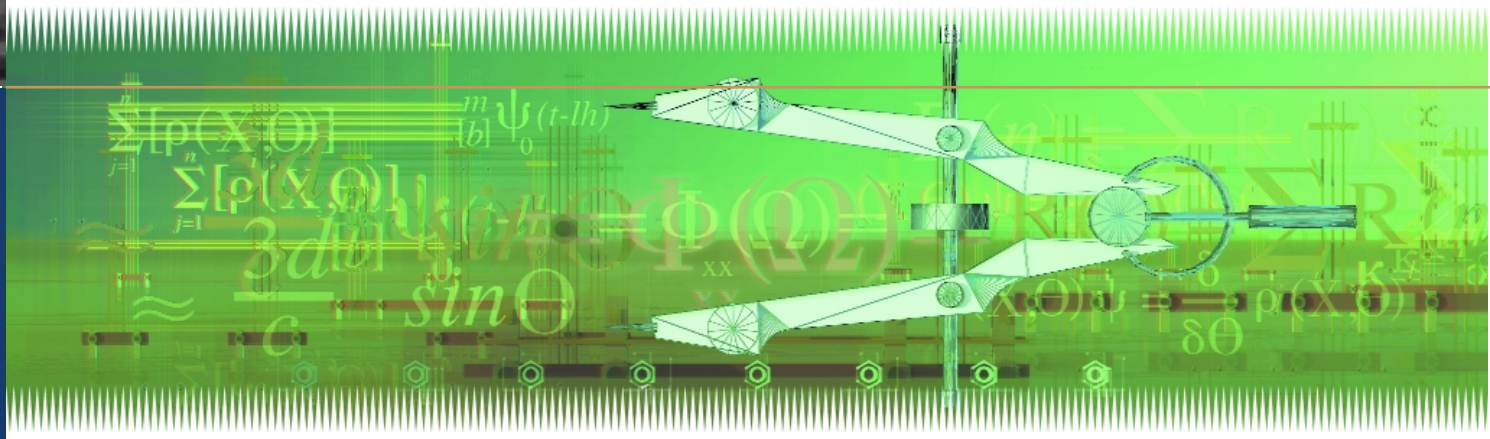
JOE LEMIEUX



Fixed-Point Math in C

Floating-point arithmetic can be expensive if you're using an integer-only processor. But floating-point values can be manipulated as integers, as a less expensive alternative.

One advantage of using a high-level language is the native support of floating-point math. This simplifies the task of programming and increases the accuracy of calculations. When used in a system that includes a hardware floating-point math unit, this can be a very powerful feature. However, if floating-point math is used on a microprocessor that supports only integer math, significant overhead can be incurred to emulate floating point, both in ROM size and in execution time. The alternative, used by assembly language programmers for years, is to use fixed-point math that is executed using integer functions. This article will discuss a method to implement fixed-point math in C.

Fixed-point math typically takes the form of a larger integer number, for instance 16 bits, where the most significant eight bits are the integer part and the least significant eight bits are the fractional part. Through the simple use of integer operations, the math can be efficiently performed with very little loss of accuracy. Unfortunately, C does not provide native support for fixed-point math. The method presented in this article is essentially the same as in assembly language and is described in the following steps.

Initialization

The first step in the fixed-point math algorithm is to initialize the system. This step defines the parameters used later in the development of the system. To illustrate each step, assume that your application requires a variable that ranges from 0 to 1 with a granularity of 0.01, another variable family that ranges from 0 to 100 with a granularity of 0.1, and a third variable family that ranges from –1,000 to 1,000 with a granularity of 0.01. The steps are performed as follows.

First, determine the maximum absolute value M that you wish to calculate for each class of fixed-point variable. The value of M for each example requirement is 1, 100, and 1,000, respectively.

Second, calculate the number of bits x required to store this number such that $2^x \geq M \geq 2^{x-1}$. If the number is to be

If floating-point math is used on a microprocessor that supports only integer math, significant overhead can be incurred to emulate floating point, both in ROM size and in execution time.

signed, add 1 to *x*. For our example requirements, *x* is 1, 7, and 11, respectively.

Then, determine the granularity G that is required. The example requirements define the granularity and no further calculation is needed.

Next, calculate the number of bits *y* required to provide this granularity such that $1/2^y \le G \le 1/2^{y-1}$. For our example requirements, this is 7 (1/128 = .0078125), 4 (1/16 = .0625), and 7.

Finally, the minimum number of bits required is *x*+*y*. Round this sum up to either 8, 16, or 32. This sum will be referred to later as the FULLSIZEINT. If this sum is greater than the maximum integer size in your system, then floating-point numbers are required and will probably be more efficient. The results for our example requirements for all the initial parameters are shown in Table 1. The resultant columns show the actual ranges and granularity available after the bit requirements are complete.

Definition

The second step in the algorithm is to create the definitions for C that are required to implement the variables. This consists of typedefs and macros, with the possibility that a function may need to be developed.

Typedef

Definition of the fixed-point algorithm requires a typedef for each fixed-point variable type that will be used by the system. This typedef is a union with an embedded structure as follows. The structure assumes that the compiler assigns the bits in an integer from most significant to least significant. If this is not the case, reverse the declaration of the structure members integer and fraction in the structure. This typedef is not portable, and is best put in a unique header file:

```
typedef union FIXEDX_Ytag {
  FULLSIZEINT full;
  struct partX_Ytag {
    FULLSIZEINT integer: x;
    FULLSIZEINT fraction:
      FULLSIZEINT-x;
  } part;
} FIXEDX_Y;
```

FULLSIZEINT is either `long`, `int`, `short`, or `char` and either signed or unsigned and cannot be longer than the maximum length integer available, or the structure will not work. If FULLSIZEINT is longer than this maximum, then floating point needs to be used.

Notice that the fractional portion of the structure is calculated as FULLSIZEINT-*x*, instead of using *y* as the size of the member. Since a complete integer is available, I have chosen to increase the granularity of the system in order to decrease the error of the calculations. In your application, you may wish to increase the integer member in order to provide a method to check for math overflow.

TABLE 1 Example initialization results

No.	Range	M	x	Resultant range	Granularity	y	Resultant granularity	Number of bits required
1	0–1	1	1	0–1	0.01	7	0.0078125	8
2	0–100	100	7	0–127	0.1	4	0.0625	16
3	-1000–1000	1000	11	-1024–1023	0.01	7	0.0078125	32

Macros

After the typedefs have been declared, macros need to be defined. Substitute the actual value for the equations using *X* and *Y* in each of the following macros.

Define a macro to convert a value into the fixed-point variable. A is the integer part and B is the decimal part, expressed as normal decimal. These values must be hard-coded constants so the preprocessor and compiler can completely resolve the macro into an integer. If either A or B are variables, then the macro will actually generate floating-point code and eliminate the savings of this algorithm. Check the results of the compile in the listing to make sure that the compiler functioned properly. (To do this, you must have the compiler interlace the assembly code with the C statements.)

```
#define FIXEDX_YCONST(A,B) (FULL-
  SIZEINT)((A<<Y) + ((B +
  1/(2^(Y+1)))*2^Y))
```

Define macros to perform multiplication and division of the fixed-point variables:

```
#define MULTX_Y(A,B) (FULL-
  SIZEINT+1)(A.full*B.full+
  2^(Y-1))>>Y
#define DIVX_Y(A,B) (FULL-
  SIZEINT+1)((A.full<<Y+1)/
  B.full)+1)/2
```

where FULLSIZEINT+1 is the next largest integer over *X*+*Y*. If FULLSIZEINT is the largest available inte-

LISTING 1 Example typedef and macros

```
*     Range 0–1.9921875
*     Granularity 0.0078125

typedef union FIXED1_7tag {
    unsigned char full;
    struct part1_7tag {
      unsigned char fraction: 7;
      unsigned char integer: 1;
    } part;
} FIXED1_7;

#define FIXED1_7CONST(A,B) (unsigned char)((A<<7) + ((B + 0.00390625)*128))
#define MULT1_7(A,B) (unsigned short)(A.full*B.full+64)>>7
#define DIV1_7(A,B)(unsigned short)((A.full<<8)/B.full)+1)/2
```

LISTING 2 Multiply and divide for 32-bit variables

```
int MULT11_21(FIXED11_21 a, FIXED11_21 b)
{
     int temp,result;
     char sign = 0;

     if (a.full < 0)
     {
       sign = 1;
       a.full = -a.full;
     }
     if (b.full < 0)
     {
       sign^ = 1;
       b.full = -b.full;
     }

     result = (((a.full & 0x0000FFFF) * (b.full & 0x0000FFFF))+1048576)>>21;
     result = result + ((((a.full>>16) * (b.full & 0x0000FFFF))+16)>>5);
     result = result + ((((b.full>>16) * (a.full & 0x0000FFFF))+16)>>5);
     temp = (a.full>>16) * (b.full>>16);
     result = result + (temp<<11);
     return (result * -sign);
}

int DIV11_21(FIXED11_21 a, FIXED11_21 b)
{
     double temp;
     FIXED11_21 result;
     unsigned char sign = 0;

     temp = (double)a.full/(double)b.full;
     if (temp<0)
     {
       temp = -temp;
       sign = 1;
     }
     result.part.integer = temp;
     result.part.fraction = ((temp-result.part.integer)*4194304 + 1)/2;
     result.part.integer *= -sign;
     return (result.full);
}
```

ger, then either floating point must be used, or a subroutine is required for multiply and divide.

Listing 1 shows the definitions for one of our example requirements sets. In this file, I first define the integer typedefs so the code that is written is portable. Next, the typedefs and macros are defined.

Listing 2 shows the multiply and divide routines used for the 32-bit example, assuming that a 64-bit integer is not available in the system, since that cannot be done in a macro. Notice that in the divide routine, I am using a floating-point number for the calculation. If your system requires division, the amount of memory used cannot be reduced this way, but the add, subtract, compare, and multiply routines will still execute faster than the comparable floating-point routines.

Usage

When using the fixed-point variables in an application, first declare variables using the new typedef. Next, use the new variables in the application as defined in the following paragraphs.

When adding, subtracting, or performing logical comparisons on two variables with the same-resolution fixed-point numbers, use the `.full` portion of the union and perform straight integer arithmetic. Listing 3 shows a small routine that performs each operation and prints the result for the number of type `FIXED1_7`. In this routine, the numbers are added and the result is printed to the display. A set of comparisons are then done to determine if a subtraction would work, and if so, the subtraction is performed and the results written to the display. If a subtraction would result in an unsigned underflow, an error is displayed.

When adding, subtracting, or performing logical comparisons on two variables with different-resolution fixed-point numbers, first scale the different resolution number to the same resolution as the result.

LISTING 3 Simple math on same granularity variables

```
void Test1_7(FIXED1_7 a, FIXED1_7 b)
{
      FIXED1_7 temp;

      printf("Results of operations on 1_7 variables\n");
      temp.full = a.full + b.full;
      printf("Addition result is %d.%2.2d\n", temp.part.integer,
        (temp.part.fraction*100+64)/128);
      if (a.full < b.full)
      {
        printf("a is less than b. Subtraction overflows.\n");
      }
      if (a.full == b.full)
      {
        printf("a is the same as b. Result = 0.\n");
      }
      if (a.full > b.full)
      {
        temp.full = a.full - b.full;
        printf("Subtraction result is %d.%2.2d\n", temp.part.integer,
        (temp.part.fraction*100+64)/128);
      }
}
```

LISTING 4 Simple math on different granularity variables

```
void Test7_9(FIXED7_9 a, FIXED1_7 b)
{
      FIXED7_9 temp;

      printf("\nResults of operations on 7_9 and 1_7 variables\n");
      temp.full = a.full + (b.full<<2);
      printf("Addition result is %d.%1.1d\n", temp.part.integer,
        (temp.part.fraction*10+256)/512);
      if (a.full < (b.full<<2))
      {
        printf("a is less than b. Subtraction overflows.\n");
      }
      if (a.full == (b.full<<2))
      {
        printf("a is the same as b. Result = 0.\n");
      }
      if (a.full > (b.full<<2))
      {
        temp.full = a.full - (b.full<<2);
        printf("Subtraction result is %d.%1.1d\n", temp.part.integer,
           (temp.part.fraction*10+256)/512);
      }
}
```

Listing 4 shows a small routine that performs some mixed operations and prints the results. This routine performs the same function as Listing 3.

When multiplying or dividing variables of the same-resolution fixed-point numbers, use the macros previously defined. Listing 5 shows a small routine that performs each operation and prints the result. This routine simply performs the multiply and divide without checking, then displays the results. With multiply and divide, you cannot pass the full value, so you must pass the structure name.

When multiplying or dividing variables of different-resolution fixed-point numbers, use the same macro as the result and scale the operands as with addition and subtraction. Listing 6 shows a small routine that performs an operation like this and prints the results.

The value can be displayed using the integer and fractional portions of the structure.

Words of caution

When these algorithms are implemented, a few areas of caution need to be addressed. If they aren't, erroneous results may be obtained.

The C language does not check integer arithmetic for overflows or underflows. The programmer must take care to either prevent or check for these conditions. This is usually accomplished by using a subroutine, possibly in assembly language, for each math operation and checking the limits. However, it can also be accomplished by adding bits to the integer portion, and limiting to a smaller number after each math operation.

The compiler may perform signed integer arithmetic in a subroutine and may not provide a tremendous benefit.

A rounding error may be injected in the result due to the value of the least significant bit. Since the expectation is an LSB of 0.1, and in fact

LISTING 5 Multiplication and division on same granularity variables

```
void Test7_9_X(FIXED7_9 a, FIXED7_9 b)
{
      FIXED7_9 temp;

      printf("\nResults of multiply and divide on 7_9 variables.\n");
      temp.full = MULT7_9(a,b);
      printf("Multiply result is %d.%1.1d\n", temp.part.integer,
        (temp.part.fraction*10+256)/512);
      temp.full = DIV7_9(a,b);
      printf("Divide result is %d.%1.1d\n", temp.part.integer,
        (temp.part.fraction*10+256)/512);
}
```

LISTING 6 Multiplication and division on different granularity

```
void Test11_21(FIXED11_21 a, FIXED7_9 b)
{
      FIXED11_21 temp;

      printf("\nResults of multiply and divide on 11_21 and 7_9 variables.\n");
      temp.full = b.full << 12;
      temp.full = MULT11_21(a,temp);
      printf("Multiply result is %d.%2.2d\n", temp.part.integer,
        (temp.part.fraction*100+1048576)/2097152);
      temp.full = b.full << 12;
      temp.full = DIV11_21(a,temp);
      printf("Divide result is %d.%2.2d\n", temp.part.integer,
        (temp.part.fraction*100+1048576)/2097152);
}
```

the value is something less, an error of ±1 LSB is injected that can compound during mathematical operations. For example, using our FIXED1_7 example, 0.02 is represented as a full number of 3, and 0.06 is represented as a full number of 8. The sum should be 0.08, or a full number of 10. The actual result is 11, which is closer to 0.09. To limit this error, use a larger number such as 16 bits, and increase the granularity from 7 bits to 15 bits.

Fixed-point math provides a small, fast alternative to floating-point numbers in situations where small rounding errors are acceptable. After implementing the algorithms described in this article, your application will be able to harness the power of C and still retain the efficiency of assembly.

All examples in this article were tested using Microsoft Visual C++ v. 4.0 Standard edition. The source code for the test files can be found on the web at *www.embedded.com/code.htm.* **esp**

Joseph Lemieux is a senior applied specialist with EDS Embedded Solutions in Troy, MI. He holds an MS in electrical engineering from the University of Michigan and has been writing software for embedded systems in the automotive industry for over 17 years. He can be reached by e-mail at joe.lemieux@eds.com.