Floating Point Processing

Numbers everywhere!

In this article, Steven Goodwin looks 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. A reasonable understanding of C and mathematical theory is required in order to follow the examples.

BY STEVEN GOODWIN

A lot of algorithms (especially those in science and multimedia) require the use of floating-point numbers. That is, numbers featuring a decimal point, like 9.2 or 0.65. We humans use them as if they were any other number: we perform arithmetic in the same way as we would for the numbers 92 or 65 – we just move the decimal point to the right place afterwards.

For a computer, this level of calculation is quite complicated, and in the Intel family, CPUs from the i486 onwards have included a *floating-point unit* (or FPU) to exclusively perform such calculations very quickly. The FPU unfortunately, consumes a lot of power, and as a result is not found on a number of mobile processors, such as those often found in PDAs.

Although the absence of an FPU is not usually a problem for text editors and HTML clients; MP3/Ogg players and graphic applications (especially those in 3D) make extensive use of its features, slowing our once nifty Linux PDA to a crawl because because the processor has to perform these calculations manually by using *floating-point emulation*.

Instead of letting the processor perform emulation, it is possible to rewrite portions of each algorithm to use *fixed-point* numbers. These look (to the processor) like normal integers (i.e. without a decimal point) and so no emulation is needed because they are processed directly. This in turn should put the whiz back into our whiz-bang application.

Here, There and Everywhere

When performing arithmetic on paper we will arbitrarily move the decimal point around according to the demands of the problem. Sometimes there are more numbers after the decimal point, and sometimes there are more preceding it. We do this automatically – and as soon as we reach the edge of the paper, we shuffle the numbers along and continue working. The decimal point therefore appears to 'float' to wherever it is needed. This model for arithmetic is called *floating-point*.

```
float fValue = 14.5f;
/* Declaring a floating-point
 variable in C */
```

Internally, each floating-point number is stored in what is known as IEEE 754 format. For single precision (32 bit) floating-point numbers, this format is detailed as in Table 1.

The two halves (E & M) of the number do not relate to values before and after the decimal point. The mantissa (M) refers to the entire number (with decimal points removed), and the number of decimal places it needs to be moved by (and

Box 1: Showing 14.5 in binary

```
      1                 1 1 1  1  1  1
. . . 2 6 3 1           - - - -- -- --   . . .
      8 4 2 6 8 4 2 1 . 2 4 8 16 32 64
      -------------------------------
. . . 0 0 0 0 1 1 1 0 . 1 0 0 0  0  0    . . .
```

in which direction) to re-create the original number is called the exponent (E). The number can then be derived as if it were written in scientific notation:

```
N = -1<+>S<+> * 1.M * B<+>E<+>
```

Where S is the sign, M is the (normalized) mantissa, B is the base (or radix) and E is the exponent. So for example:

```
Sign: 0 (positive number)
Exponent: -1 (divide by ten)
Mantissa: 145

N = 145 * 10-1  = 14.5
```

This is only a broad overview of the format, however. The IEEE standard explores a number of additional concepts (such as normalization and bias), and introduces special bit patterns to represent numbers such as infinity, and NaNs (Not a Number)! If you wish to study floating point in more depth, then please refer to one of a number of good articles [1] and [2] on the web that explore this in (excruciating) detail!

Floating-point provides an incredible range of numbers (up to 3.402823… * 10^{38}), but with the drawback of limited precision – that is, not every distinct number can be represented exactly. This can either be a mild nuisance, or the biggest pain you've ever had, depending on your application.

If you're measuring the solar system, adding the width of atom will make no difference to your calculations – floating-point representation will not be able to resolve the difference in magnitude resulting in an unchanged solar system. However, if you are working at the level of atoms, then adding another atom *will* work, because, relatively speaking, their exponents are very close.

One of the annoying issues with floating-point occurs when a fairly small number, like 2.8, can not be shown exactly, and is rendered on screen as 2.7999999999999. This occurs because although 1.4 may be exact in base 10 (that we work in), it isn't in base 2 (which the computer does). Refer to the IEEE standard below to see how 1.4 would be represented internally, in order to demonstrate this fact.

If floating-point is not enough, however, there is always the possibility of using double precision numbers or double-extended precision.

As you can imagine, arithmetic on two numbers in this format requires a lot more work than not using it. The FPU has to consider different size exponents, overflowing mantissas, both sign bits and the evaluation of a new exponent – in addition to the actual arithmetic that we are trying to handle!

Stuck in the middle

Fixed-point, by contrast, works by specifying a consistent number of bits before and after the decimal point across the whole range of numbers. The integral part will always be (say) 12 bits, and the fractional part will always have 20 (see Box 2: Banana Splits). This limits the *range* of integers available, but benefits from an increased precision, since every single number in this range can be represented.

The number 14.5 could be shown thus: 14 in the 12 bit binary for the mantissa is 000000001110 For the fractional part, the .5 in 20 bit binary is 10000000000000000000. And so, the complete number would encode as 00000000111010000000000000000000 in binary format! Which, if displayed as an integer (which it is), shows up as 15204352!

Reading fixed-point binary fractions is no more difficult than reading binary integers. Simply extend the 'powers of two' to other side of the decimal point, as shown Box 1, and you're there!

Calculations can then be performed using integer calculations, removing the need for an FPU. You will have noticed that the .5 component is stored as 2^{19} – half of the number 2^{20} – so that 0.5 + 0.5 will overflow into the 12 integral bits automatically and create the number 1.0.

We have not explicitly given ourselves a sign bit here. We do not need to, since we shall use the integral component as a 2's complement binary number (similar to normal integers) as this is more natural for the CPU to handle, saving yet more of our valuable processor time. The range of values for our 12.20 split system

Table 1:IEEE 754 format

Bit:	31	30-23	22-0
Description:	Sign bit	Exponent	Mantissa
	s	eeeeeeee	mmmmmmmmmmmmmmmmmmmmmmm

Box 2: Banana Splits

The 12.20 split is arbitrary since all fixed-point operations work naturally with integral operations. You can decide where the split should occur yourself, by studying the problem domain, and considering your hardware.

When looking at your problem, determine within which range the numbers will fall. If you are doing signal processing with sine waves, then most values will fall between -1 and +1. So, assuming we'll sum no more than 16 of these sine waves together, a range of +/- 16 would not be unreasonable, so we can adopt a 6.26 approach. This actually gives a higher level of accuracy than general purpose floating-point work!

As for the question of hardware, I'm using a 32 bit (i86) machine as a common, easy-to-understand system, so it makes sense to utilize the full 32 bits to represent our fixed-point numbers. However, a much smaller embedded system might only have 16 bits, so we would have to tailor our code to that. Since fixed point is usually only used for improving speed (and sometimes memory usage), it may be tied to an architecture more closely than other software, and so it is not unreasonable to make machine-specific decisions. As always, these should be clearly documented and labelled accordingly.

lies between -2048 and +2047 (-2^{11} to $2^{11}-1$).

Fixed-point mode may appear rather limiting for general applications – and it is! But because we are developing a solution for a specific problem, we accept the limitations of the system, and utilize the benefits of speed and accuracy that it gives us.

Hold Me Now

Fixed-point numbers are stored as a single integral type, usually an *int* or a *long*. A greater range can be achieved by using the 'long long' data type, or two variables – this is at the expense of extra processing in order to perform the carry from one variable to the other. Since our purpose in this case is to *reduce* the processing required, this method buys us very little benefit.

Get This Party Started

Whether you are writing a brand new fixed-point routine, or retro-fitting the code to an existing application, you need a solid base to work from. This base will invariably be a fixed.h header file, containing all the necessary macros and function prototypes for your code. A supplementary fixed.c should exist too, providing your function implementations. Although the final code build will most probably use macros exclusively (for speed purposes), a C source file makes the development process (including debugging) much easier. It can also include functions to create (and destroy) temporary workspaces that the fixed-point library may need during its life. We will see examples of these later.

There are a number of different components to a library of this kind. For example:

- Conversion routines, between fixed and floating-point in both directions
- Basic operators (addition, subtraction, multiplication and division)
- Mathematical constants (such a pi and e)
- Mathematical functions (like sine and cosine)
- Miscellaneous helper functions (like rounding)

This may appear like a very large body of work. However, since we will only be converting small parts of our application, we only need to write (or integrate from another source) a small portion of these components. Even operations like division that we take for granted in real life, may be completely unnecessary to us!

Our primary header file should contain all other implementation details for our code. It would, for example, be a good idea to avoid the magic numbers of 12 and 20 when describing the number of bits used in our notation since they might change later. We should also specify the data type for our numbers also, to make sure we have enough bits for both fractional and integral parts.

```
typedef int FIXP;
/* This must be 32 bits */

#define FIX_FRACBITS 20
#define FIX_INTBITS 12
```

We have adopted the style of prefixing our macros with 'FIX_'. We have also chosen to label each component with explicit numbers (12 & 20), although in practice the size of the integral component can be deduced thus:

```
#define FIX_INTBITS ((sizeof(↩
FIXP)*8)-FIX_FRACBITS)
```

Either method is acceptable. The compiler will optimize this macro during the build, so (on the very few occasions it is used) it will not cause any slow down. We are also considering the sign bit to be part of FIX_INTBITS.

One of biggest practical considerations in software (of all types) is that of test-

Box 3: C++

C++ programmers have a language feature known as operator overloading. This allows you to change the way the basic operators (like addition, subtraction and so forth) work, when applied to classes. It would therefore be possible to create a class called 'CFixedPoint' which removed the need for macros and made the fixed-point elements of the code transparent, giving greater readability to the algorithm.

If your language supports such functionality you should always play to its strengths, and consider such an implementation.

Java and the real world

Throughout this article, examples are in 'C' and given as functions for clarity. Java programmers may care to read similar code at [3]. For a complete, albeit minimal, real-world example, the fixed-point Ogg Vorbis decoder (that was the inspiration for this article) can be found at [4]. My thanks to Richard Cohen for sourcing it!

Listing 1: Three checks

```
01 FIXP fixFloatToFixed(float fValue)
02 {
03 FIXP Unity, Result;
04
05    if ((int)fValue >= (1<<(FIX_INTBITS-1)))         /* -1 because of
   sign bit */
06        fprintf(stderr, "FIXP: Integral overflow of %f\n", fValue);
07
08    if ((int)fValue < -(1<<(FIX_INTBITS-1)))         /* -1 because of
   sign bit */
09        fprintf(stderr, "FIXP: Negative integral overflow of %f\n",
   fValue);
10
11    if (fValue < 1.0f/(1<<FIX_FRACBITS) && fValue > -
   1.0f/(1<<FIX_FRACBITS) && fValue != 0)
12        fprintf(stderr, "FIXP: Fractional underflow of %f\n", fValue);
13
14    Unity = 1<<FIX_FRACBITS;
15    Result = (FIXP) (Unity * fValue);
16
17    return Result;
18 }
```

ing. How will we test this? Can we check for errors easily? And so on.

In fixed-point work, choosing the bias between integral and fractional bits can be a little tricky, since it is possible for the numbers to go out of our set range easily, and for errors to be accumulated without any warning. Since the effort of checking the value of each number at every step of the process, to ensure that this does not happen, would eradicate any speed gain we might otherwise have made, it is a very good idea to use a macro that can represent the algorithm *in situ*, or to call a separate function.

This can be done on GCC by compiling with a -DDEBUG flag, and then switching between the two variations of code with:

```
#ifdef DEBUG
#define FLOAT_TO_FIXED(__f) fixFloatToFixed(_f)
#else
#define FLOAT_TO_FIXED(__f) (FIXP)((__f)*(1<<FIX_FRACBITS))
#endif
```

The 'fixFloatToFixed' function can perform additional work like checking the range of both components, outputting errors, or even hold statistics for the most number of integral or fractional bits used; something that would be unwieldling in an macro.

Listing 2: Back to floating

```
float fixFixedToFloat(FIXP iValue)
{
float Result;
float fIntegral, fFraction;

   fIntegral  = (float)(iValue>>FIX_FRACBITS);

   fFraction  = (float)(iValue & ((1<<FIX_FRACBITS)-1)); /* isolate fraction */
   fFraction /= (float)(1<<FIX_FRACBITS);

   Result = fIntegral + fFraction;

   return Result;
}
```

With these basic ideas, we can start to build our arsenal.

One to Another

Naturally we will want at least two functions into this category – float to fixed, and fixed to float. We can however, add others, like 'two ints' to fixed. However, we should follow the ideals of the lazy programmer, and only implement what we need to do the job.

We need to make three checks for converting numbers into fixed point. This is,

1. Make sure they're not too big (e.g. greater than 2047)
2. Make sure they're not too small (e.g. less than -2048)
3. Make sure the fractional part can be represented using the bits we've got available

The code in Listing 1 demonstrates all three.

When converting back into floating point, as in Listing 2, we will not be checking the range or precision. It is possible (but quite tricky) to do so, and doesn't buy us anything important.

The macro-ized version of this last function can be written without all the (explicit) castings like this:

```
#define FIXED_TO_FLOAT(__i) ( ((__i)>>FIXP_FRACBITS) + (float)(((__i)&((1<<FIXP_FRACBITS)-1))/(1<<FIXP_FRACBITS))
```

Since both the integral and fractional component of the number are combined into one variable, it is inevitable that there will some bit-wise logic to accompany it. The logic itself is not problematic – there's just a lot of it, which when compressed into macros makes it slightly more difficult to follow. However, it all boils down to a couple of variations on the same basic theme of bit shifts and masks (see Box 4: Bit shifts).

We can then test these conversion functions with a few simple lines of C, such as:

```
FIXP  fixed;
float not_fixed;
 fixed = fixFloatToFixed(14.5f);
 printf("14.5 in decimal = %d?\n", fixed);
 not_fixed = fixFixedToFloat(fixed);
 printf("Does 14.5 = %f?\n", not_fixed);
```

Bend me, Shape Me

All of the code so far makes the assumption that every number in our fixed-point system will be in the same 12.20 format. This can be true. However, some software may require two (or more) different formats within the same program. It is impractical to supply two headers with different names, just to support the extra precision in specific, critical, areas. Instead, work with one format as your default, and provide general-purpose macros which can be used by your library, and the main user program.

```
#define FLOAT_TO_FIXED_GEN(__f, __fracbits)  (FIXP)((__f)*(1<<(__fracbits)))
#define FLOAT_TO_FIXED(__f) FLOAT_TO_FIXED_GEN(__f, FIX_FRACBITS)
```

Now we've seen how to create, handle, and output fixed-point numbers. Next month we'll look at how to handle them in the more complex area of mathematics. ■

Box 4: Bit shifts

The most common shifting operation you'll come across is of the '1<<n' variety (where, in our examples, n=20). This is used in two main places. The first is to represent the value of 1.0 in your fixed-point notation, which can then be used with a single multiplication for float->fixed conversions.

Secondly, when used in the expression (1<<n)-1 you get a result where every bit in the fractional part of the number is set to 1. This makes it an excellent bit mask to isolate the fraction, as seen in the fixRound function.

INFO

[1] Standard for Binary Floating-Point Arithmetic: *http://grouper.ieee.org/groups/754*

[2] IEEE Floating Point Standard: *http://www.tutorgig.com/encyclopedia/getdefn.jsp?query=IEEE_754*

[3] Java versions: *http://www.ai.mit.edu/people/hqm/imode/fplib/FP.java*

[4] Ogg Vorbis decoder: *http://lorien.handhelds.org/ftp.arm.linux.org.uk/people/nico/vorbis*