FIXED-POINT MATH IN TIME-CRITICAL C APPLICATIONS

Mark B. Kraeling
Cummins Engine Company
1460 National Road, MS C7004
Columbus, Indiana 47201
email: kraeling@cel.cummins.com

ABSTRACT

When optimizing C to improve the speed of execution, using fixed-point math as opposed to floating-point emulation can help the user make big improvements. It is the single largest improvement a programmer can make in C to reduce execution time.

This paper will start with floating-point equations, and show the steps necessary to convert them to fixed-point equations. Basic equations will be used at the beginning, but then will progress to more advanced problems in the fixed-point implementation. The paper will also analyze the assembly output after converting to fixed-point, and data will be presented from multiple platforms that shows the speed improvement.

MATHEMATICAL IMPLEMENTATIONS

There are three distinct ways that math can be implemented on microprocessors and microcontrollers today. Each carries different complexity and costs to the platform.

The first implementation of math is hardware-assisted floating-point. This involves the use of hardware to perform floating-point operations. The floating-point hardware can take floating-point instructions used in the software code and manipulate them. Such hardware is complex, and takes up a significant amount of space and power in the microprocessor layout. Most microprocessors and microcontrollers used for embedded or smaller platform development do not have this hardware, since the cost of adding the hardware can be expensive.

The second implementation is floating-point emulation. This involves using the microprocessors' integer operations to perform floating-point math. Special compiler libraries convert the floating-point used in the software to use the available integer operations. These floating-point libraries take up a significant amount of RAM and fixed memory space, as well as more processing time. The amount of program space the libraries take up is dependent on the complexity of floating-point used in the software. Program space and execution time are also dependent on the compiler vendor, since each vendor implements the floating-point libraries differently.

The third implementation is using fixed-point math. Signed or unsigned integers are used to represent fractional numbers. This representation is done through the use of a fixed scaling for the number. Using fixed-point math allows the compiler to use normal microprocessor machine instructions for mathematical operations. There is no need for the compiler to pull in special libraries. This implementation yields low per platform costs since floating-point hardware is not required. It also saves program space and execution time since no special libraries for floating-point manipulation need to be called.

FLOATING-POINT AND FIXED-POINT FORMAT

Floating-point numbers used in software are declared using the "float" declaration in ANSI C. When these floating-point numbers are declared,

ISBN# 0-7803-3274-1

the resulting space allocated for the variable is divided up between the sign bit, mantissa, and exponent. The mantissa controls the accuracy of the number, using the bits allocated to it to hold the floating-point number not including the exponent. The exponent controls the scaling of the number by using a number to show decimal place movement.

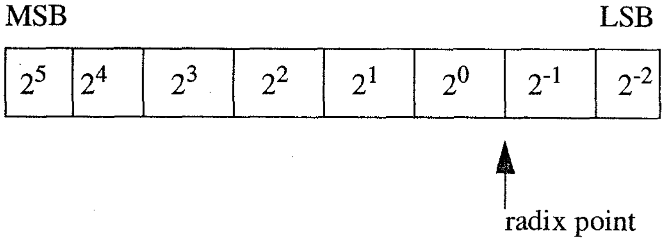
When floating-point numbers were first implemented my microprocessor manufacturers, no clear standard for allocation of the mantissa, exponent, and sign bit existed. IEEE-754, the standard for floating-point arithmetic, was then completed to show this data representation and how it should be used. The allocation implementing a 32-bit "float" in ANSI C consists of a sign bit, an 8-bit biased exponent, and a 23-bit mantissa. The allocation implementing a 64-bit "double float" in ANSI C consists of a sign bit, an 11-bit biased exponent, and a 52-bit mantissa.The biasing of the exponent consists of the exponent value with a fixed +127 offset to take care of the signed issues. The calculation of the mantissa consists of determining the fraction field, adding the implicit normalized bit, and determining the value based on this number and the addition of exponents. The process becomes more complicated as more decimal digits are added to the floating-point number.

Below are some floating-point numbers and the values placed in the sign-bit, the biased exponent, and the mantissa. The floating-point libraries would then have to manipulate this final result 32-bit number.

Number	Sign (bit 31)	Biased Exponent (bits 30-23)	Mantissa (bits 22-0)
1.0	0	0b01111111	0b000000000...
0.0	0	0b00000000	0b000000000...
-1.0	1	0b01111111	0b000000000...
0.75	0	0b01111110	0b100000000...

Fixed-point numbers used in software are declared with integral data types. These data types and their corresponding size are compiler and platform dependent. Using these fixed-point numbers allows the compiler to use machine instructions already built in to the microprocessor to manipulate the numbers. Cases where the programmer needs to manipulate numbers larger than the microprocessor size will cause the compiler to use libraries, much like the libraries used for floating-point manipulation.

There are two popular ways to implement integer and fixed-point math. The first way is through the use of a radix point within an integer. This point marks the spot where bits to the right represent a fractional base-2 addition to the number on the left of the radix point. The first bit to the right of the radix point would represent 0.5 (2^{-1}), the second 0.25 (2^{-2}), and so on. This way the programmer can represent fractional quantities but still use the built-in microprocessor math manipulation instructions. As the radix point is moved to the left, the fractional part increases but the min/max of the variable decreases. Shown below is an example layout of a 8-bit unsigned integer with a radix point that allows a resolution of 0.25 and a range of 0 - 63.75.



Fixed-Point Number	Whole Part (left of radix)	Fractional Part (right of radix)	Final Result
1.0	0b000001	0b00	0x04
0.0	0b000000	0b00	0x00
30.5	0b011110	0b10	0x7A
0.75	0b000000	0b11	0x03

The second way is through the used of a fixed scale factor for each variable used. The radix point only allows scale factors of base-2 numbers, such as 1, 2, 4, 8, 16, etc. In order to use a fixed-scale factor, such as 10, the programmer keeps track of this scaling then remembers to divide by 10 to get the final result after manipulation. This is a less desirable way to implement fixed-point math since it doesn't allow the compiler to take advantage of bit-shifting for certain multiply and divide operations.

In both implementation strategies for fixed-point math, the programmer must keep track of the radix point or the non base-2 scaling. It carries with it some added complexity since the programmer must now do extra operations to take care of scaling issues. These issues and ways to implement fixed-point operations will now be shown.

MOVING FROM FLOATING- TO FIXED-POINT

The improvements in moving from floating-point to fixed-point will vary by compiler and platform. In order to see the increase in speed of execution, two different compilers for two different platforms were used. On an 8-bit platform, compiler A took 44 clock cycles to set up and multiply two 8-bit fixed-point numbers. It took compiler A 690 clock cycles to set up and multiply two floating-point numbers. On the same 8-bit platform, compiler B took 48 clock cycles to set up and multiply two 8-bit fixed-point numbers. It took compiler B 650 clock cycles to set up and multiply two floating-point numbers. On a 32-bit platform, compiler C took 65 clock cycles to set up and multiply two 32-bit fixed-point numbers. It took compiler C 95 clock cycles to set up and multiply two floating-point numbers. On the same 32-bit platform, compiler D took 72 clock cycles to set up and multiply two 32-bit fixed-point numbers. It took compiler D 112 clock cycles to set up and multiply two floating-point numbers. These results are summarized in the table below.

Platform Size	Compiler	Fixed-Point Time (clk)	Floating-Point Time (clk)
8-bit	A	44	690
8-bit	B	48	650
32-bit	C	65	95
32-bit	D	72	112

As discussed in the previous section, with floating-point numbers you do not need to worry about the scaling of the number. When floating-point numbers are multiplied, the compiler will multiply the numbers and take care of the mantissa, sign, and exponent for you. For fixed-point numbers, the programmer must keep track of the scaling of the numbers in order to use them. In order to see how scaling is taken care of, this section will walk through some examples and show what is needed for the floating-point and fixed-point cases.

Consider the following formula with variables A, B, C, D:

$$A = B + C - D$$

When using floating-point to calculate the result, the type "float" is used in the software code. Each of these variables are declared with this type, and added together normally. The compiler automatically adjusts the variables to keep maximum accuracy while also preventing overflow. The following code segment shows how these floating-point numbers are manipulated.

```
float A, B, C, D;

A = B + C - D;
```

When using fixed-point to calculate the result, you must declare an integral type that matches the accuracy and range of each variable. Say for instance that variables B, C, and D each had a minimum of zero, a maximum of 300, and a required accuracy of 0.05. In order to determine the scaling of the number, scaling is equal to 1/accuracy or greater. For this example the required scaling is at least 20. When using addition and subtraction, it is much easier to manipulate fixed-point numbers when the scaling for the variables is the same, so variables A, B, C, and D will have a scaling of 20.

In order to determine what size of integral type to declare, take the scaling and multiply by the range of the variable. For types B, C, and D, multiplying 20 by 300 yields 6000. For this example, B, C, and D will be declared as 16-bit unsigned integers. Since variable A is the result of the calculation, its range is determined by B, C, and D. For a maximum value, A is equal to the maximum of B plus the maximum of C minus the minimum of D. For a minimum value, A is equal to the minimum of B plus the minimum of C minus the maximum of D. So A will have a minimum of -300, a maximum of 600, and keep the same accuracy requirements. Taking the scaling of type A and multiplying by the range, or multiplying 20 by 900, yields 18000. For this example, A will be declared as a 16-bit signed integer.

Since the scaling for variables A, B, C, and D are the same, no special manipulation needs performed in the calculation. The following code segment shows how these fixed-point numbers are manipulated. ANSI C requires type casting the unsigned variables B, C, and D to yield a signed result. This type casting is not needed if all variables are declared as integers.

```
unsigned int B, C, D;
int          A;

A = (int)B + (int)C - (int)D
```

For this example we could also declare the scaling to be 32. In this way, we could keep base-2 scaling which is preferred to help the compiler utilize bit-shifting instead of multiplies and divides, making the code faster. The code written above would not change in any way if you were to pick a scaling of 32 instead of 20, since the integral types would not change. The programmer will always need to remember the scaling used for the variables so whenever they are referenced or needed in the code, scaling issues can be handled.

For the same example used above, lets say that the variables B, C, and D are already used in other places in the code. The scaling for B and C is 16, and the scaling of D is 32. Since they are used other places, it is too much of a mess to change what the scaling is without affecting other code locations. 16-bit unsigned integers were chosen as the integral types for these variables. Taking the minimum and maximum and multiplying by the scaling of each of the variables should be within the 16-bit unsigned integer boundaries.

Since the scaling of variables B, C, and D are different, they cannot be added normally. The radix point for the variables is in different places, so each of the bits in the integral type does not carry the same weight. In order to add and subtract these variables, they must have the same scaling. In order to keep as much accuracy as possible, adjust the lower scaling numbers (B,C) to the higher scaling numbers (D). The result (A) should maintain the higher scaling whenever possible. The code segment below shows the equation implemented with the different scalings for A, B, C, and D:

```
unsigned int   B, C, D;
int            A;

A = (int)( B + C ) * 2 - D;
```

The code segment above shows B and C being adjusted to a scaling from 16 to 32 by adding weight to the numbers. The table below shows the real values of B, C, and D, the software variable fixed-point values of B, C, and D, and the result of the calculation.

Real B Value Fixed-point B Variable	Real C Value Fixed-point C Variable	Real D Value Fixed-point D Variable	Real A Result Fixed-point A Result
2.375 2.375 * 16 = 38	4.4375 4.4375 * 16 = 71	.03125 .03125 * 32 = 1	6.78125 (38 + 71)*2 - 1 = 217
1 1 * 16 = 16	1 1 * 16 = 16	1 1 * 32 = 32	1 (16 + 16)*2 - 32 = 32

Notice in the table above that the fixed-point A result is the scaling of A multiplied by the real A result. This is a good check to make sure that the calculation is written correctly.

Consider the following formula with variables A, B, C, D:

$$A = B * C / D$$

When using floating-point to calculate the result, the type "float" is again used in the software code. Each of these variables are declared with this type, and multiplied together normally. The compiler automatically adjusts the variables to keep maximum accuracy while also preventing overflow. The following code segment shows how these floating-point numbers are manipulated.

```
float A, B, C, D;

A = B * C / D;
```

When using fixed-point to calculate the result, you must declare an integral type that matches the accuracy and range of each variable. Say for instance that variables B, C, and D each had a minimum of zero, a maximum of 400, and a required accuracy of 0.1. In order to determine the scaling of the number, scaling is equal to 1/accuracy or greater. For this example the required scaling is at least 10. It is preferable to use base-2 numbers, so a scaling of 16 will be used, which is greater than the required 10. When using multiplication and division, it isn't as important to try to keep the same scaling as it was with addition and subtraction. But keeping consistency will help with later calculations.

In order to determine what size of integral type to declare, take the scaling and multiply by the range of the variable. For types B, C, and D, multiplying 16 by 400 yields 6400. For this example, B, C, and D will be declared as 16-bit unsigned integers. Since variable A is the result of the calculation, its range is determined by B, C, and D. For a maximum value, A is equal to the maximum of B multiplied by the maximum of C divided by the minimum of D. For a minimum value, A is equal to the minimum of B multiplied by the minimum of C divided by the maximum of D. So A will have a minimum of 0 and a maximum of 40,960,000, as long as D is not equal to zero. Whenever numbers are multiplied in an equation, the accuracy of the result increases. Whenever they are divided, the accuracy of the result decreases. When using multiplication and division, it is important to determine what the required accuracy is before laying out the equation. For this example, let's keep the accuracy at 16. Taking the scaling of type A and multiplying by the range, or multiplying 16 by 40,960,000, yields 655,360,000. For this example, A will be declared as a 32-bit unsigned integer.

Since we are multiplying and dividing, the scalings of the fixed-point numbers interfere in the result of the calculation. The scaling of the final result A is equal to the scaling of B times the scaling of C divided by the scaling of D. This comes out to be 16, which is what we require. So scaling does not interfere in the calculation.

```
unsigned int       B, C, D;
unsigned long int A;

if ( D != 0 )
   A = (unsigned long int)B     *
          (unsigned long int)C     /
          (unsigned long int)D;
```

For the same example used above, lets say that the variables B, C, and D are already used in other places in the code. The scaling for B and C is 16, and the scaling of D is 8. Since they are used other places, it is too much of a mess to change what the scaling is without affecting other code locations. 16-bit unsigned integers were chosen as the integral types for these variables. Taking the minimum and maximum and multiplying by the scaling of each of the variables should still be within the 32-bit unsigned integer boundaries.

For this example, since D only has an scaling of 8, we will choose the scaling of A to also be 8. The scaling of the final result A without manipulation is the scaling of B (16) times the scaling of C (16) divided by the scaling of D (8), which yields 32. In order to get A to have a scaling of 8, we must divide the result by 4 (32/4 = 8). The code segment below shows the equation implemented with the different scalings for A, B, C, and D:

```
unsigned int       B, C, D;
unsigned long int A;

A = (unsigned long int)B     *
       (unsigned long int)C    /
       (unsigned long int)D    /    4;
```

The code segment above shows the equation adjusted by 4 to get A to the proper scaling. The table below shows the real values of B, C, and D, the software variable fixed-point values of B, C, and D, and the result of the calculation.

Real B Value Fixed-point B Variable	Real C Value Fixed-point C Variable	Real D Value Fixed-point D Variable	Real A Result Fixed-point A Result
2.375 2.375 * 16 = 38	4.4375 4.4375 * 16 = 71	1.25 1.25 * 8 = 10	8.43125 38 * 71 / 10 / 4 = 67
1 1 * 16 = 16	1 1 * 16 = 16	1 1 * 8 = 8	1 16 * 16 / 8 / 4 = 8

Notice in the table above that the fixed-point A result is the scaling of A multiplied by the real A result. This is a good check to make sure that the calculation is written correctly.

FIXED-POINT IMPLEMENTATIONS

Now that the basis for fixed-point and the improvements using fixed-point have been covered, this section will show different implementations of fixed-point and how to deal with special cases. It will also show some examples of things to watch out for when implementing fixed-point math equations.

As mentioned in the previous section, wherever base-2 fixed-point constants are used, the compiler may substitute the operations with left- and right-shift operations. This is one of the added benefits of using fixed-point math. Below is an example where bit shifting is used in place of a machine divide.

```
Code:
     static unsigned char j, k, n;

     static void example( void )
     {
          j = k * n / 2;
     }
```

```
Assembler Output:
     ldab  n    ; places n in register
     ldaa  k    ; places k in register
     mul        ; performs machine multiply
     lsrb       ; bit shift 1 to right (/2)
     stab  j
     rts
```

It is also important to remember that whenever a fixed-point divide is used, the resulting decimal digits from the divide are chopped off. This can cause inaccuracies in your calculation. In order to keep as much accuracy as possible, multiply in the equation first then divide. In the example A = B * C / D, multiply B and C first then divide by D. If the scaling needs adjusted in the equation by multiplying, then adjust the scaling first before dividing.

If B and C have a scaling of 4, and D has a scaling of 16, and we wish A to have a scaling of 4, we need to multiply the equation by 4 to adjust the scaling. Remember, this is calculated by taking the scaling of B (4), multiplying by the scaling of C (4), and dividing by the scaling of D (16), which comes out to 1. We want A to have a scaling of 4, so we need to multiply by 4. Multiplying is done first, then dividing in the code segment below.

```
unsigned int      B, C, D;
unsigned long int A;

if ( D != 0 )
  A = (unsigned long int)B  *
       (unsigned long int)C  *
       4  /  (unsigned long int)D;
```

When adding and multiplying in the same equation, the scaling becomes even more confusing. Remember when adding or subtracting, the scaling of the two variables must be the same. Remember when multiplying, the scaling of the result will be equal to the multiplication of the two scale factors. Remember when dividing, the scaling of the result will be equal to the division of the two scale factors. To come up with an example of having to address many scaling issues, consider the following variables without worrying about ranges. B has a scaling of 8, C has a scaling of 4, and D has a scaling of 16. The result A needs to have a scaling of 8. The equation A = (B + C) * D will be used.

```
unsigned int        B, C, D;
unsigned long int A;

A = ( (unsigned long int)B +
     ( (unsigned long int)C * 2 ) ) *
     (unsigned long int)D
     / 16;
```

In the code segment above, the multiplication by 2 adjusts C to have the same scaling as B (8) so they can be added together. This result is then multiplied by D. The last step is to divide the result by 16 so it can have the correct units for A. Again, make sure the scaling is the same before adding, and adjust the scaling as needed after multiplying and dividing.

When using fixed-point math, minimize the number of different scale factors as much as possible. This can be achieved by always using a standard scaling for types of parameters. For instance, whenever engine speed is used in the code, use a fixed scaling such as 4. Having an accuracy of 1/4 RPM wherever it is used in the code will help keep consistency and minimize extra manipulation in the calculations. When comparing engine speed to limits or other engine speed values, regular calculations can be made. Keep common scale factors in a header file group together, wherever it is appropriate.

Avoid using "magic numbers" as well. The values "2" and "16" in the code above come from fixed numbers. The "2" could actually be a #define that is defined as the scaling of B divided by the scaling of C. The same could be done for "16". In this way, if someone decides to change the scaling of one of the variables, it is less work to change it, since the equations that use this variable will automatically adjust when re-compiled. Ranges should still be checked to make sure there are no overflow problems.

Don't forget when multiplying fixed-point numbers, to cast the variables when necessary. The compiler normally interprets the result size of the calculation to be whatever size that is being used to calculate it. When multiplying 16-bit numbers and placing in a 32-bit result, you must cast the 16-bit number to 32-bit before multiplying.

CONCLUSIONS

Whether or not to use fixed-point math instead of floating-point emulation should be decided based on execution speed and memory constraints. As shown in the examples, implementing fixed-point math can be tricky. The scaling of the fixed-point numbers must constantly be considered, otherwise using them in multiplication can yield incorrect results.

Make sure when implementing fixed-point math that your calculations are double-checked. This can easily be done by making up some floating-point numbers and calculating what the result should be. Then convert these numbers to their fixed-point counterparts, and plug into the equation. Calculate the number as written in the code with a calculator, and chop off any decimal places after any divide or shift operation. Once the final fixed-point result is found, divide by the scaling of the result to make sure it is equal to the expected result.

The suggestions and ideas presented in this paper should be tried out on your particular platform and compiler. It is important to measure the percent improvement in moving from floating-point emulation to fixed-point to quantify the change. If there is a large measured improvement, the payoffs in execution time, RAM, and fixed-memory are right around the corner.