Floating Point Processing

Calculate Long Hand

In this article, Steven Goodwin follows on from the last [1] to look at fixed-point processing and how it can be used to improve performance in PDA-based applications, how it differs from floating-point, along with the issues involved in porting software from one to the other and how to overcome them.

BY STEVEN GOODWIN

The reason we were originally using floating-point numbers within our programs, was for its mathematically qualities. Now, without the standard maths library to support us, we must resort to coding the necessary portions of the code for ourselves. Or at least we should Google for the appropriate algorithms and copy them into our code!

Add N to X

Fortunately, this is not as daunting a prospect as it sounds, since a lot of it is very straight forward, and a very well understood problem in computer science. Let us first consider the basic math functions: addition, subtraction, multiplication and division. Since fixed-point numbers are stored as integers, a fixed-point operation works like any other integral operation, and so requires no new code.

```
#define MTH_ADD(__a, __b) ➚
( (__a) + (__b) )
#define MTH_SUB(__a, __b) ➚
( (__a) - (__b) )
```

Using macros in our code might appear unnecessary. However, by doing so, we can replace them with function calls later on that will check for overflow, report usage, and assist with debugging.

Even multiplication has only one caveat, and that is for big numbers. It is possible (and not too difficult) to overflow the integral part of a fixed-point number during the calculation – even though the final result might fit neatly into 32 bits. So, to prevent this, we must temporarily typecast our (32 bit) type into something larger to maintain the precision. Then, once we have the result, we can convert it back to FIXP, losing only the precision of the least significant fractional part.

```
#define MTH_MUL(__a, __b) ➚
(FIXP)( ((long long)(__a) * ➚
(__b)) >> FIX_FRACBITS )
```

To ease code maintenance, the 'long long' type should be afforded its own type, say FIXP_BIG. In this way, an 8.8 fixed-point system (using short int's) can use a simple 'long' for FIXP_BIG, which is a more manageable (and usually faster) type.

Lay All Your Love on Me

You may notice I use macro names prefixed with MTH_ instead of the usual FIX_ here. This layer of abstraction will permit us to re-compile the application for floating-point operation at a later stage by simple changing the macro to:

```
#define MTH_MUL(__a, __b) ➚
( (__a) * (__b) )
```

We can switch between fixed and floating-point in the same manner as we turn the debug information on and off. That is, compile with a -DFIXED_POINT argument to gcc, and use code such as:

```
#ifdef FIXED_POINT
#define MTH_MUL(__a, __b) ➚
(FIXP)( ((FIXP_BIG)(__a) * ➚
(__b)) >> FIX_FRACBITS )
#else
#define MTH_MUL(__a, __b) ➚
( (__a) * (__b) )
#endif
```

This will keep the code base identical for both versions, requiring minimal changes – and therefore maintenance – over time. This is especially useful later on in the development cycle when new features are added and something goes wrong. We can switch back to our floating-point implementation to determine whether it is our algorithm, or fixed point code that is wrong.

Love Spreads

Multiplication isn't the only operation that requires our FIXP_BIG type. Division does too. Both of these operations work by the same principles as fixed point itself. That is, a fixed-point number acts like a floating point number that has been pre-multiplied by a constant factor, say 100000. All numbers are then 100000 times bigger than they should be – including fractions; 14.5 becomes 1450000, for example. Now if you divided this number by 100000 you'd get the answer 14 – the fractional part (0.5) would get lost because it's an integer calculation.

So to preserve the fractional component (the 0.5 which was represented with a value less than 100000) we make this number 100000 bigger _again_ (to 145000000000!), which will maintain the fractional component after a division. Naturally, multiplying these larger numbers requires a larger type, and therefore we need a FIXP_BIG type. If we were using a system where such a type was unavailable, then specific multiplication and division algorithms would have to be written. We shall not cover them here as such cases are rare, and the methods are easy to deduce.

```
FIXP fixDiv(FIXP num, ➘
FIXP divisor)
{
FIXP_BIG tmp = (FIXP_BIG)num;

 tmp <<= FIX_INTBITS+➘
FIX_FRACBITS;
 tmp /= divisor;
 tmp >>= FIX_INTBITS;
 return (FIXP)tmp;
}
```

Comparisons like greater than and less than will work directly with the > and < symbols. Including negative numbers! And since the numbers already exist in fixed-point form at this time, there is no need to wrapper them into macros because any overflow would have already happened, and been trapped.

Constant Craving

Depending on your application, you may need to make use of mathematical constants like Pi and e. Naturally, the maths library will not contain them in your specific fixed-point format, so you will have to create them yourself.

```
pi = FLOAT_TO_FIXED➘
(3.14159265358979f);
```

Or we could use the definitions of M_PI and M_E provided for us in /usr/include/math.h, as they contain more precision.

```
pi = FLOAT_TO_FIXED(M_PI);
```

This is a good example where a 'fixInitialiseLibrary' function comes in useful. When the program starts, this initialisation routine can prepare a set of constants, such as pi and e, which the code can then reference directly, without any additional processor overhead. See Listing 1.

Again, we've used the MTH_ naming convention to allow a floating point version of the code to be built with minimal effort. We create a special constant for 2pi (instead of using g_MthConstant_PI*2) because by doing so we can regain an extra decimal place of precision.

Some fixed-point library code will define values of pi directly into the source without an initialize function. This is fine, however, such code is also fixed (pardon the pun) to a specific format, such as 16.16.

Where it's at

The initialize function is also very useful for our next category – functions. One of the larger sections of the maths library are the trigonometry functions like sine, cosine and tangent. These are normally calculated with mathematical techniques like the Taylor series. To do these accurately in fixed point maths can be a lot of work, and would create a large performance bottleneck.

To save the processing, we don't calculate them! Since the sine function (along with cosine) is periodic (that is, it repeats itself at specific intervals) we can create a simple look-up table for that interval and reference it without any significant overhead. Look up tables can also be used for arc-sine and arc-cosine which also fall within a specific range.

fixInitialiseLibrary can create a table with, say, 1024 elements using the (slow) floating-point arithmetic functions and use it throughout the rest of the program. See Listing 2.

The size of this table will be governed by you – according to the amount of memory you wish to devote to it, and the accuracy required by your application, but 1024 should suffice for most applications. This table can be re-used for the cosine function since they produce the

Listing 1: Setting constants

```
/* fixed.c */
FIXP g_MthConstant_PI, g_MthConstant_2PI, g_MthConstant_E;

void fixInitialiseLibrary(void)
{
   g_MthConstant_PI  = FLOAT_TO_FIXED(3.1415926535f);
   g_MthConstant_2PI = FLOAT_TO_FIXED(6.2831853071f);
   g_MthConstant_E   = FLOAT_TO_FIXED(2.7182818285f);
}

/* fixed.h */
#define MTH_PI          g_MthConstant_PI
#define MTH_2PI         g_MthConstant_2PI
#define MTH_E           g_MthConstant_E

extern FIXP g_MthConstant_PI, g_MthConstant_2PI, g_MthConstant_E;
```

same wave pattern, albeit with a phase difference of pi/2 radians.

```
FIXP fixCos(FIXP theta)
{
 return fixSin(theta + MTH_PI/2);
}
```

This look-up table can be further improved by interpolating any values not present in the table. Or, if you have a mistrust of tables, it can be calculated fairly efficiently using an *approximation* of a Taylor series or the Remez algorithm.

A good maths textbook will provide you will methods to compute other functions, such as tangents.

Silent All These Years

One set of functions that can not be represented with a simple table (like sine) are non-cyclic functions (like cubics) and those with input that does not fall within confined boundaries, like the square root. A table for these functions would be impractical, and so the full algorithm or, at least, an approximation of it must be created. Before implementing the algorithm (which costs development and processor time), consider the following questions:

- Is it necessary? A function that finds the distance between two points may use Pythagorus; but if it's only to find which of two points are closer then the root function becomes unnecessary.
- Do they need to be accurate? Not as silly as it sounds. A lot of file formats are lossy (JPG, MP3), and there is no reason why processing can't be as well. In the example of a distance between two points, a good approximation can be found by adding the length of the longer side to 1/4 of the shorter.
- Can the algorithm be re-written as to avoid the function, or use a different one with less processing overhead? If so, do it!

Cost

Some companies, such as Ekkla Research can provide a fixed-point library that your code can link to in place of the floating-point GNU library, reducing your workload to nothing more than a re-compile. However, this library (along with a number of others) costs money, so consider the 'time vs money' question carefully before starting out.

If you find it is necessary to implement a fixed-point function for (say) the square root function then you should prefer a scalable algorithm. Most algorithms in mathematics work on a method of iteration: the same operation is performed over and over again until the error is small enough to be insignificant. By implementing such an algorithm you can reduce the number of iterations to suit your processor budget. In the example below for the square root, you might like to experiment with the magic number '8'! You'll be able to see how much extra precision can be gained from the function, how much longer it takes to process, and make a judgement call on an optimal trade off for your application. See Listing 3.

Examples of these type of algorithm can be found at [2]. More complex algorithms for functions such as natural logarithms can be found in Knuth [3].

Smoking in the Boys Room

Finally, there are a few odd functions that you'll find yourself needing. Or wanting. Their implementation is generally very easy as they always use well known algorithms. As we've said before, implement only what you need. See Listing 4.

This example demonstrates the most common 'gotcha' in math library programming: negative numbers are slightly different to positive numbers, especially when it comes to truncation or rounding. As general purpose library code you'll have to cope with these.

In addition, you might also want to implement a reciprocal function (since it's quicker than performing 1/N), and a custom 'fixPrint' routine, which doesn't

Listing 2: Fixed.c

```
/* fixed.c - compile with -lm option to link in maths library for sin() fn */
#include "math.h"

static FIXP g_SineTable[1024];

void fixInitialiseLibrary(void)
{
   ...

   for(i=0;i<1024;i++)
      {
      fSineValue = sin(M_PI * 2.0f * i / 1024.0f);
      g_SineTable[i] = FLOAT_TO_FIXED(fSineValue);
      }

}

FIXP fixSin(FIXP theta)
{
/* We need to scale theta from 0 to 2pi to 0 to 1023 */
/* i.e. offset = theta/2pi * 1024 */

/* Compute the scale as a float */
float scale = 1024.0f / (2.0f * M_PI);

/* Convert to our fixed-point notation */
FIXP fixed_scale = FLOAT_TO_FIXED(scale);

/* Find our genuine offset */
int offset = MTH_MUL(theta, fixed_scale);

/* Convert out fixed-point number into a 'normal' int */
offset >>= FIX_FRACBITS;

/* Sine is a cyclic function, so make sure an offset of 1024
   maps to 0, 1025 map to 1, and so on.
   (A table of 1024 elements was chosen to make masking easy) */
offset = offset & 1023;

/* Give our result back to the world */
return g_SineTable[offset];
}
```

LPI Anzeige

have the side effects of lost precision caused by the float conversion. When writing and testing your code you will doubtless discover more.

Fast Car

After porting your code to the new (super fast!) fixed-point notation, it is possible you will _still_ need more speed. In addition to the usual optimization techniques we should also look at those specific to fixed-point.

The first consideration is to re-order expressions (such as a*b/c) so that numeric constants are grouped together. Consider the offset calculation when doing a lookup to the sine table. If this were the slowest part of our application, and the guilty expression was,

```
n/2pi * 1024
```

We could re-write this in the form,

```
n * (1024/2pi)
```

Then, when the constant value of 1024/2pi has been computed, this constant can be used in place of the expression and produce a significant improvement in speed over the original. This constant can then be placed in fixInitialiseLibrary.

This is true of anything requiring division. Dividing, whether fixed-point or floating point, is slow. So, wherever possible, replace a 'divide by N' operation with a 'multiply by 1/N'. The value of 1/N should be pre-computed as above to provide an extra speed up, possibly by using a special reciprocal function, as we saw in the previous section.

Listing 3: Fixed point Square root function

```
FIXP fixSqrt(FIXP num)
{
FIXP Unity = 1<<FIX_FRACBITS;
FIXP tmp = num + Unity;
int i;

 tmp >>= 1;       /* Divide by two */

 for(i=0;i<8;i++)
  {
  tmp = tmp+fixDiv(num,tmp);
  tmp >>= 1;
  }

 return tmp;
}
```

I would also suggest the consideration of specific cases. If the speed of the algorithm relies on a 'division by 2', and it now reads 'multiply by 0.5', then consider a specific piece of code to manage the multiply by 0.5. In this case we use the bit shift trick we saw in the square root function.

```
iHalfOfN = N >> 1;
```

As long as N is an integer then, regardless of fixed-point format used (12.20 or 16.16), this operation will halve its value in virtually no time at all. The same is true when dividing (or multiplying) by any multiple of 2 (i.e. 2,4,8,16,32,64…). Although this trick does not work with *floating* point arithmetic it doesn't matter – we're working in *fixed*-point – so we need to use solutions that are better suited to it. Even if they appear a little off the wall. The implementation of fixRound above features another such case.

You can also apply a similar trick with other constants by borrowing the idea of pre-computing the constant. A 'multiply by 3' operation can be optimized by evaluating the fixed-point value of 3 (with FLOAT_TO_FIXED(3) for example) and then multiplied directly with the variable.

```
iThreeTimesBigger = fixed_point_number * to_fixed3;
```

And finally, there's always the possibility of writing some hand-tuned assembler code. Since the aim of a fixed-point project is speed (and often tied to a piece of hardware), it is acceptable to create a macro like USE_STRONGARM_ASM, and surround specialist assembler code with it.

Finally, a word of warning concerning the FLOAT_TO_FIXED macro. Some processors have a large penalty attached to the conversion between integral and floating-point numbers, in either direction. If you are repeating such conversions in tight loops, then I strongly recommend moving them outside, and calculating then once, as we did for the maths constants above. If in doubt, pre-compute.

Only You

We have covered a lot of ground here, discovering techniques and tricks that will allow a good port of hi-performance software to small platforms where the 'one size fits all' mentality doesn't apply. Sample code from this article can be found at [4], in the sources directory. ■

Listing 4: Odd functions

```
FIXP fixRound(FIXP num)
{
FIXP round, trunc;

        if (num & (1<<(FIX_FRACBITS-1)))
                round = 1<<FIX_FRACBITS;
        else
                round = 0;

        if (num > 0)
                {
                trunc = num & ~((1<<FIX_FRACBITS)-1);
                return trunc + round;
                }
        else
                {
                trunc = (-num) & ~((1<<FIX_FRACBITS)-1);
                return -trunc + round;
                }
}
```

INFO

[1] Floating Point Processing – Numbers everywhere! Linux Magazine, Issue 38 – January 2004, p65

[2] Example algorithms: *http://www.ai.mit.edu/people/hqm/imode/fplib/cordic_code.html*

[3] Donald Knuth's, "The Art of Computer Programming"

[4] Article sample code: *http://www.bluedust.com/pub*