# Implementing a C++ Fixed-Point Class for Embedded Systems

Oliver Schloesser
University of Applied Sciences NW Switzerland, Institute of Microelectronics.
Steinackerstrasse 1, 5210 Windisch, Switzerland, E-Mail: oliver.schloesser@fhnw.ch

***Abstract*—If small micro controllers are used for calculation intensive programs, they often contain a floating-point unit. This leads to a significant increase in the cost even tough this is not always needed. In this paper, the implementation of a C++ fixed-point class is shown and discussed. Although the primary purpose is for small embedded systems, it is not restricted to that.**

## I. Introduction

Modern micro controller mostly give the option of adding a floating-point unit to the core. This will speed up calculation using Float and Double types. The prize for a small device will increase 30 to 50%. If no floating-point unit is used, mathematical floating-point operations are evaluated with library functions, which usually is quite inefficient. If a more expensive micro controller is not an option and processing speed is crucial, fixed-point arithmetic can be the solution.

In this paper the differences between fixed and floating-point numbers will be explained. The mathematical operations will be discussed and existing libraries are evaluated. With that information, a new fixed-point implementation is proposed, improving the major drawbacks of the existing ones.

## II. Fixed-Point and Floating-Point Structure

Given the Number of Bits $N$ and the number of fraction bits $P$, the value of a fixed-point number is given by

$$x_{fixed} = \sum_{i=0}^{N-1} b_i \times 2^{i-P}, N \leq P. \tag{1}$$

The structure of a floating-point number is given by a sign bit s, F bits for the fraction part and E bits for the exponent. The value calculated as follows.

$$x_{float} = (-1)^s \left(1 + \sum_{i=1}^{F} b_{F-i} \times 2^{-i}\right) \times 2^{\left(2^{E-1}-1\right)} \tag{2}$$

Usually only the 32 (single precision, 8 bit exponent) and 64 (double-precision, 11 bit exponent) bit floating-point numbers are used [1]. According to equation 1 the minimal step size $\epsilon$ and therefore the precision of the fixed-point format is given by

$$\epsilon = 2^{-P}. \tag{3}$$

For a floating point value the precision is given by the LSB of the fraction part and the exponent. This leads to a high dynamic range and a constant relative error. Comparing a fixed-point value of 32 bits with a Float type, we will end up with 9 bits more for the fraction part and therefore 9 bits more precision. Obviously we well not be able to cover the same range. If the 9 bits are used as integer part and the remaining 23 bits as fraction part, it is possible to express a value $v$ between

$$-2^{(N-P)-1} = -256 \geq v \tag{4}$$

and

$$v > 255.998046875 = 2^{(N-P)} - 2^{-P}. \tag{5}$$

## III. Operations

Fixed-point addition is given by a normal integer addition while a floating-point addition requires a shift of the fraction part and an addition. Also the sign bit has to be evaluated. The fixed-point addition will perform very fast, as this is a built only one assembly command.

Fixed-point multiplication is done by integer multiplication of the double bit size and a shift of the sum of the precisions. Floating-point multiplication is done with an integer multiplication of the fraction parts, an integer addition of the exponents and a XOR of the sign bit. Subtraction and Division are neglected here, as they work in s similar same ways. Even thought fixed-point multiplications seem more extensive, this is not the case. Modern assembly command combine the required calculation steps and reduce therefore the cycles needed.

## IV. Existing Libraries

There exist various free fixed-point libraries for the programming languages C and C++. An example for a C implementation is given by *fixedptc* [2]. It is based on defines for fraction length and inline functions for mathematical operations.

The *libfixedmath* [3] is an open source library for C++. The major drawback is the non-arbitrary fraction point. Only 16.16 and 0.32 fixed-point type are supported so far. This demands for the implementation of a better fixed-point library.

## V. C Implementation

Fixed-point arithmetic in C is straightforward. C does not know the concept of operator overloading. Each mathematical operation will be implemented as function. Listing 1 shows an approach for multiplication using a $define$ statement to represent the number of fraction bits.

```
1 // define base type
2 typedef int FP;
3 // define number of fraction bits
4 #define FP_FRAC_BITS 23
5 // multiplication
6 static inline FP_mul(FP A, FP B)
7 {
8       return (((FP)A * (FP)B) >> FP_FRAC_BITS);
9 }
```

Listing 1. C implementation of fixed-point operations with integer as base type.

The defined fraction size (line 4) makes it impossible to mix up fixed-point numbers with different fraction sizes. Even thought the in-line implementation (line 6) prevents a function call for each appearance of the multiplication, the function-like call is not very handy for mathematical operations. To overcome these drawbacks, we will consider a C++ implementation.

## VI. C++ Implementation

Every calculation that can be done in during compilation will increase the performance of the fixed-point class. This will be the main goal, as embedded systems mostly do not have very performant CPUs. To allow the mathematical operations on fixed-point values with a different amount of fraction bits, the concept of template classes is used [6]. Listing 3 shows the basic structure of the class with the number of fraction bits as the template parameter $PREC$ (line 1), the base type $T$ (line 5) and private container for the value $v$ (line 11).

```
1  template<unsigned PREC> /** Number of fraction bits */
2       class FP /** fixed point class */
3       {
4            public:
5                 typedef int T; /** typdef of base type*/
6                 /** define class as friend to itself for
7                      every precision PP */
8                 template<unsigned PP>
9                      friend class FP;
10           private:
11                T v; /** the fixed point value */
12      }
```

Listing 2. Basic structure of the fixed-point template class.

Note that different values of $PREC$ will lead to different classes. Therefore it is necessary to define the class as a friend of itself. Each class will perfom fast, but will also require space in the program code. As we want to create fixed-point objects out of built-in types, some constructors are added. A default construcor (line 2) will initialize the value to zero. The copy constructor will the copy process of the value $v$. For the built-in types, the constructors are shown on line 6, 8 and 10.

```
1  /** Default constructor */
2  FP():v(0){}
3  /**  Copy constructor */
4  FP(const FP& x):v(x.v){}
5  /** Create from int */
6  FP(int x):v(x*(1<<PREC)){}
7  /** Create from float */
8  FP(float x):v(x*(1<<PREC)){}
9  /** Create from double */
10 FP(double x):v(x*(1<<PREC)){}
```

Listing 3. Example constructors for the fixed-point class. Built in types are supported for constructing a fixed-point object.

The class should also support explicit casts. Additional casts for the built in types have to be created. Lisitng 4 shows the implementation of the cast operators. Line 8 and following implement a cast to a different fixed-point object. This is very fast, as $PREC$ and $PP$ are known at compile time and good compilers will even optimize the if statement.

```
1  /** Cast to int */
2  operator int () const {return (int)(v>>PREC);}
3  /** Cast to float */
4  operator float () const {return ((float)v)/(1<<PREC);}
5  /** Cast to double */
6  operator double () const {return ((double)v)/(1<<PREC);}
7  /** Cast to other FP */
8  template<unsigned PP>
9       operator FP<PP> () const {
10           if (PP>PREC) {return (v>>PP-PREC);}
11           else {return (v<<PREC-PP);}
12           }
```

Listing 4. Cast operators for the fixed-point class. Casting a different fixed-point results in a shift. Shift values are calculated at compare time, which results in maximal processing speed.

With operator overloading a intuitive usage of the class becomes possible. Compared to a C implementation with functions, this concepts leads to the same syntax as for built-in types. Listing 5 shows the implementation of the assignment operator. The built-in types are straightforward. The assignment to a fixed-point value with diffent precision is described on line 18 and following.

```
1  /**  Assignment of int */
2  FP& operator=(const int& x){
3       v = (T)(x*(1<<PREC));
4       return *this;}
5  /** Assignment of float */
6  FP& operator=(const float& x){
7       v = (T)(x*(1<<PREC));
8       return *this;}
9  /** Assignment of double */
10 FP& operator=(const double& x){
11      v = (T)(x*(1<<PREC));
12      return *this;}
13 /** Assignment of FP */
14 FP& operator=(const FP& x){
15      v = x.v;
16      return *this;}
17 /** Assignment of FP with different precision */
18 template<unsigned PP>
19 FP& operator=(const FP<PP>& x){
20      if (PP>PREC) {v = (x.v >> (PP-PREC));}
21      else         {v = (x.v << (PREC-PP));}
22      return *this;}
```

Listing 5. Assignment operator for built-in types and fixed-point class members.

The following listing shows the implementation of the addition. Subtraction is neglected, as it can easily be derived from the implementation below. To add other built-in types, the constructor is used (line 12 and 18 in listing 6).

```
1  /** Addition assignment */
2  FP& operator+=(const FP& x){
3       v += x.v; //check
4       return *this;}
5  /** Addition */
6  FP operator+(const FP& x) const{
7       FP res(*this);
8       return res += x;}
9       /** Addition assignment with other types */
10 template<typename TT>
11 FP& operator+=(const TT& x){
12      this+=FP(x);
13      return (*this);}
14 /** Addition with other types */
15 template<typename TT>
16 FP operator+(const TT& x){
17      FP res(*this);
18      return res+=FP(x);}
```

Listing 6. Addition implemented with operator overloading results in intuitive usage of the fixed-point class.

To apply standart mathematical operations the multiplication and division are missing. Listing 7 shows the implementation of these functions.

```
/** Multiplication assignment  */
FP& operator*=(const FP& x) {
typedef long long int T2;
     v = (T)(((T2)v*(T2)x.v)>>PREC);
     return *this;}
/** Multiplication */
FP operator*(const FP& x) const {
     FP res(*this);
     return res*=x;}
/** Multiplication assignment with other types */
template<typename TT>
FP& operator*=(const TT& x){
     *this*=FP(x);
     return (*this);}
/** Multiplication with other types */
template<typename TT>
FP operator*(const TT& x){
     FP res(*this);
     return res*=FP(x);}
/**  Division assignment */
FP& operator/=(const FP& x)  {
     typedef long long int T2;
     v = (T)((((T2)v)<<PREC)/((T2)(x.v)));
     return *this;}
/** Division */
FP operator/(const FP& x) const {
     FP res(*this);
     return res/=x;}
```

Listing 7. Multiplication and Division implementation of the fixed-point class.

Shift operation is implemented as follows. Note that for simplicity only the shift up case is shown. The other cases can be derived form this.

```
/** Bitwise shift up assignment */
FP& operator<<=(const unsigned x){
     v <<= x;
     return *this;}
/** Bitwise shift up */
FP operator<<(const unsigned x) const{
     FP res(*this);
     return res<<=x;}
```

Listing 8. Bit-wise shift up operation implementation for fixed-point class.

Logical comparison functions are straight forward. Most things can be adopted from the integer base type.

```
/** Smaller than */
bool operator<(const FP& x) const{
     return (v < x.v);}
/** Smaller than with other type */
template<typename TT>
     bool operator<(const TT& x) const{
          return (*this < FP(x));}
```

Listing 9. Logical comparison operation. Only smaller than is implemented, the others can be derived from this.

The following functions allow to shift a fixed-point object to an output stream (E.g. $std :: cout$ in C++). The print function defines the format of the output. This function should not be overused, as it implies a cast to a double type. However it is very useful for debugging purposes. More enhanced function could be realized, but this would go beyond the scope of this paper.

```
/**  Print operator. Shift value to output stream. */
public:
template<typename OUT>
     friend OUT& operator<<(OUT& out,const FP& f){
               return f.print(out);}
private:
/**  Print function. Defines the appearance of the
  * printed fixed-point value. E.g.: 3.5466 <4.28> */
template<typename OUT>
     OUT& print(OUT& out) const{
     return ( out << ((double)v)/(1<<PREC)
                    << " <" << (sizeof(T)*8)-PREC
                    << "." << PREC << ">" );}
```

## VII. Simulation

Simulation with Keil $\mu$Vision and a Freescale Kinetis K10 have shown, that a integer multiplication needs 7 cycles. 4 cycles are used to load the operands and 2 cycles to store the result. This makes sense, as a ARM CPU can perform the actual multiplication in 1 cycle. The integer addition needs the same number of cycles. Even if the full 64 bit result is needed, this only takes one cycle, as this is a supported assembly command. However, a floating-point multiplication takes 47 cycles.

## VIII. Conclusion

For small micro controllers and embedded systems it is reasonable to consider the use of fixed-point data types over the use of floating-point data types. When dynamic range is not needed, we can achieve better results for precision and faster processing time with minimized cost. Especially if the additional cost for a floating-point is a fundamental criteria.

Although the same advantages apply for a C implementation, the concept of object oriented programming as in C++ allows a design which is similar to the built-in types as Integer and Float. It can therefore be changed to the fixed-point class by a simple *typedef*.

With the use of template classes and functions, as well as operator overloading, a multi-purpose fixed-point class for embedded system use can be developed. Implementation of standard mathematical operation and type casting lead to a full functional and intuitive usable class.

As template calsses create program code for every different template parameter, program code will increase in size. However, normally we will not use a lot of differnet fixed-point types in an embedded system, and the increase in used memeory can be neglected.


## IX. Acknowledgements

I would like to thank Hans Buchmann for his continuing support and advise, and Hans-Peter Schmid and Alex Huber for their suggestions on this paper. Moreover i like to thank the Team of the Institute of Microelectronics of the University of Applied Sciences for the interesting discussions on the content of this paper.


## References


[1] *IEEE Standard for Floating-PointArithmetic*, IEEE Computer Society, 2008.

[2] “Fixed point math library for c,” http://sourceforge.net/projects/fixedptc/.

[3] “libfixmath - cross platform fixed point maths library,” http://code.google.com/p/libfixmath/.

[4] D. Goldberg, “What every computer scientist should know about floating-point arithmetic,” 1991.

[5] R. Yates, “Fixed-point arithmetic: An introduction,” 2001.

[6] B. E. M. Stanley B. Lippman, Jose Lajoie, *C++ Primer - Schneller und effizienter Programmieren lernen*. Addison-Wesley, 2005.