Fixed-Point Numbers

Table of contents

The binary point 1
Converting a decimal value 2
Format names: $m.n$ 3
Reading a pattern 3
Two's complement fixed point 4
Range against precision 4
Worked examples 5
Why use it 6
Common mistakes 8
See also 8

How a fixed number of bits can hold a fraction: put a binary point at an agreed place and give the bits to its right the weights $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, ... This note covers the binary point and its place values, converting a decimal value such as 4.375 into bits, the $m.n$ format names, reading a bit pattern in a given format, unsigned and two's complement fixed point, and the trade-off between range and precision as the point moves across a byte. The way round that trade-off, letting the point float, is in floating-point-numbers[1]. Whole numbers and the signed formats this builds on are in binary-number-representations[2]. First met in 2026-10-19-floating-point-numbers[3].

The binary point

Binary place values carry on to the right of the point exactly as decimal ones do. In decimal the column after the point is worth 10^{-1}; in binary it is worth $2^{-1} = \frac{1}{2}$, the next $2^{-2} = \frac{1}{4}$, then $\frac{1}{8}$, $\frac{1}{16}$, and so on, each half the one before.

[1] courses/csse2010/floating-point-numbers.pdf
[2] courses/csse2010/binary-number-representations.pdf
[3] courses/csse2010/2026-10-19-floating-point-numbers.pdf

Position	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}
Worth	8	4	2	1	.	0.5	0.25	0.125	0.0625

The point is not stored anywhere. The bits are an ordinary 8-bit pattern, and **where the point sits is an agreement**, made by whoever writes the program, in the same way that "this byte is two's complement" is an agreement (binary-number-representations[4]). Because the agreed place never moves, the scheme is called **fixed point**.

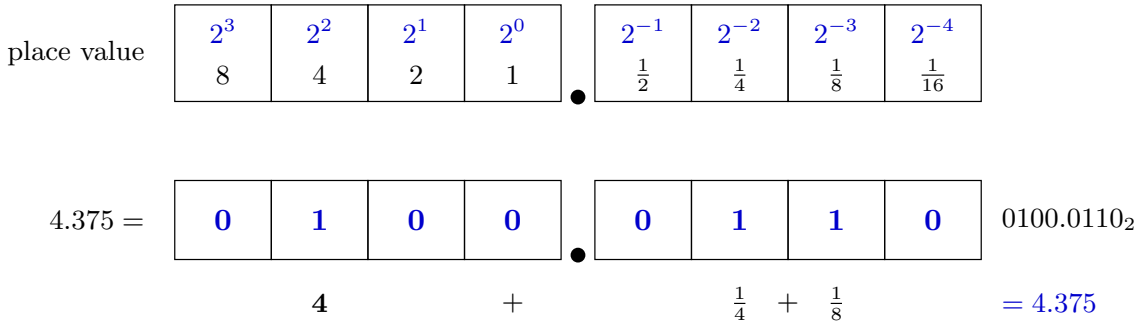


Figure 1: Eight bits with the binary point agreed to sit after the fourth. The four bits to the left are worth 8, 4, 2 and 1 as usual; the four to the right carry on halving, to 1/2, 1/4, 1/8 and 1/16. The value 4.375 is 4 + 1/4 + 1/8, so the bits worth 4, 1/4 and 1/8 are set and every other bit is 0, giving 0100.0110. Nothing in the byte marks where the point is: the same eight bits read as a whole number are 70, which is 4.375 times 16.

Converting a decimal value

Two methods, both worked on 4.375 into a format with 4 bits after the point.

Method 1: whole part and fraction separately. The whole part converts as usual: $4 = 100_2$. The fraction converts by **repeated doubling**: double it, the digit before the point is the next bit, and carry on with what is after the point. Stop when the fraction reaches 0 or the bits run out.

Fraction	×2	Bit	Carry on with
0.375	0.75	0	0.75
0.75	1.5	1	0.5

[4] courses/csse2010/binary-number-representations.pdf

Fraction	×2	Bit	Carry on with
0.5	1.0	1	0

The bits come out **left to right**, starting just after the point: $0.375 = 0.011_2$. So $4.375 = 100.011_2$, and padded to four bits each side, `0100.0110`.

Doubling is the mirror image of the repeated division for whole numbers (binary-number-representations[5]): division pulls bits off the right-hand end, doubling pulls them off the left-hand end of the fraction. The fraction can also be taken apart by subtraction, as with Method 1 for whole numbers: $0.375 - 0.25 = 0.125$, and $0.125 = \frac{1}{8}$, so the bits worth $\frac{1}{4}$ and $\frac{1}{8}$ are set.

Method 2: scale to a whole number. With n bits after the point, every value is a whole number of 2^{-n} steps. Multiply by 2^n, convert the whole number, and put the point back n places from the right. With $n = 3$: $4.375 \times 8 = 35 = 100011_2$, so $4.375 = 100.011_2$. With $n = 4$ it is $4.375 \times 16 = 70 = 1000110_2$, the same bits with one more 0 on the end.

If the product is not a whole number, the value has no exact representation with n fractional bits (below).

Format names: $m.n$

A fixed-point format is named by how the bits are split: $m.n$ **means** m **integer bits before the point and** n **fractional bits after it**, $m + n$ bits in all. Every 8-bit format has $m + n = 8$:

- **8.0** is an ordinary unsigned byte, the point after the last bit;
- **5.3** has five integer bits and three fractional bits, `xxxxx.xxx`;
- **0.8** has the point before the first bit, so every value is below 1.

The same value sits in different formats with the point in different places. 4.375 is `0100.0110` in 4.4, `00100.011` in 5.3, and `100.01100` in 3.5. Moving the point one place right with the bits fixed doubles the value; so to keep the value the same, the bits have to move the other way.

Reading a pattern

To find the value of a bit pattern in a given format, either add the place values of the 1 bits, or **read the bits as a whole number and divide by** 2^n. The second is quicker when the pattern is long.

`10110101` as unsigned 5.3: the whole number is 181, and $181/2^3 = 22.625$. By place values, $10110.101_2 = 16 + 4 + 2 + \frac{1}{2} + \frac{1}{8} = 22.625$, the same.

[5] courses/csse2010/binary-number-representations.pdf

Two's complement fixed point

Signed fixed point uses two's complement in the same way as signed whole numbers (binary-number-representations[6]). **The MSB's weight is negated**: in an $m.n$ format it is worth -2^{m-1} instead of $+2^{m-1}$, and every other bit keeps its place value. So in 5.3 the bits are worth

$$-16,\ 8,\ 4,\ 2,\ 1\ .\ \tfrac{1}{2},\ \tfrac{1}{4},\ \tfrac{1}{8}$$

and in 4.4 they are worth $-8, 4, 2, 1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}$.

Negating works as for integers: invert every bit and add 1 **at the last place**, which is now worth 2^{-n}, not 1. Equivalently, the value is the unsigned reading minus 2^m whenever the MSB is 1, which is the divide-by-2^n rule with 256 subtracted first: `10110101` is $(181 - 256)/8 = -75/8 = -9.375$ in two's complement 5.3. The same question with the two tempting wrong answers, -10.625 and -9.625, explained is 2026-10-19-floating-point-numbers[7].

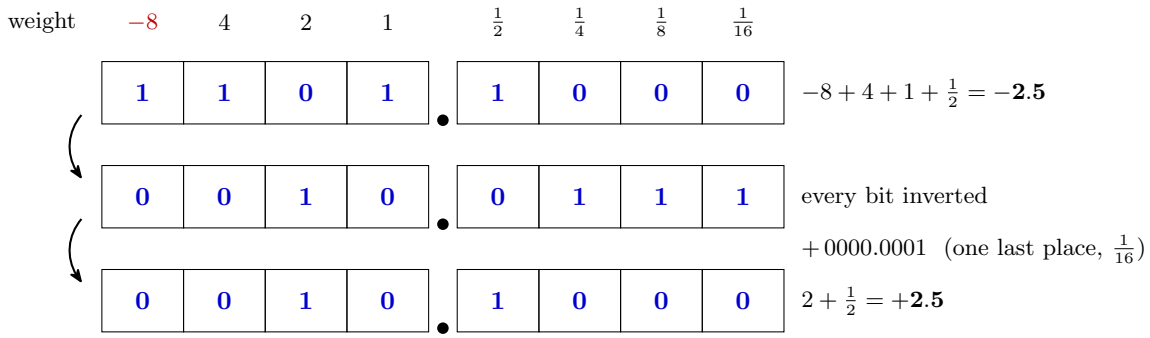


Figure 2: Two's complement fixed point in the 4.4 format. The only change from unsigned is the MSB, whose weight becomes minus 8 instead of plus 8; the fraction bits keep their positive weights. So 1101.1000 is minus 8 plus 4 plus 1 plus 1/2, which is minus 2.5. Negating it still means inverting every bit and adding one, but the one is added in the last place, which is worth 1/16: 0010.0111 plus 0.0001 is 0010.1000, which is plus 2.5.

Range against precision

Eight bits give 256 patterns whatever the format. The format only decides **how far apart** the 256 values are and so **how far they reach**. For an unsigned $m.n$ format:

[6] courses/csse2010/binary-number-representations.pdf
[7] courses/csse2010/2026-10-19-floating-point-numbers.pdf

- the **precision**, the gap between neighbouring values, is the weight of the last bit, 2^{-n};
- the **maximum** is all ones, $(2^8 - 1) \times 2^{-n} = 255 \times 2^{-n}$, one step short of 2^m;
- the minimum is 0.

Two's complement $m.n$ has the same precision and runs from -2^{m-1} to $2^{m-1} - 2^{-n}$.

Format	Unsigned maximum	Precision	Two's complement range
"10.$-$2"	1020	4	−512 to 508
"9.$-$1"	510	2	−256 to 254
8.0	255	1	−128 to 127
7.1	127.5	0.5	−64 to 63.5
6.2	63.75	0.25	−32 to 31.75
5.3	31.875	0.125	−16 to 15.875
4.4	15.9375	0.0625	−8 to 7.9375
3.5	7.96875	0.03125	−4 to 3.96875
2.6	3.984375	0.015625	−2 to 1.984375
1.7	1.9921875	0.0078125	−1 to 0.9921875
0.8	0.99609375	0.00390625	−0.5 to 0.49609375

Each place the point moves right doubles the maximum and doubles the step. Range and precision trade against each other one for one, and no 8-bit format gets both.

"9.$-1"*and*"10.**-$2"** carry the pattern past 8.0. The point is now *to the right of the byte*: one or two invisible bits, always 0, sit between the last stored bit and the point. The stored bits are worth 2^8 down to 2^1 (or 2^9 down to 2^2), so every value is a multiple of 2 (or 4), and the maximum is $255 \times 2 = 510$ (or $255 \times 4 = 1020$). The "−1" counts fractional bits that are missing: $n = -1$ means the last bit is worth $2^{-(-1)} = 2$. The quote marks are there because the name stretches the $m.n$ idea past where it reads naturally, but the formulas above still hold with negative n.

Worked examples

Worked: 4.375 in three formats

Write 4.375 in the 8-bit unsigned formats 4.4, 5.3 and 3.5.

$4.375 = 100.011_2$ (Method 1 or 2 above). It needs 3 integer bits and 3 fractional bits, so it fits all three formats exactly; only the padding changes.

Format	Bits	Check
4.4	`0100.0110`	$70/16 = 4.375$

Format	Bits	Check
5.3	`00100.011`	$35/8 = 4.375$
3.5	`100.01100`	$140/32 = 4.375$

In 2.6 it would not fit: 4 needs a bit worth 4, and 2.6's highest bit is worth 2.

Worked: choosing a format

> A byte must hold a percentage, 0 to 100, as precisely as possible. Which unsigned format? And for a signed value between -3 and $+3$?

Percentage. The maximum must reach 100. Going down the table, 7.1 reaches 127.5 and 6.2 only 63.75, so **7.1**, with a precision of 0.5.

± 3**.** In two's complement, 3.5 runs from -4 to 3.96875 and 2.6 from -2 to 1.984375. So **two's complement 3.5**, with a precision of $\frac{1}{32}$.

In both cases the rule is: take the **smallest** m that still reaches the largest value, because every integer bit not needed is a fractional bit lost.

Worked: 0.1 has no exact form

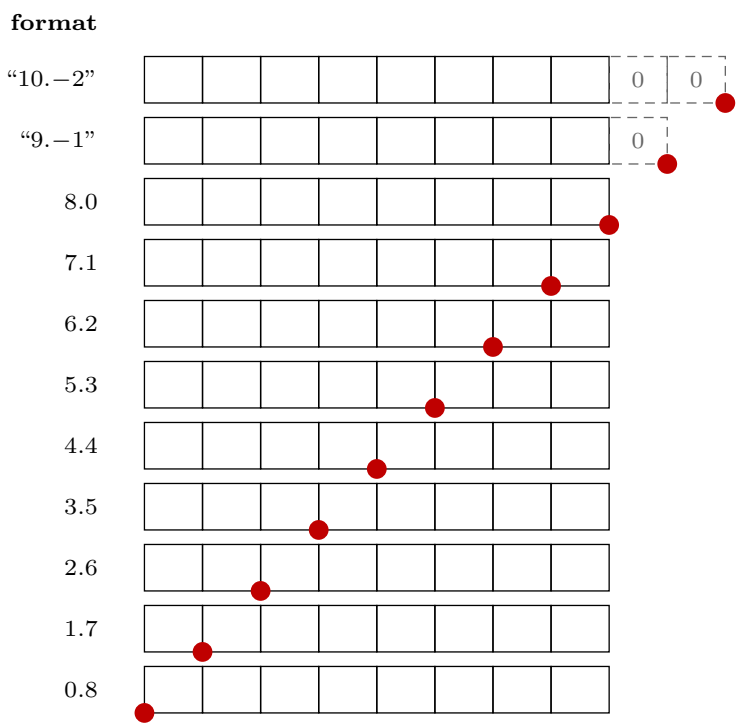
> Store 0.1 in unsigned 4.4.

Method 2: $0.1 \times 16 = 1.6$, not a whole number, so 0.1 cannot be stored exactly. The nearest step is $2/16 = 0.125$ (`0000.0010`); truncating gives $1/16 = 0.0625$ (`0000.0001`). Repeated doubling shows why no number of bits is enough: $0.1 \to 0.2 \to 0.4 \to 0.8 \to 1.6 \to 1.2 \to 0.4 \to \ldots$ gives $0.000110011\ldots_2$, with `0011` repeating for ever. A fraction is finite in binary only when its denominator is a power of 2, so 0.375 ($\frac{3}{8}$) is exact and 0.1 ($\frac{1}{10}$) is not.

Why use it

Fixed point needs **no new hardware**. Two values in the same $m.n$ format add and subtract with the ordinary integer adder, because the points line up; only the reading changes. That makes it the natural choice for fractions on a small processor like the AVR, which has integer instructions only (instruction-set-architecture[8]). The price is the trade-off above: the range and the precision are fixed once the format is chosen, and a value too large or too small for it simply cannot be held. floating-point-numbers[9] stores the position of the point alongside the digits so that it can move.

[8] courses/csse2010/instruction-set-architecture.pdf
[9] courses/csse2010/floating-point-numbers.pdf



format	maximum	precision
“10.−2”	1020	4
“9.−1”	510	2
8.0	255	1
7.1	127.5	0.5
6.2	63.75	0.25
5.3	31.875	0.125
4.4	15.9375	0.0625
3.5	7.96875	0.03125
2.6	3.984375	0.015625
1.7	1.9921875	0.0078125
0.8	0.99609375	0.00390625

Figure 3: The binary point moving across the same eight bits. The highest bit is worth 2 to the m minus 1 and the lowest is worth 2 to the minus n, so each step of the point to the right doubles both the maximum value and the gap between neighbouring values. At 8.0 the byte is an ordinary unsigned integer. Past that, in the 9.-1 and 10.-2 formats, the point lies beyond the byte, with one or two bits that are always 0 between them (drawn dashed), so only even numbers or multiples of 4 can be held, up to 510 or 1020. At the other end, 0.8 holds only values below 1, in steps of 1/256.

Common mistakes

- **Reading the fraction bits as a whole number.** `.011` is $\frac{1}{4} + \frac{1}{8} = 0.375$, not "0.3".
- **Forgetting that the point is an agreement.** A byte has no point in it; the same `10110101` is 181, 22.625 or −9.375 depending on the format chosen.
- **Giving the MSB a positive weight in two's complement.** In two's complement 5.3, the first bit is worth −16; only that bit is negative, and the fraction bits still add.
- **Negating by adding 1 to the integer part.** The 1 is added in the *last* place, 2^{-n}.
- **Taking the maximum as 2^m.** It is one step short: $2^m - 2^{-n}$ (31.875 for 5.3, not 32).
- **Reading the doubling remainders in the wrong order.** For a fraction the bits come out left to right, the first one just after the point. That is the opposite of repeated division for a whole number.
- **Expecting every decimal fraction to be exact.** Only fractions with a power-of-2 denominator are.

See also

- floating-point-numbers[10] — letting the point move, IEEE 754
- binary-number-representations[11] — place values, unsigned and two's complement whole numbers
- instruction-set-architecture[12] — the AVR's 8-bit integer data type, the only one it has in hardware
- 2026-10-19-floating-point-numbers[13] — Lecture 23

[10] courses/csse2010/floating-point-numbers.pdf
[11] courses/csse2010/binary-number-representations.pdf
[12] courses/csse2010/instruction-set-architecture.pdf
[13] courses/csse2010/2026-10-19-floating-point-numbers.pdf