Fixed Point Math with AVR-GCC

Paul R. Potts

October 2016

Embedded C

I recently had reason to do some non-integer math on a small microcontroller, a member of the Atmel ATtiny series. There is no floating-point hardware. It is possible to do some floating-point math using library functions, but they will certainly eat up an enormous amount of the available program space, and will not be speedy. So there is an incentive to avoid using floating-point math altogether, if possible.

So, I began looking into fixed-point math. It is always possible to roll your own code for this kind of thing, but I thought I would see if I could take advantage of existing, debugged library code first. I found some free software libraries online, but it was not very clear how to use them or whether they would fit on the ATtiny chips.

I discovered that there is, in fact, a standard for fixed-point types in C. It has not been widely adopted. Like the C standard itself it is a little loose in parts, in that it doesn't dictate the numeric limits of types, but rather specifies a range of acceptable sizes. And it turns out that my toolchain supports this standard, at least in part.

I won't try to describe everything covered in the Embedded C document. I'll spare you my struggle trying to find adequate documentation on how to do certain things in an implementation that doesn't implement everything in Embedded C. Instead I will try to do something more modest, and just explain how I managed to use a couple of fixed-point types to meet my specific needs.

You can find more information on the Embedded C standard here.

The actual Embedded C standards document in PDF form can be found here (note: PDF link).

At the time of this writing, this seems to be the latest version available, dated April 4, 2006. The document indicates a copyright, but unlike the C and C++ standards, it looks like you can download it for no cost, at least at present.

avr-gcc

The compiler I'm using is **avr-gcc**. My actual toolchain for this project is Atmel Studio version 7.0.1006. Atmel Studio is available for download at no cost. The **avr-gcc** toolchain that Atmel Studio uses under the hood is available in other toolchains and as source code. I'm not going to try to document all the ways you can get it, but you can find out more here.

As I understand it, these Embedded C extensions are not generally available in other versions of GCC.

The Basics of Fixed Point Types in Embedded C

I'm assuming I don't have to go into too much detail about what fixed-point math is. To put it briefly, fixed point types are like signed or unsigned integral types except there is an implicit *binary point* (not a decimal point, a binary point). To the left of that binary point, the bits indicate ascending powers of two, from 20 up: 1, 2, 4, 8, etc. To the right of that binary point, the bits indicate fractional powers of two, from 2-1 up: 1/2, 1/4, 1/8, etc.

The Embedded C extensions for fixed-point math came about, I believe, because many microcontrollers and digital signal processors have hardware support for fixed-point math. I've used DSPs from Motorola and Texas Instruments that offered accumulators for fixed-point math in special wide sizes, such as 56 bits, and also offered saturation arithmetic. Using these registers from C required special vendor-specific compiler support. The idea was that if programmers had a standardized way to access this functionality, using Embedded C types, they would have a better shot at writing portable code; on platforms without hardware support, fixed-point math could be implemented using relatively simple library functions.

There are a couple of basic approaches to these types mentioned in the standard. There are *fractional* types, indicated with the keyword **Fract**, with values between -1.0 and 1.0, and types that have an integral part and a fractional part, indicated with the keyword **Acum**. It is expected that implementations will give these aliases, like **fract** and **accum**, but I think the authors did not want to introduce potential name clashes with existing code.

The standard specifies the *minimal* formats for a number of types. For example, **unsigned long accum** provides a minimum of 4 integer bits and 23 fractional bits. On the platform I'm using, **unsigned long accum** actually provides 32 integral bits and 32 fractional bits, aka "32.32." It maps to an underlying type that can hold the 64 bits. In this case, that underlying type is **unsigned long long**.

Accumulator Types

For my algorithms, I don't have much interest in the **Fract** types, and I'm going to use only the **Accum** types. I would have more interest in **Fract** types if there were standard ways available to multiply an **Accum** by a **Fract**. If that existed, I could use a **Fract** type as a scale factor to apply to a large-ish integer in **Accum** representation. For example, let's say I want to generate an unsigned binary value to send to a DAC that accepts 18-bit values. I could create a value of an **Accum** type that represents the largest 18-bit value, and scale this by a **Fract** value indicating the fraction to apply. The advantage of this approach would be, I thought, that I would use types that were only as wide as I needed, resulting in less code. However, since this does not seem easy or convenient to do, in my own code I am using only **Accum** types at present. In fact, I'm using only unsigned **Accum** types, specifically, the aforementioned **unsigned long accum**, but also **unsigned accum**, which gives me a 16-bit unsigned integer value and a 16-bit fractional value (aka "16.16") whose underlying type is **unsigned long** (32 bits).

Fixed Point Constants

There are new suffixes to allow specifying fixed-point constants. For example, instead of specifying **15UL** (for **unsigned long**), one can write **15UK** for an **unsigned accum** type, or **15ULK** for an **unsigned long accum** type. One can specify both the integer and fractional part, for example **1.5UK**. On this platform, **1.5UK** assigned to a variable of **unsigned accum** type will produce the 16.16 bit pattern **0000 0000 0000 0001 1000 0000 0000 0000** (hex **00018000**), where the most significant 16 bits represent the integer part, 1, and the least significant 16 bits represent the fractional part, 0.5.

Accuracy

For our purposes we will mostly be using the integer results of fixed-point calculations. We don't need to use the **FX_FULL_PRECISION** pragma; error of 2 ULPs for multiplication and division operations is fine.

A Very Simple Example

Here's a small program that shows a very simple calculation using **unsigned accum** types. I created a simple project in Atmel Studio that targets the ATtiny 841 microcontroller, which has 512 bytes of SRAM and 8 KiB of flash memory for programs. Today I'm not actually connecting the debugger to hardware. It is possible to configure the project's "Tool" settings to use a simulator, instead of a hardware debugger or programmer.

```
#include <avr/io.h>
#include <stdfix.h>
```

```
static unsigned accum sixteen_sixteen_half = 0.5UK;
static unsigned accum sixteen_sixteen_quarter = 0.25UK;
static unsigned accum sixteen_sixteen_scaled;

int main(void)
{
    sixteen_sixteen_scaled =
        sixteen_sixteen_half * sixteen_sixteen_quarter;
}
```

We can watch this run in the debugger. In fact, this is the reason for including the **volatile** keyword in the variable declarations — I want to be able to see what is happening, and without **volatile**, even with optimizations turned off, the compiler will still aggressively put variables in registers and avoid using memory at all if it can.

While I don't seem to be able to use watches on these variables, as I can when using a hardware debugger and microcontroller, I can see the values change in memory as I step through the program. The values are in little-endian form. Translating this, I can see that **sixteen_sixteen_half** shows up as **0x00008000**, **sixteen_sixteen_quarter** shows up as **0x00004000**, and the result of the multiplication operation, **sixteen_sixteen_scaled**, is assigned **0x00002000**, representing one-eighth.

Code Size

If I bring up the Solution Explorer window (via the View menu in Atmel Studio), I can take a look at the output file properties by right-clicking. The generated **.hex** file indicates that it is using 310 bytes of flash. What happens if I scale up to a larger type? Well, if I change my **unsigned accum** declarations to use **unsigned long accum**, suddenly my flash usage goes up to 2776 bytes. That's a lot given that I have 8192 bytes of flash, but it still leaves me quite a bit of room for my own program code.

A Few Techniques

Let's say we want to scale a value to send to a linear DAC. Our DAC accepts 18-bit values. That means we can send it a value between **0x0** and **0x3FFFF**.

To work directly with an **Accum** type that will represent these values, we have to use an **unsigned long accum**. To declare an **unsigned long accum** variable that is initialized from an **unsigned long** variable, I can just cast it:

```
unsigned long accum encoder_accum =
    ( unsigned long accum )encoder_val;
```

We can also cast from a shorter integral type — for example, from an **unsigned accum** — and get the correct results. Beware of mixing signed and unsigned

types! (As you always should, when working in C.)

We can do math on our **unsigned long accum** types using the usual C math operators.

Let's say we want to get the **unsigned long accum** value converted back to an integral type. How would we do that? We use **bitsulk** to get the bitwise value (this is actually just a cast operation under the hood). Because we're going to truncate the fractional part, I add **0.5ULK** first.

```
unsigned long val =
    bitsulk( encoder_accum + 0.5ULK ) >> ULACCUM_FBIT;
```

If we want the remainder as an **unsigned long accum**, we can get it. Remember that the fractional part of the accumulator type is in the range [0.0..1.0) (that is, inclusive of zero, exclusive of 1). Note that the use of the mask here is not very portable, although there are some tricks I could do to make it more portable, but for now, I am more concerned about readability.

```
unsigned long accum remainder =
    ulkbits( bitsulk( encoder_accum ) & 0xFFFFFFFF );
```

The **ulkbits** and **bitsulk** operations are just casts, under the hood, so this boils down to a shift and mask.

The Embedded C specification defines a number of library functions that work with the fractional and accumulator types. For example, **abslk()** will give absolute value of an **unsigned long accum** argument. There are also rounding functions, like **roundulk()**. I have not actually had need of these. They seem to be supported in avr-gcc, but so far I have not needed them.

I hope this very brief tutorial may have saved you some time and aggravation in trying to use these rather obscure, but very useful, language features. Happy programming!
