Q (number format)

2 languages

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.
Find sources: "Q" number format – news · newspapers · books · scholar · JSTOR *(May 2015) (Learn how and when to remove this template message)*

The **Q notation** is a way to specify the parameters of a binary fixed point number format. For example, in Q notation, the number format denoted by `Q8.8` means that the fixed point numbers in this format have 8 bits for the integer part and 8 bits for the fraction part.

A number of other notations have been used for the same purpose.

Definition [edit]

Texas Instruments version [edit]

The Q notation, as defined by Texas Instruments,[1] consists of the letter `Q` followed by a pair of numbers *m*`.`*n*, where *m* is the number of bits used for the integer part of the value, and *n* is the number of fraction bits.

By default, the notation describes *signed* binary fixed point format, with the unscaled integer being stored in two's complement format, used in most binary processors. The first bit always gives the sign of the value(1 = negative, 0 = non-negative), and it is *not* counted in the *m* parameter. Thus the total number *w* of bits used is 1 + *m* + *n*.

For example, the specification `Q3.12` describes a signed binary fixed-point number with a *w* = 16 bits in total, comprising the sign bit, three bits for the integer part, and 12 bits that are the fraction. That is, a 16-bit signed (two's complement) integer, that is implicitly multiplied by the scaling factor 2^{-12}

In particular, when *n* is zero, the numbers are just integers. If *m* is zero, all bits except the sign bit are fraction bits; then the range of the stored number is from −1.0 (inclusive) to +1 (exclusive).

The *m* and the dot may be omitted, in which case they are inferred from the size of the variable or register where the value is stored. Thus `Q12` means a signed integer with any number of bits, that is implicitly multiplied by 2^{-12}.

The letter `U` can be prefixed to the `Q` to denote an *unsigned* binary fixed-point format. For example, `UQ1.15` describes values represented as unsigned 16-bit integers with implicit scaling factor of 2^{-15}, which range from 0.0 to $(2^{16}-1)/2^{15}$ = +1.999969482421875.

ARM version [edit]

A variant of the Q notation has been in use by ARM. In this variant, the *m* number includes the sign bit. For example, a 16-bit signed integer would be denoted `Q15.0` in the TI variant, but `Q16.0` in the ARM variant.[2][3]

Characteristics [edit]

The resolution (difference between successive values) of a Q*m*.*n* or UQ*m*.*n* format is always 2^{-n}. The range of representable values depends on the notation used:

Range of representable values in Q notation

Notation	Texas Instruments Notation	ARM Notation
Signed Q*m*.*n*	-2^m to $+2^m - 2^{-n}$	-2^{m-1} to $+2^{m-1} - 2^{-n}$
Unsigned UQ*m*.*n*	0 to $2^m - 2^{-n}$	0 to $2^m - 2^{-n}$

For example, a Q15.1 format number requires 15+1 = 16 bits, has resolution 2^{-1} = 0.5, and the representable values range from -2^{14} = −16384.0 to $+2^{14} - 2^{-1}$ = +16383.5. In hexadecimal, the negative values range from 0x8000 to 0xFFFF followed by the non-negative ones from 0x0000 to 0x7FFF.

Math operations [edit]

Q numbers are a ratio of two integers: the numerator is kept in storage, the denominator d is equal to 2^n.

Consider the following example:

- The Q8 denominator equals 2^8 = 256
- 1.5 equals 384/256
- 384 is stored, 256 is inferred because it is a Q8 number.

If the Q number's base is to be maintained (*n* remains constant) the Q number math operations must keep the denominator d constant. The following formulas show math operations on the general Q numbers N_1 and N_2. (If we consider the example as mentioned above, N_1 is 384 and d is 256.)

$$\frac{N_1}{d} + \frac{N_2}{d} = \frac{N_1 + N_2}{d}$$

$$\frac{N_1}{d} - \frac{N_2}{d} = \frac{N_1 - N_2}{d}$$

$$\left(\frac{N_1}{d} \times \frac{N_2}{d}\right) \times d = \frac{N_1 \times N_2}{d}$$

$$\left(\frac{N_1}{d} / \frac{N_2}{d}\right) / d = \frac{N_1 / N_2}{d}$$

Because the denominator is a power of two, the multiplication can be implemented as an arithmetic shift to the left and the division as an arithmetic shift to the right; on many processors shifts are faster than multiplication and division.

To maintain accuracy, the intermediate multiplication and division results must be double precision and care must be taken in rounding the intermediate result before converting back to the desired Q number.

Using C the operations are (note that here, Q refers to the fractional part's number of bits) :

Addition [edit]

```c
int16_t q_add(int16_t a, int16_t b)
{
    return a + b;
}
```

With saturation

```c
int16_t q_add_sat(int16_t a, int16_t b)
{
    int16_t result;
    int32_t tmp;

    tmp = (int32_t)a + (int32_t)b;
    if (tmp > 0x7FFF)
        tmp = 0x7FFF;
    if (tmp < -1 * 0x8000)
        tmp = -1 * 0x8000;
    result = (int16_t)tmp;

    return result;
}
```

Unlike floating point ±Inf, saturated results are not sticky and will unsaturate on adding a negative value to a positive saturated value (0x7FFF) and vice versa in that implementation shown. In assembly language, the Signed Overflow flag can be used to avoid the typecasts needed for that C implementation.

Subtraction [edit]

```c
int16_t q_sub(int16_t a, int16_t b)
{
    return a - b;
}
```

Multiplication [edit]

```c
// precomputed value:
#define K   (1 << (Q - 1))

// saturate to range of int16_t
int16_t sat16(int32_t x)
{
    if (x > 0x7FFF) return 0x7FFF;
    else if (x < -0x8000) return -0x8000;
    else return (int16_t)x;
}

int16_t q_mul(int16_t a, int16_t b)
{
    int16_t result;
```

```
    int32_t temp;

    temp = (int32_t)a * (int32_t)b; // result type is operand's type
    // Rounding; mid values are rounded up
    temp += K;
    // Correct by dividing by base and saturate result
    result = sat16(temp >> Q);

    return result;
}
```

Division [edit]

```
int16_t q_div(int16_t a, int16_t b)
{
    /* pre-multiply by the base (Upscale to Q16 so that the result will be in Q8 format) */
    int32_t temp = (int32_t)a << Q;
    /* Rounding: mid values are rounded up (down for negative values). */
    /* OR compare most significant bits i.e. if (((temp >> 31) & 1) == ((b >> 15) & 1)) */
    if ((temp >= 0 && b >= 0) || (temp < 0 && b < 0)) {
        temp += b / 2;    /* OR shift 1 bit i.e. temp += (b >> 1); */
    } else {
        temp -= b / 2;    /* OR shift 1 bit i.e. temp -= (b >> 1); */
    }
    return (int16_t)(temp / b);
}
```

See also [edit]

- Binary scaling
- Fixed-point arithmetic
- Floating-point arithmetic

References [edit]

1. ^ "Appendix A.2". *TMS320C64x DSP Library Programmer's Reference* (PDF). Dallas, Texas, USA: Texas Instruments Incorporated. October 2003. SPRU565. Archived (PDF) from the original on 2022-12-22. Retrieved 2022-12-22. (150 pages)
2. ^ "ARM Developer Suite AXD and armsd Debuggers Guide". 1.2. ARM Limited. 2001 [1999]. Chapter 4.7.9. AXD > AXD Facilities > Data formatting > Q-format. ARM DUI 0066D. Archived from the original on 2017-11-04.
3. ^ "Chapter 4.7.9. AXD > AXD Facilities > Data formatting > Q-format". *RealView Development Suite AXD and armsd Debuggers Guide* (PDF). 3.0. ARM Limited. 2006 [1999]. pp. 4–24. ARM DUI 0066G. Archived (PDF) from the original on 2017-11-04.

Further reading [edit]

- Oberstar, Erick L. (2007-08-30) [2004]. "Fixed Point Representation & Fractional Math" (PDF). 1.2. Oberstar Consulting. Archived (PDF) from the original on 2017-11-04. Retrieved 2017-11-04. (Note: the accuracy of

the article is in dispute; see discussion.)

External links [edit]

- "Q-Number-Format Java Implementation". Archived from the original on 2017-11-04. Retrieved 2017-11-04.
- "Q-format Converter". Archived from the original on 2021-06-25. Retrieved 2021-06-25.

Category: Computer arithmetic