RETURN

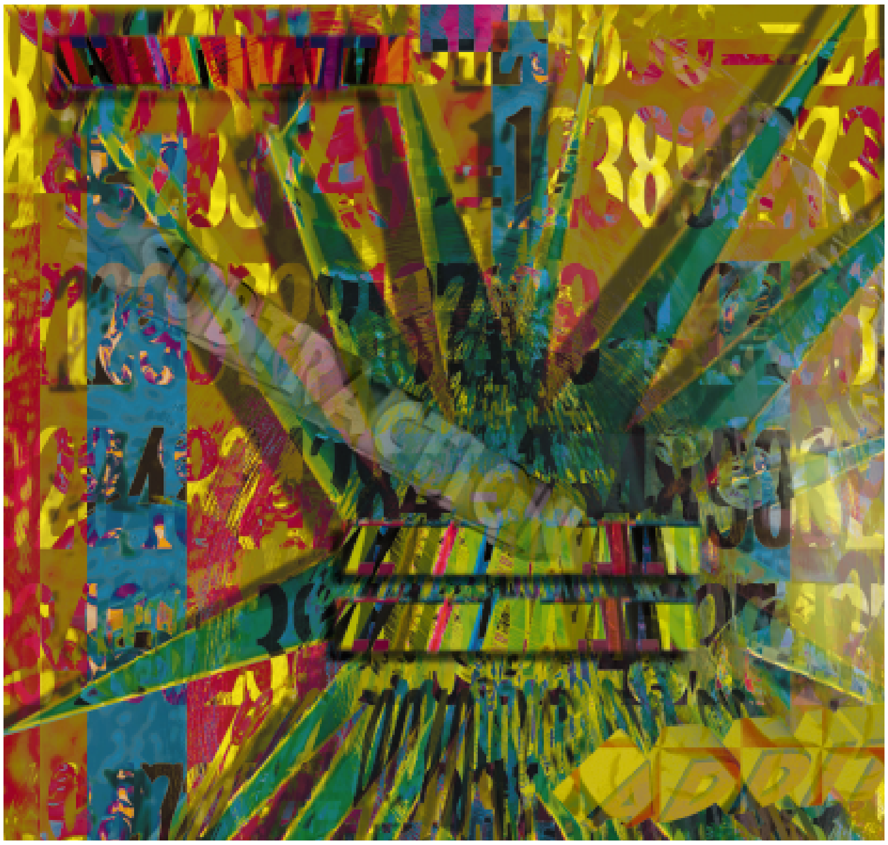
by ROBERT GORDON

A Calculated Look at Fixed-Point Arithmetic

Q: When is a real number not a real number?
A: When it's a scaled integer.
This article explores the subject of fixed-point numbers and presents techniques you can use to implement efficient, fixed-precision numeric applications.

Certain types of embedded systems require the handling of real numbers (or at least what appear to be real numbers). Real numbers can have fractional parts—in other words, something after the decimal point—in contrast to integers which are always whole numbers.

Few microprocessors offer real-number support, such as for floating-point data types and operations at the instruction level. Those which do are generally large, complex, expensive, and not intended for embedded applications. Certainly none of the small 4- and 8-bit microcontrollers support floating point, even though these are precisely the processors that are going to be at the heart of many apparently "real-number" applications.


Nance Paternoster

Consider a small, battery-powered handheld temperature meter that features a temperature probe, an LCD display, a few buttons, and of course a microcontroller. Clearly a high-end 32-bit microprocessor with floating-point support isn't the processor to use in this application. It would be too expensive, consume too much power, be physically too large, and would be overkill—the processing required would consume 1% of its potential.

The processor to use for this meter is a 4- or 8-bit microcontroller (the choice depending on the sophistication and features of the meter). The problem is that the meter is expected to display the temperature to several decimal places, and to offer the user the choice of units (˚C, ˚F, K). Also, let us say that the same meter software is supposed to support a number of different sensor ranges (0˚C to 50˚C, 0˚C to 100˚C, -20˚C to 100˚C, an so on). At the manufacturing stage, the desired temperature sensor would be installed, and links on the board would be configured to inform the software of the range required.

Suppose we read the temperature via

Scaling is easy to implement if your microcontroller provides suitable multiplication and division operations.

a 10-bit ADC (the range of values is zero to 1,023), and let us consider the 0˚C to 50˚C range. At two decimal places, for instance, this range is really 0.00 to 50.00. Reserving the two extreme ADC values to indicate "under-range" and "over-range," we have to make the mapping shown in Table 1.

In this case, the decimal point is really an illusion—there are always two decimal places, so instead of working with numbers in the range 0.00 to 50.00 we actually use 0 to 5,000 (the values are scaled up by a factor of 100). The decimal point is always displayed in the same position. When formatting the display value, consideration must be given to the effective position of the decimal point. We want to provide some leading zeros when the display result is less than 100. We want a value of seven to be displayed as "007" rather than " 7" (because "7" in this scaled scheme represents "0.07"). The display should show "0.07" rather than ".7."

This leaves the calculation to perform. We want to scale the range one to 1,022 to give zero to 5,000. Subtracting one from the ADC value gives the range zero to 1,021. Now both ranges start at zero, so we can say that the ADC value is converted to the display value by subtracting one and multiplying by (5,000/1,021); that is, multiplying by 4.897. Note that the ADC range is smaller than the display range, meaning that although the display will give the appearance of two decimal places, not all display values will be obtainable. This condition may or may not be acceptable—it depends on the application. If not, then either a different ADC should be chosen which offers more resolution (more bits), or the number of decimal places shown on the display should be decreased. If one decimal place was used then the display range would be zero to 500, which is now smaller than the ADC range, thus all display values should be obtainable. Another option would be to step the last digit in twos or fives (instead of in units). This will also decrease the display range.

Let's change the display to one decimal place so that every display value is obtainable. Our display range is now zero to 500, so the scaling factor is 0.4897.

How do we multiply the ADC value by 0.4897? This looks like a real-number (floating-point) operation. The input value is integer (zero to 1,021), and the result is integer (zero to 500). Can we perform integer operations to scale the input to the result? Yes we can. The solution is to perform two integer operations: a multiplication followed by a division. The constants chosen for each operation will form a fraction that is equivalent to the scaling factor:

Result = Input * 0.4897
0.4897 is equivalent to M/D (where M and D are integers)

For example:

M = 4,897; D = 10,000
4,897/10,000 = 0.4897

So, to perform the scaling we could do:

Result = (Input * 4897) / 10,000

Clearly, integer division will produce an integer result. What happens if

TABLE 1
Mapping for "under-range" and "over-range" ADC values.

ADC Value	Display Value
0	-ur- (under range)
1	0.00
..	..
1,022	50.00
1,023	-or- (over range)

the denominator doesn't divide exactly into the numerator (meaning, the result would have a fractional part)? In this case, an integer division will truncate the result to the previous whole number. In most cases, we would prefer a result of 499.99 to come out as 500 rather than 499. In other words, we would like the division to round to the nearest integer rather than truncate to the previous integer. How can we achieve this rounding? We need to add the equivalent of 0.5 before the truncation takes place. The equivalent of 0.5 is going to be *D*/2 for any denominator *D*. In the previous example, we need to add 5,000 (or, 10,000/2) to the numerator before performing the division.

Now we have the following scaling calculation:

Result = ((Input * 4,897) + 5,000)/10,000

Straightforward? Okay, straightforward in theory, perhaps. Is it easy to implement? Is it efficient? Is there anything else we need to consider?

Certainly this is a much more attractive option than having to implement floating-point arithmetic routines. This is much easier and more efficient.

This scaling is easy to implement if your microcontroller provides suitable multiplication and division operations. (By suitable, I mean operations with sufficient range to hold the intermediate results—more on this shortly.)

The scaling could be made more efficient. Multiplication and division on a conventional processor are slow operations (division being slower than multiplication). If a small microcontroller which doesn't have multiplication or division instructions is used (or has instructions with insufficient range), then these operations are going to have to be performed in software (either by subroutines or by in-line code). If this is the case, multiplication and division are going to be extremely slow. We can improve matters by eliminating one of these operations. Invariably the division is eliminated because that offers the greatest savings. To allow this, we must choose constants *M* and *D* such that *D* is a power of two ($D=2^1$, 2^2, 2^3, 2^4, and so on). Division by a power of two can be substituted by a right shift. The number of places to shift the intermediate result to the right is the same as the power to which two is raised in the divisor *D* (if $D = 2^8$, then shift right by eight places). Shifting is efficient when compared to division. In fact, if the power is a multiple of eight, no actual shifting is necessary—the result is obtained by discarding the least significant byte(s).

In the previous example, we might try *D* = 8,192 (2^{13}). Therefore, *M* = 0.4897 * 8,192. This comes to 4011.6224. We need an integer, so we will have to choose 4,011 or 4,012. Which do we choose? Will either be accurate enough?

The only way to find out is to try a scaling calculation. The top of the range input value is a good choice for this check:

Result = ((1,021 * 4,011) + 4,096)/8,192
= 500.4061

This will truncate to 500 during the division, so we can use *M* = 4,011. Incidentally, *M* = 4,012 gives a result of 500.53, which would also be acceptable—but this isn't always the case. Sometimes neither option gives a sufficiently accurate result, so a different divisor, *D*, must be tried.

If the values at both ends of the range of inputs scale correctly, it's reasonable to assume that all of the intermediate values will also scale correctly with this method. In our example, the bottom of the result range is zero. Zero will always scale to zero using this method, so it doesn't need to be checked.

Note that for the scaling part of the calculation, we always work with a result range with a lower limit of zero. After the scaling is complete, the result can then be adjusted by addition to the desired display range. For example, consider a display range of -10.0 to +30.0. In scaled integer terms, this will be -100 to 300 (call these Display_Low and Display_High). From the scaling calculation, we will produce a result in the range zero to 400 (that is, the total range adjusted to start from zero: Display_High - Display_Low). After scaling, the intermediate result is then adjusted by adding Display_Low; in this case, Display_Low = -100.

As I hinted earlier, we must ensure that during all of the calculation stages we are using storage locations and operations suitable for the range of values required at that stage. It will be obvious how big the input storage locations should be (those for "Input" and *M*). Because *M* is a constant, we may not need to store it explicitly in a location. Usually, every value involved in the calculation (at every stage) will be held in a conveniently sized location—even if this location is bigger than is required. For example, if the input value required 11 bits, that value would most likely be held in a 16-bit location. If a suitable multiplication instruction is available, this will dictate the size of the multiplication result (for example, an instruction which multiplies two 16-bit values and produces a 32-bit result). If no suitable instruction is available (no multiplication instructions provided with sufficient range for the inputs), then we will have to write a multiplication routine. In this case, the range of the intermediate result will have to be calculated. A routine providing a 24-bit result may be sufficient, so it would be wasteful to use a 32-bit result routine.

The result range is simply zero to ((Largest_Input * *M*) + *D*/2), because we always scale with a range based at zero. The higher of these values (hint: not zero, but the other one) is the biggest value of intermediate result that we need to handle. This value can then be transformed into a "required bits" figure by comparing it with powers of two. The required bits figure will be increased to the next convenient boundary—usually a multiple of eight bits. This, then, defines the result range required of the multiplication stage. After the "division" (shifting or discarding) stage, we will have a number in the range zero to (Display_High - Display_Low). Depending on the signs (+/-) of the display limits, this may be a larger or smaller range than the final display range; so this value may have to be extended into a larger location before the addition of Display_Low takes place. For example, if the display limits are 100 and 300, then the division result will be in the range zero to 200. This result will fit into an 8-bit location. However, before this value can be converted to the display range by the addition of 100, we will need to extend it into a larger space because values above 255 can't be stored in an 8-bit location.

In any given conversion calculation, there may be stages which do not apply. For instance, if the low limit of the display range is zero, then all of the Display_Low/Display_High stuff can be ignored. Anything which does not apply in a particular case may be dumped in the interest of efficiency (very important on a small, 4- or 8-bit microcontroller; less so on larger processors).

As a slight digression, let's examine an alternative approach that could have been used to solve the conversion/display problem if sufficient memory were available. All of the conversion and display formatting work could be performed in advance during development of the software, and a table of display strings (or compressed display data) created. Then the input value would be used as a mere index to pick out the corresponding display value. A table is required for each variant of display output supported (different display units, ranges, resolutions, and so forth).

This scaling for display is one form of fixed-point operation. More generally, fixed-point arithmetic refers to the handling of numbers which are scaled up by a certain factor to allow space for fractional parts. There are a fixed number of digits after the decimal point; the resolution is explicit. For instance, a number which is stored and manipulated such that its stored value is 1,000 times larger than the number it represents would have three fixed decimal places. Its resolution is exactly 0.001 under all circumstances. Creating numbers of this form is easy—just multiply the desired value by 1,000. Addition and subtraction are the same operations as they are for ordinary integer values. Perform integer addition on the representation of two fixed-point values and the result will be in the same format, with the same resolution. For example, 2.000 + 1.104 in integer form becomes 2,000 + 1,104, which is equal to 3,104. Interpreting this in fixed-point notation gives us 3.104, as we would expect.

Multiplication is a little more involved. Since both operands are scaled up by a factor *N* (1,000 in this example), then the multiplication result is going to be scaled up by N^2 ($1{,}000^2$). To restore the scaling we need to divide the result by *N*. Let's consider 2.000 * 3.000. In integer form, this will be 2,000 * 3,000, which gives us 6,000,000. (If this weren't rescaled, the interpretation looks like 2.000 * 3.000 = 6000.000, which is clearly wrong.) Dividing by 1,000 gives us 6,000 which comes out correctly as 6.000.

Division cancels the scaling. If *X* * *N* is divided by *Y* * *N*, then we have the equivalent of *X*/*Y* (there is no scaling on the result). The scaled result we require is actually (*X*/*Y*) * *N*. However, note that multiplying *X* by *N* before the division by *Y* is essential, otherwise resolution is lost. Thus, what we really do is (*X* * *N*)/*Y*.

Using the same scaling factor again (1,000), consider 10.000/4.000. If we do this in the wrong order we get 10,000/4,000, which gives two (integer division truncates). Multiplying by 1,000 and interpreting this looks like 10.000/4.000 = 2.000 (loss of precision). If we do this in the correct order, we are doing (10,000 * 1,000)/4,000 which is 2,500. Interpreting this gives the result we would expect, 10.000 / 4.000 = 2.500.

Unfortunately, both multiplication and division involve intermediate values which are *N* times larger than the operands. This means that the range of the intermediate type must be *N* times larger than the range of the operand type.

The previous examples show a power-of-ten scaling factor. This condition is purely to make the examples easy to read and easy to understand. In practice, using power-of-two scaling is much more efficient, as the extra scaling operations involved in multiplication and division can be performed using shifts. ESP

Robert Gordon works in firmware development for Nortel Broadband Networks in Newtownabbey, U.K. He may be reached at rgordon@nortel.ca.