsigned int x = 1234;

===> unsigned int x = 0x04D2;

**4.** How to store a floating point number like 12.37 into computer memory?  
**(1)** According to IEEE-754 Standard (aka binary32 format), a float number (+ve or -ve) will be coded into 32-bit data as per following template in order to store it into computer memory. This is what we call '32-bit representation of a float number'.  
  
Figure-1: 32-bit representation of a float number (for example: 0.15625)

**(2)** Manual procedure to obtain binary32 formatted value for the float number 0.15625.  
**(a)** Calculating binary bits of 0.15625

0.15625x2 = 0.3125

0.3125x2   = 0.625

0.625x2     = 1.25

1.25x2       = 0.50

0.50x2       = 1.00

... continue until the residual is exhausted to 0 or 23 fractional bits are accumulated

**(b)** 0.1562510 = 0.001012  
==>0x20+0x2-1+0x2-2+1x2-3+0x2-4+1x2-5  
==> 0 + 0 + 0 + 0.125 + 0 + 0.03125  
==> 0.15625

**(c)** 0.001012  
==> 1.012\*2-3

**(d)** binary32 format value of 0.15625 as per Fig-1.  
Sign (1-bit: b31: 0 (the given number is +ve)  
Biased exponent (8-bit : b30 - b23): -3 (from Step-4c) + 127 (fixed bias) = 124 = 7Ch  
fraction (23-bit: b22 - b0) 01000000000000000000000 (from Step-4c)

**(e)** binray32 value: 0(sign) 01111100(biased exponent) 01000000000000000000000(fraction)

**(f)** Arranging as nibbles: 0011 1110 0010 0000 0000 0000 0000 0000

**(g)** Presenting in hex: 3E200000 (4-byte = 4x8 = 32-bit)

**5.** Programming codes to generate binary32 formatted value for a float number (say: 0.15625)  
**(1)** When we make the following declaration using the keyword **float**, the binary32 formatted 32-bit (4-byte) value is automatically saved into 4 consecutive memory locations. The low order memory location holds the lower byte of the data.

float x = 0.15625;

**(2)** We may execute the following codes to collect the 32-bit data from the unseen memory locations and show it in the Serial Monitor.

float x = 0.15625;

unsigned long \*ptr;

ptr = (unsigned long\*) &x;

unsigned long m = \*ptr;

Serial.println(m, HEX); *//shows: 3E200000*

**6.** Precision of a binary32 formatted float number  
Precision refers to the number of digits after the decimal point that we can present. For the binary32 formatted float number, the precision is 23 digits.

**7.** Accuracy of a binary32 formatted float result  
Accuracy refers to 'how many' digits are coming out exactly in the result during the processing of two/more float numbers. For example:

float x1 = 12.12345678;

float x2 = 23.12345678;

----------------------------------------------

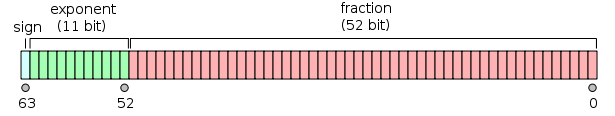
Sum on manual calculation: 35.24691356

Sum **using** program codes:

float x = x1 + x2;

Serial.print(x, 23); *//shows: 35.246913 90991210937500000*

The accuracy is 6-digit; whereas, in manual calculation we have 8-digit accuracy.

**8.** double type floating point number.  
This is a 64-bit representation of a floating point number. In this representation, there are 53 fraction bits after the decimal point in addition to a very large integer part. The result of the processing of double type float numbers gives about 15-digit accuracy. The encoding format (known as binary64) is:  
  
Figure-2: 64-bit (binary64 formatted) representation of a float number

**9.** Arduino UNO, NANO, and MEGA (all 8-bit AVR) supports only 32-bit representation of float number. In these Arduinos, the keywords **float** and **double** mean the same thing -- the binary32 format.

**10.** Arduino DUE supports both 32-bit and 64-bit representation of decimal numbers via the keywords **float** and **double** respectively.

I apologize and seek correction if any misconception is being carried by this post.

