C and C++ extensions simplify fixed-point DSP programming

Denise Ombres, Texas Instruments Inc

Assembly-language programming has always been a practical necessity with fixed-point DSPs. New and proposed extensions to C and C++, however, make high-level-language programming much easier.

The use of cheap, power-miserly, fixed-point DSPs is on the rise. At the same time, the use of high-level languages in DSPs is also increasing, primarily to minimize development costs. The combination of these two trends puts fixed-point DSP programmers in a bind. The standard ANSI C language doesn't support fixed-point data types, thus forcing programmers to write in assembly language and to deal with complicated and error-prone scaling issues.

Extensions to C—a fixed-point data type's being one example—can enable high-level-language code to execute efficiently on fixed-point DSP devices (see box, "Why C needs extensions for DSP"). With proper use of extensions that software companies are now integrating into ANSI-compliant compilers, you can produce code that's both fast and compact. And, although code portability is a concern with extensions, you can take steps to minimize portability problems.

Fixed-point, or fractional, data types are appealing for DSP programming, because they implement real numbers using integers (see box, "Data-type basics"). Consequently, they provide real arithmetic without the complex algorithms needed for floating-point operations—algorithms that tend to make floating-point processors expensive and power-hungry. Fractional arithmetic is, therefore, cheaper to implement than floating-point arithmetic but nevertheless maps well to DSP algorithm needs.

Unfortunately, standard C doesn't support fractional data types. It supports both signed and unsigned integers, as well as floating-point numbers, by means of its int, short, long, float, and double types together with the unsigned modifier. However, because C wasn't developed with DSPs in mind, it provides no fixed-point support.
To enable C to take advantage of the fractional type operations supported by DSP hardware, compiler writers can extend the C language with a new base type. Like integers and floats, this type can be a first-class type with a set of predefined values and operations. Like integral and floating-point types, fractional types are properly classified as arithmetic types (Reference 1).

Fractional types have many of the same uses as floating-point types. You can pass them as parameters and use them as elements of arrays, as return types, and in constant declarations.

Like floating-point representations, fractional types are not integral types, so they cannot appear as switch-statement control expressions or within pointer arithmetic. However, as with floating-point numbers, bidirectional conversions to and from integers are permissible.

Consider the specific example of the Motorola DSP568xx. It has six types of fractional representations: signed and unsigned 16-bit, signed and unsigned 32-bit, signed 20-bit, and signed 36-bit. (Both the 20- and the 32-bit representations have 4 bits to the left of the decimal rather than one.)

**New built-in fixed-point types**

A reasonable implementation of a C compiler might map these hardware types to two new C base types, frac and accum. These types could then combine with the modifiers unsigned and long to produce six new built-in types: frac, unsigned frac, long frac, unsigned long frac, accum, and long accum. Given those built-in types, you could then easily program a vector addition, for example, using fractional types (Figure 1).

Figure 1’s example illustrates the seamless integration of the fractional type into the C language and the flexibility of its use in declarations, assignments, conversions, indirections, and as a return type. If you changed the word frac in this code to float, the result would be a perfectly legitimate ANSI C program.

As with other built-in types in the C language, fractional types have a predefined set of conversion rules. These rules follow a conversion hierarchy for fractional types that is parallel to that for the other built-in types. For the Motorola DSP568xx, this hierarchy (Figure 2) is linear, with one exception: You can't promote the type accum to long accum, because accum has additional digits to the left of the binary point that the frac types cannot represent.

Because the hierarchy of fractional types is separate from the hierarchy of other built-in types, operations involving fractional types and other types require explicit casts. Consistent with C's general philosophy of type conversion, out-of-range errors when casting from float or double to fractional types yield undefined values. Results of casts from integral types may truncate.

Writing either a fractional or a floating-point number is most convenient with the notation of real numbers; you can simply write, for example, "1.0" or some other number. And, outside a programming context, it is not meaningful to ask whether "1.0" is a float or a frac. It's neither—it's simply the real 1.0.

**How to tell float from frac**

Unfortunately, using the same real notation for both float and frac can cause problems in the processing of C literals, which are traditionally of type float. Although it's theoretically possible for a
compiler to deduce the expected type from contextual information, a compiler that could do so would be unduly complicated. A notational convention for removing the ambiguity would work—for example, something like 0y1.0, by analogy with 0x001 for hexadecimal. A more natural solution for C, however, would be simply to use explicit casts to the appropriate fractional type (Figure 3).

By now, you should be able to appreciate the advantages of built-in fixed-point types relative to the traditional approach of mapping fixed-point numbers onto integers. Without the built-in types, you have to do literal conversions by hand. For example, to initialize a variable to a constant, you write:

```
int x = 0x4000; /* I really mean 0.5 */
```

Worse yet, without the built-in types, a compiler can't distinguish between integer and fractional operations. Writing x=y*z to multiply two 16-bit fractional values simply does not work. In fixed-point multiplication, you need the 16 MSBs of the result, but, in integer multiplication, you get the 16 LSBs. To get the correct result, you need to use a complicated expression of the form:

```
x = (int) ((long)y * (long)z)>>15;
```

Notice that the code generated for this expression is quite inefficient; it includes a 32332-bit multiplication and a shift.

By using special fractional types that are inherently different from integer types, a compiler can distinguish between operations performed on integers and operations performed on fractionals. It can then pick the best instruction to implement an operation. Initializing a variable would then be as easy as writing:

```
frac x = (frac)0.5;
```

and the instruction x = y * z works as expected, using the most efficient implementation—a single assembly instruction.

**Making code more efficient**

Now, consider an example that uses a *mac* (multiply accumulate) instruction. In many DSP algorithms, expressions of the form a+=b*c need to be implemented with a *mac* instruction to generate efficient code. Usually, *mac* instructions operate only on fractional types and don't apply to integers. So, if a compiler can't distinguish between real integer variables and fractional variables, it can't generate a *mac* instruction but must instead generate a whopping total of 16 instructions:

```
long a;
int b,c;

a += ((long)b * (long)c)>>15
```

With compiler support, however, you can write:

```
accum a;
frac b,c;

a += (accum) (b * c);
```

This sequence generates only a single assembly instruction:

```
mac y1,y0,A
```
You can, of course, have difficulties with portability if you incorporate language extensions, such as the fractional type, into an application. Fortunately, certain programming conventions can make porting easier.

Often, in applications ultimately targeted for DSPs, you may do initial development with self-hosted tools, and the self-hosted compiler may not accept the new keywords that denote fractional types. To get around this problem, you can use the C language typedef construct to map fractional types to their nearest neighbor—the floating-point type:

```c
typedef float frac
```

This approach enables you to detect most syntax bugs and even some semantic ones. It may not, however, enable the detection of all range restrictions or produce correct results because of the underlying differences in the internal representations of fractional and floating-point types. Nevertheless, it can be useful in the early stages of development.

**Portability and runtime libraries**

Another portability issue involves the use of fractional types with runtime libraries. You need more than just the definition of operations performed upon fractional types. You also need a predefined set of support routines that can perform mathematical functions and macros to define limits and characteristics of fractional types. Unfortunately, these function and macros may not be the same as those defined for floating-point types, so the previous solution (using `typedef`) does not work for those that don't overlap. However, you can use the `#ifdef` preprocessor directive to allow these constructs only during development, using compilers that support these extensions:

```c
#ifdef DSP568xx
#include
#endif
```

Both of these solutions impose a minimal amount of overhead on the programmer. They let you begin development of an application on a self-hosted system while you wait for the DSP hardware. In addition, after installation of the DSP hardware, the time needed to port the application is less. You can reuse code for a different DSP simply by modifying all the target-specific features that are already clearly delimited.

The use of high-level languages for DSP programming will increase in the years ahead because of several distinct advantages: a higher level of abstraction, faster time to market, and lower maintenance costs. As a result, vendors of development tools will feel pressure to provide programmers with tools that take full advantage of DSPs' features. In many cases, this pressure will result in the extension of high-level languages. The discussion here has centered on fixed-point types, but several vendors have already made extensions for multiple memory spaces and modulo addressing modes.

Ultimately, the portability problems caused by language extensions will have to be solved by DSP language-extension standards. Such standards will allow programmers to take advantage of both worlds: portable high-level language implementation on the one hand and optimal DSP performance on the other.
Why C needs extensions for DSP

Unfortunately for DSP programmers, C wasn't designed with DSP hardware in mind. Fractional numbers—also known as fixed-point types—are a case in point. C's type model comprises only two basic numeric types: integer and floating point. These two types are sufficient for most general-purpose processors and even for many DSPs, but they fall short for fixed-point DSPs.

For a high-level language to support the fractional types of DSP hardware, only three basic approaches are available. The traditional approach—the mapping of fixed-point types onto integer types—has two disastrous consequences. First, programming becomes an error-prone nightmare of illegible type conversions and misleading literals. Second, the compiler is unable to optimize for fractional arithmetic; as far as the compiler is concerned, there is no such thing as fractional arithmetic.

The second approach is to encapsulate fractional types as an Abstract Data Type (ADT)—for example, to provide a library of subroutines that create and manipulate fixed-point data. But, despite the appeal of ADTs in general, this solution is not acceptable in the case of fixed-point DSPs for two reasons. First, imposing function-call syntax on a feature as pervasive as arithmetic is intolerably cumbersome to a programmer. Second, even if the language were flexible enough to allow user-defined infix operators, the procedure-call overhead would be unacceptable in a DSP environment.

The third approach, and the one advocated here, is to follow the example of Ada and to extend the C language itself to include built-in fixed-point base types. By introducing these types, the compiler can optimize operations to make the best use of the underlying architecture registers and the processor's instruction set. It can also automatically introduce conversions to and from other types (integer, for example). Finally, it can support a natural notation that lets users concentrate on implementing the DSP algorithm at hand.
Data-type basics

The problem of numeric representation has occupied computer science since its inception. Positive integers were relatively straightforward; changes of base have long been well-understood. Negative numbers were a bit more challenging, and several different schemes appeared before the two's-complement representation achieved predominance. Still more troublesome are real numbers; the fact that they can't be exactly represented by a fixed number of bits has led to a multitude of approximate representations.

Currently, four data representations (Figure A) are in common use on binary computers to express numeric values: signed integers, unsigned integers, floating-point numbers, and fractional numbers.

An example of a signed integer is a 16-bit two's-complement value. In this representation, the value 5 appears as 0x0005 and the value -4 as 0xfffc. The values of integers can extend from -215 to +215-1.

In a 16-bit representation of an unsigned integer, the value 5 would still be represented as 0x0005, but 0xfffc now represents the value 65532. Integers extend from 0 to 65536 (216-1).

Floating-point numbers, such as the IEEE-754 single-precision, 32-bit value shown in Figure A, represent real values, where

\[ \text{value} = (-1)\text{Sign}32\text{Exponent}-12731.\text{Mantissa}. \]

For example, the value 2.0 appears as 0x40000000, and the value 1.2273 appears as 0x3f9d182b. The range of values is (approximately) -3.40331038 to +3.40331038. The smallest positive value is 1.192310-38.

In fractional types, such as the DSP568xx 16-bit fractional value shown in Figure A, the binary point is always assumed to be to the right of the MSB (hence, the name "fixed-point types"). In this example, the value 0.5 would be represented as 0x4000 and the value -0.75 as 0xA000. The values that can be represented in this way extend from -1.0 to 0.9997 (1-2-15). Other variants of fractional types include extending the precision to 31 bits to the right of the MSB (for a range of -1.0 to 0.9999999953) or extending the integer part to the left of the MSB by 4 additional bits (for a range of -16.0 to 15.9997).

References


Author's biography

Denise Ombres is a member of the C and C++ product team at Tartan (now a part of Texas Instruments), specializing in compiler front ends. She has a master's degree in computer science and a bachelor's degree in mathematics and computer science, both from the University of Pittsburgh. In her spare time, Ombres enjoys music, crafts, and exercise.