Floating-Point Numbers

Table of contents

Why the point has to float . . . 2
Scientific notation . . . 2
Why "mantissa" . . . 3
IEEE 754 . . . 3
Encoding: decimal to IEEE . . . 5
Decoding: IEEE to decimal . . . 7
Special values . . . 8
Finite precision . . . 12
The formats compared . . . 12
Every quarter-precision value . . . 14
Floating point in C and on the AVR . . . 16
Common mistakes . . . 17
See also . . . 19

How a computer holds numbers as large as 6×10^{23} and as small as 3.6×10^{-4} in the same few bytes: store the digits and the position of the point separately, as scientific notation in binary. This note covers why a fixed point is not enough, scientific notation and normalisation, where the word "mantissa" comes from, the IEEE 754 single and double precision formats (sign, excess-127 or excess-1023 exponent, mantissa with an implicit leading 1, and why the fields are in that order), converting in both directions, the special values (zero, subnormal numbers, infinity and NaN), what finite precision does to arithmetic, the family of formats from quarter to octuple precision with every quarter-precision value listed, and floating point in C and on the AVR. Fixed point, the alternative this improves on, is in fixed-point-numbers[1]. The excess notation behind the exponent is in binary-number-representations[2]. First met in 2026-10-19-floating-point-numbers[3].

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

Why the point has to float

A fixed-point format spends its bits once and for all between range and precision (fixed-point-numbers[4]). Real quantities do not cooperate.

- **Avogadro's number** is $6.02214076 \times 10^{23}$, written out `602214076000000000000000`. As a whole number it needs **79 bits** ($\log_2 \approx 78.99$), and only its first nine digits carry any information.
- **The volume of a cup** is about 0.00036 m^3. In fixed point its first 1 is in the 12th place after the point ($2^{-12} \approx 0.000244$), so it needs at least 12 fractional bits just to be non-zero, and more for any precision.

To hold both in one format, a fixed-point number would need about 79 integer bits *and* a few dozen fractional bits, nearly all of them zero in any given value. The way out is the one people already use: write the significant digits once and say separately **where the point goes**.

Scientific notation

Everyday language already lets the point move. **SI prefixes** do it in words: a kilometre is 10^3 m, a milligram 10^{-3} g, and the scale runs out to femto (10^{-15}), yocto (10^{-24}) and beyond. The number in front stays a convenient size and the prefix carries the scale.

Scientific notation does it in symbols:

$$\text{value} = \text{significant digits} \times \text{base}^{\text{exponent}}$$

so Avogadro's number is 6.02×10^{23} and the cup is 3.6×10^{-4} m^3. The digits say *what* the number is and the exponent says *how big*. Moving the point one place left and adding 1 to the exponent leaves the value unchanged, which is exactly the freedom fixed point lacks.

Normalisation

That freedom means one value has many spellings: $602 \times 10^{21} = 60.2 \times 10^{22} = 6.02 \times 10^{23} = 0.602 \times 10^{24}$. A **normalised** number picks one of them: **exactly one non-zero digit before the point**. In decimal that puts the digits in the range $[1, 10)$, and 6.02×10^{23} is the normalised form.

In binary the same rule gives the range $[1, 2)$**.** The only non-zero binary digit is 1, so a normalised binary number always looks like $1.xxxx_2 \times 2^e$. Something like 3.985×2^{77} is not normalised in binary, even though 3.985 would be fine in front of a power of 10: it should be

[4] courses/csse2010/fixed-point-numbers.pdf

1.993×2^{78}. Halving the digits and adding one to the exponent, or doubling and subtracting one, moves between the spellings.

Normalising gives every value a single representation, so no bit patterns are wasted on duplicates, and it guarantees no precision is spent on leading zeros. In binary it does one more thing, used below: the first digit is always 1, so it need not be stored.

Why "mantissa"

The digits part of a floating-point number is called the **mantissa** (the IEEE standard's own term is *significand*). The word comes from **logarithm tables**.

Before calculators, multiplication was done with $\log(a \times b) = \log a + \log b$: look up two logarithms, add them, and look up the antilog. A printed table only needs the logarithms of 1.00 to 9.99, because scaling by 10 only adds 1 to the logarithm:

x	$\log_{10} x$	Integer part	Fractional part
0.123	-0.9101	-1	0.0899
1.23	0.0899	0	0.0899
12.3	1.0899	1	0.0899
123	2.0899	2	0.0899

The fractional part, the same 0.0899 for all four, depends only on the significant digits 1, 2, 3, and it was called the **mantissa**. The integer part, which only says where the point goes, was the **characteristic**. A four-figure table lists the mantissas: the row for 12 and the column for 3 give `0899`, meaning $\log_{10} 1.23 = 0.0899$. (For 0.123 the log is $0.0899 - 1 = -0.9101$; tables wrote it $\bar{1}.0899$ to keep the mantissa visible.)

A floating-point number splits the same way: digits that say *what*, an exponent that says *where the point goes*. The digits inherited the name, although strictly the old mantissa was a logarithm and the new one is the digits themselves.

IEEE 754

Almost every computer stores floating point in the formats of the **IEEE 754** standard. A number is stored as three fields, **sign**, **exponent** and **mantissa**, and stands for

$$(-1)^S \times 1.M_2 \times 2^{E-\text{bias}}$$

Format	Bits	Sign	Exponent	Mantissa	Bias
Single precision	32	1	8	23	127
Double precision	64	1	11	52	1023

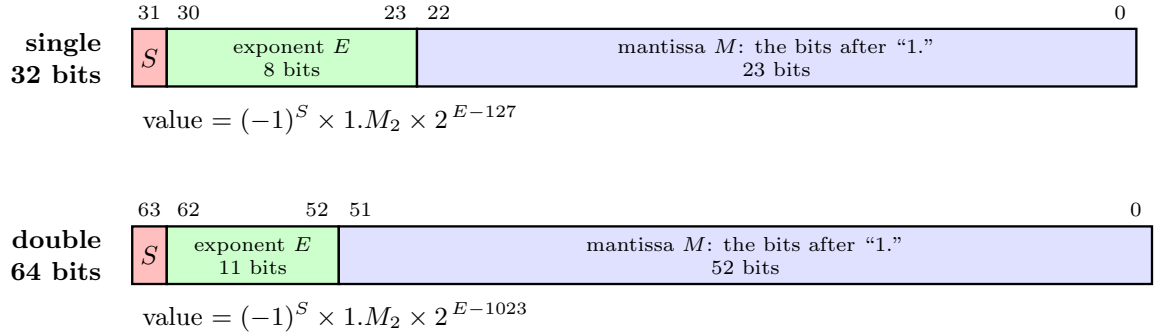


Figure 1: The IEEE 754 single and double precision layouts, MSB on the left, with the bit number at each field boundary. The sign bit is 0 for plus and 1 for minus. The exponent field E holds the true exponent plus the bias, 127 or 1023. The mantissa field M holds only the bits after the point, because a normalised binary mantissa always starts with a 1 that does not need storing. Reading a pattern means putting those three back together as (-1) to the S, times 1.M, times 2 to the E minus the bias. The exponent and mantissa are drawn to scale within each row, double precision at half the scale of single; the sign bit is drawn wider than one bit so it can be labelled.

The biased exponent

The exponent can be negative, so it needs a signed format. IEEE uses **excess** (biased) notation: the field holds the true exponent plus a fixed **bias**,

$$E = e + \text{bias}, \qquad \text{bias} = 2^{k-1} - 1 \text{ for a } k\text{-bit exponent.}$$

So single precision is **excess-127** and double precision **excess-1023**. A true exponent of 4 is stored as $4 + 127 = 131 = 10000011_2$; a true exponent of -2 as $125 = 01111101_2$.

This is the excess notation of binary-number-representations[5], with one difference to watch: the bias there is 2^{k-1} (excess-128 for 8 bits), while IEEE uses one less, 127. The stored values 0 and 255 are kept for the special values, so the normal true exponents run from $1 - 127 = -126$ to $254 - 127 = +127$.

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

The implicit leading 1

A normalised binary mantissa is always $1.xxxx_2$ (normalisation). A bit that is always 1 carries no information, so **IEEE does not store it**. The mantissa field holds only the bits after the point, and the 1 is put back when the number is read. Single precision therefore has **24 significant bits** for the price of 23. The one exception is the subnormal numbers, where the hidden bit is 0.

Why sign, exponent, mantissa

The order is chosen so that **floating-point numbers sort like integers**. Compare 1×2^{100} with 3×2^{20}: the first is far larger, even though its mantissa is smaller, because the exponent decides the scale and the mantissa only refines it. Putting the exponent in the high bits, above the mantissa, means that comparing two patterns as plain integers compares exponents first and mantissas only when the exponents are equal, which is exactly numeric order.

The biased exponent is what makes this work. In excess form a larger exponent is always a larger unsigned field, negative exponents included, so no two's complement sign gets in the way. The result: for two non-negative floats, the one whose bit pattern is the larger unsigned integer is the larger number, and the integer comparison hardware can sort floats. The quarter-precision table shows it: reading the patterns in counting order reads the values in increasing order. The sign bit comes first so that sign alone splits positive from negative; negative numbers are sign and magnitude, so among them the order reverses.

Encoding: decimal to IEEE

1. **Convert to binary scientific notation**, normalised: $\pm 1.xxxx_2 \times 2^e$ (fixed-point-numbers[6] for the fraction bits).
2. **Store the sign**: $+$ gives 0, $-$ gives 1.
3. **Store the biased exponent**: $e + 127$ in 8 bits (single) or $e + 1023$ in 11 bits (double).
4. **Store the mantissa**: drop the leading 1 and **pad with zeros on the right** to 23 (or 52) bits. Right, because these are fraction bits: zeros after the last 1 do not change the value.
5. **Group into nibbles** and write the 32 bits as 8 hex digits.

And **beware of special values**: zero, infinity and NaN do not follow these steps (below).

-23.25 worked through these steps to `0xC1BA0000` is in 2026-10-19-floating-point-numbers[7].

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

Worked: 5.75 to single precision

Write 5.75 as an IEEE single-precision number in hexadecimal.

1. $5.75 = 101.11_2 = 1.0111_2 \times 2^2$.
2. Positive: $S = 0$.
3. $E = 2 + 127 = 129 = 10000001_2$.
4. $M =$ `0111` followed by 19 zeros.
5. `0 10000001 01110000000000000000000` = `0100 0000 1011 1000 0000 0000 0000 0000` = `0x40B80000`.

Worked: -0.375 to single precision

Write -0.375 as an IEEE single-precision number in hexadecimal.

1. $0.375 = 0.011_2 = 1.1_2 \times 2^{-2}$. The point moves *right*, so the exponent is negative.
2. Negative: $S = 1$.
3. $E = -2 + 127 = 125 = 01111101_2$.
4. $M =$ `1` followed by 22 zeros.
5. `1 01111101 10000000000000000000000` = `1011 1110 1100 0000 0000 …` = `0xBEC00000`.

Worked: 0.1 to single precision

Write 0.1 as an IEEE single-precision number. Is it exact?

$0.1 = 0.0\,0011\,0011\,0011\ldots_2$ repeating (fixed-point-numbers[8]), which normalises to $1.1001\,1001\,1001\ldots_2 \times 2^{-4}$. So $S = 0$ and $E = -4 + 127 = 123 = 01111011_2$. The mantissa should be `1001` repeated for ever. Only 23 bits fit:

`10011001100110011001100` | `1100…` (the bits that do not fit)

What is cut off starts with a 1 followed by more 1s, so it is more than half a unit in the last place, and the stored mantissa **rounds up** to `10011001100110011001101`. The pattern is `0 01111011 10011001100110011001101` = `0x3DCCCCCD`, and it stands for 0.100000001490116 ..., not 0.1. **0.1 has no exact binary floating-point form**, for the same reason $\frac{1}{3}$ has no exact decimal one.

[8] courses/csse2010/fixed-point-numbers.pdf

Decoding: IEEE to decimal

1. Write the hex as 32 bits and **split 1 / 8 / 23**.
2. **Sign**: 0 is +, 1 is −.
3. **Exponent**: check for all 0s or all 1s (special values); otherwise $e = E - 127$.
4. **Mantissa**: put back the hidden 1, giving $1.M_2$.
5. Shift the point e places (right if $e > 0$) and convert.

`0x422F0000` decoded this way to 43.75 is in 2026-10-19-floating-point-numbers[9].

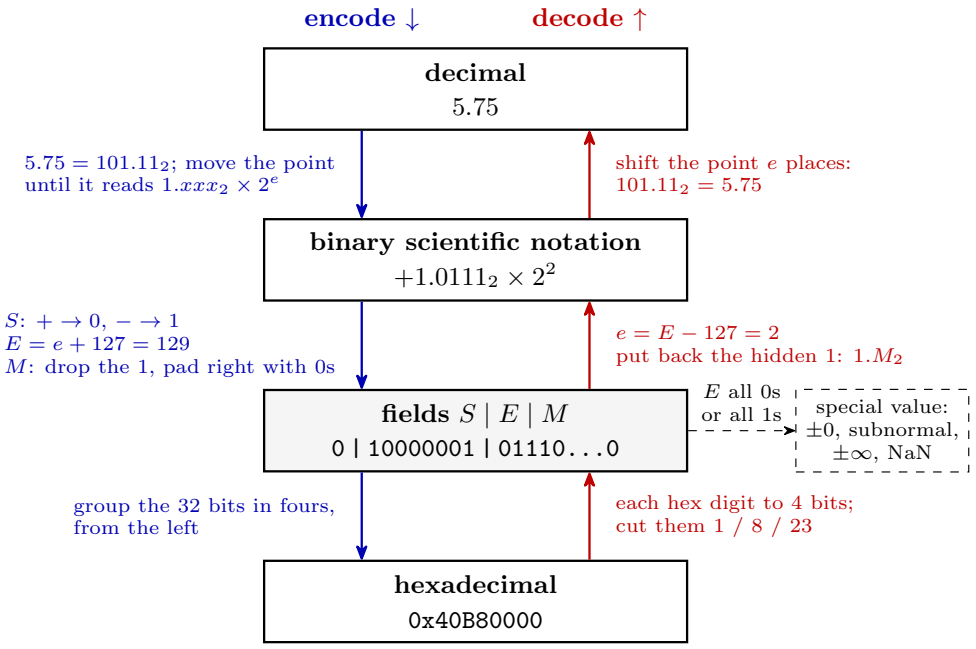


Figure 2: Encoding and decoding are the same four stages walked in opposite directions, shown with 5.75. Encoding (down the left) writes the value in binary and moves the point until one 1 is left in front of it; the sign becomes S, the power of 2 plus 127 becomes E, and the bits after the leading 1 become M, padded with zeros on the right; the 32 bits are then grouped in fours for hex. Decoding (up the right) undoes each step: expand every hex digit to four bits, cut the bits 1 / 8 / 23, take 127 off E and put the hidden 1 back, and shift the point. The one extra step is on the way up: an exponent field of all 0s or all 1s is not a normal number and has to be read as a special value instead.

[9] courses/csse2010/2026-10-19-floating-point-numbers.pdf

Worked: `0x41200000`

What decimal value does the single-precision pattern `0x41200000` hold?

`0100 0001 0010 0000 …` splits as $S = 0$, $E = 10000010_2 = 130$, $M = 0100\ldots$. So the value is $+1.01_2 \times 2^{130-127} = 1.01_2 \times 2^3 = 1010_2 = \mathbf{10}$.

Worked: two special patterns

What do `0x80000000` and `0x7F800000` hold?

- `0x80000000` is `1 00000000 000…0`: exponent all 0s and mantissa 0, with the sign set. It is -0, which compares equal to $+0$.
- `0x7F800000` is `0 11111111 000…0`: exponent all 1s and mantissa 0, sign clear. It is $+\infty$. Following steps 4 and 5 blindly would give 1.0×2^{128}, which is the mistake the special-value check is there to catch.

Special values

The smallest and largest exponent fields are reserved. What a pattern means then depends on whether the mantissa field is zero. The exponent here is the raw field as stored, with no bias taken off, so the reserved values are 0 (all 0s) and the maximum (all 1s, 255 in single precision):

Exponent field (as stored)	Mantissa = 0	Mantissa ≠ 0
0 → all 0s	±0	subnormal numbers
anything else	normal numbers	normal numbers
maximum → all 1s (255 for single, 2047 for double)	± infinity	± NaN (not a number)

This raw field is often called the *unbiased* exponent, in the sense that the bias has not yet been subtracted; the true exponent of a normal number is the field minus 127 (or 1023).

Zero

Zero cannot be normalised (it has no leading 1), so it gets its own pattern: **every exponent and mantissa bit 0**. Because the sign bit is separate there are two of them, $+0$ (`0x00000000`) and -0 (`0x80000000`). They compare equal; the sign survives only in things like $1/-0 = -\infty$.

stored exponent	**mantissa** $= 0$	**mantissa** $\neq 0$
all 0s 00000000	± 0	subnormal $\pm 0.M_2 \times 2^{-126}$
anything else 1 to 254	normal: $\pm 1.M_2 \times 2^{E-127}$	
all 1s 11111111	$\pm\infty$	$\pm$ NaN

single precision shown; double uses 11-bit fields, 2047 and 2^{E-1023}, 2^{-1022}

Figure 3: What a floating-point pattern means, decided by the stored exponent field (rows) and whether the mantissa field is zero (columns). Every exponent except all 0s and all 1s gives an ordinary normalised number, with the hidden 1 in front of the mantissa. All 0s gives zero, or a subnormal number with a hidden 0 instead of a 1 and the smallest normal exponent, which is 2 to the minus 126 in single precision. All 1s gives infinity, or NaN when any mantissa bit is set. The sign bit applies in every cell, so there are two zeros, two infinities, and NaNs of either sign.

Subnormal numbers

With an all-0s exponent and a non-zero mantissa, the number is **subnormal** (also called denormal):

$$(-1)^S \times 0.M_2 \times 2^{-126} \quad \text{(single precision)}$$

Two rules differ from normal numbers.

- **The hidden bit is 0, not 1.** This breaks the scientific-notation rule of one non-zero digit before the point, on purpose: it lets values smaller than the smallest normal number be held, with fewer significant bits as they shrink.
- **The exponent is 2^{-126}, the same as the smallest normal numbers**, not $2^{0-127} = 2^{-127}$ as the excess rule would give. With 2^{-126} the subnormals continue the smallest normal binade at exactly the same spacing, 2^{-149}, all the way down to 0, with no gap and no overlap. With 2^{-127} they would be spaced twice as finely and leave a gap below 2^{-126}.

The smallest normal single is $2^{-126} \approx 1.18 \times 10^{-38}$, and the smallest subnormal is $2^{-23} \times 2^{-126} = 2^{-149} \approx 1.4 \times 10^{-45}$. The effect is easiest to see in quarter precision (figure below).

Infinity

All-1s exponent, zero mantissa: $\pm\infty$. It is the result when

- a number is **too big to store**: anything past the largest finite value (about 3.4×10^{38} for single) **overflows** to ∞;
- a calculation is mathematically infinite: $1 \div 0 = +\infty$, $-\log 0 = +\infty$ ($\log 0 = -\infty$), $\tan(\pi/2)$.

On the last: in practice `tan(M_PI/2)` returns a large finite number, about 1.6×10^{16} in double, because $\pi/2$ itself cannot be stored exactly and the tangent of the stored value is merely huge.

Infinity behaves sensibly in further arithmetic: $\infty + 1 = \infty$, $1/\infty = 0$, $-\infty < \$$ every finite number.

NaN

All-1s exponent, non-zero mantissa: **NaN, not a number**. It is the result of an operation with **no meaningful answer**:

- $0 \div 0$, $0 \times \infty$, $\infty - \infty$;
- $\sqrt{-1}$, $\log(-1)$, $\sin^{-1} 20$ (no real angle has a sine of 20);

- 0^0, as an indeterminate form. IEEE 754's `powr(0, 0)` gives NaN; C's `pow(0, 0)` returns 1 by convention.

A NaN propagates: any arithmetic with a NaN operand gives NaN, so one bad step poisons everything computed from it, and the NaN at the end shows something went wrong. There are a few exceptions, functions whose answer does not depend on the missing value: `pow(NaN, 0)` is 1, and `fmax(NaN, 3)` is 3. Comparisons do not produce NaN at all; they return false. In particular **`x == x` is false when `x` is NaN**, which is the standard test for one (or `isnan(x)` in C). With 23 mantissa bits there are $2^{23} - 1$ different NaN patterns per sign, all meaning NaN.

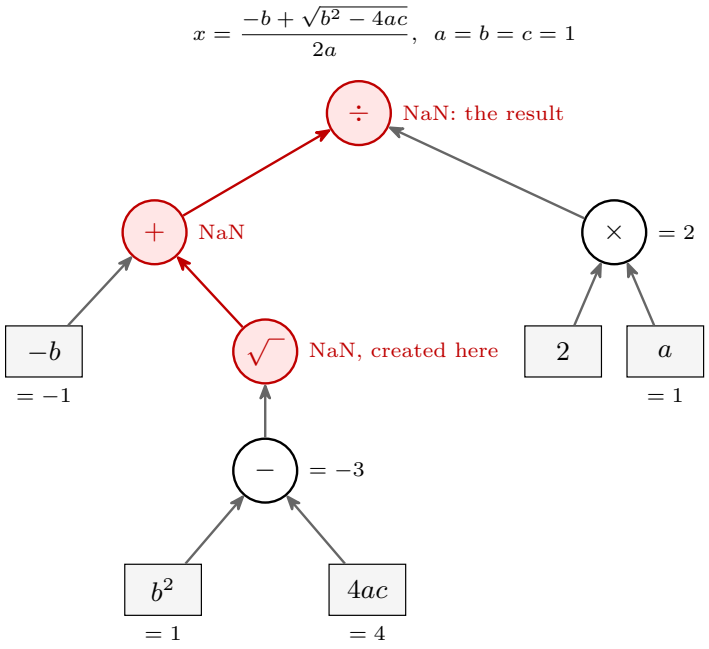


Figure 4: How one NaN poisons a calculation, in the quadratic formula with a = b = c = 1, which has no real root. Arrows show which results feed which operation. The discriminant b squared minus 4ac is -3, an ordinary number. Its square root has no real answer, so that operation creates a NaN. Every operation that then takes the NaN as an operand gives NaN too, so the sum and the final division are NaN. The branch that never touches it, 2a, is still 2. The NaN that comes out at the top is the signal that something went wrong further down. Comparisons are the exception: they do not give NaN but false, so result == result is false for a NaN.

Finite precision

A float has a fixed number of mantissa bits, so most real numbers **cannot be represented exactly** and are **rounded** to the nearest value that can (ties go to the neighbour whose last mantissa bit is 0, "round half to even"). The error is relative: single precision keeps 24 significant bits, about 7 decimal digits, wherever the point is; double keeps 53, about 16.

Rounding happens after **every** operation, and that changes the arithmetic rules.

Division and multiplication need not undo each other. $10 \div 3 = 3.333\ldots = 11.010101\ldots_2$, which repeats for ever and has to be cut short. In 3-digit decimal the same thing gives $10 \div 3 = 3.33$ and $3.33 \times 3 = 9.99 \neq 10$. Whether the binary version lands back on 10 depends on how the roundings fall: for $10 \div 3 \times 3$ they happen to cancel in single and double, but $(1 \div 41) \times 41 = 0.99999994$ in single, and $(1 \div 49) \times 49 = 0.9999999999999999$ in double. So $(x \div y) \times y$ **may not equal** x, and code should never test floats for exact equality after arithmetic; it should test whether they are within a tolerance.

The order of operations matters. This is often put loosely as floating point being *occasionally non-commutative*. Strictly, addition is still commutative ($a + b = b + a$ always); what fails is **associativity**: $(A + B) + C$ **need not equal** $A + (B + C)$, because the two orders round different intermediate results. In double, $(0.1 + 0.2) + 0.3 = 0.6000000000000001$ while $0.1 + (0.2 + 0.3) = 0.6$. The order of a long sum changes its answer.

Absorption

If $X \gg \delta$, then $(X + \delta) - X$ can be 0. Near X the representable values are spaced by about $X \times 2^{-23}$ (single). If δ is less than half that gap (or exactly half, when the tie goes to X), $X + \delta$ rounds straight back to X, and subtracting X leaves nothing: δ has been **absorbed**.

- Single precision: $2^{24} = 16\,777\,216$, and the gap there is 2. So $2^{24} + 1$ is a tie between $16\,777\,216$ and $16\,777\,218$, rounds to the even one, and $(2^{24} + 1) - 2^{24} = 0$. The same happens with $10^8 + 1 - 10^8$.
- It also breaks associativity: $(2^{24} + 1) + 1 = 2^{24}$, since each 1 is absorbed in turn, but $2^{24} + (1 + 1) = 2^{24} + 2$.

So adding many small numbers to a large total can lose them all; adding the small ones together first keeps them.

The formats compared

IEEE 754 defines a family of binary formats that differ only in field widths. Each follows the same rules: bias $2^{k-1} - 1$ for a k-bit exponent, hidden 1, the same special values. With k exponent bits and m mantissa bits:

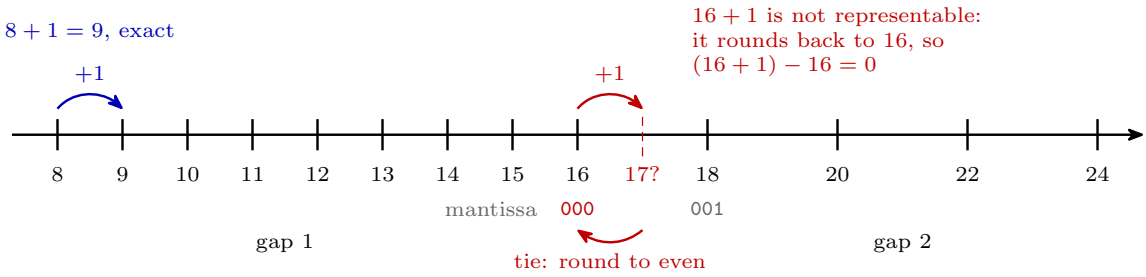


Figure 5: Absorption, shown on the quarter-precision values between 8 and 24. Between 8 and 16 neighbouring values are 1 apart, so $8 + 1 = 9$ is exact. Between 16 and 32 they are 2 apart, and $16 + 1 = 17$ falls exactly halfway between 16 and 18. The tie goes to the neighbour whose last mantissa bit is 0: 16 has mantissa field 000 and 18 has 001, so $16 + 1$ rounds back to 16 and $(16 + 1) - 16 = 0$. The same thing happens in any format whenever the number added is less than half the gap at the size of the larger number, or exactly half with the tie going back down.

- the **next value after 1** is $1 + 2^{-m}$, the precision near 1;
- the **largest finite value** is $(2 - 2^{-m}) \times 2^{\text{bias}}$: all-1s mantissa, the exponent field one below all 1s.

	Quarter	Half	Single	Double	Quadruple	Octuple
Total bits	8	16	32	64	128	256
Sign bits	1	1	1	1	1	1
Exponent bits	4	5	8	11	15	19
Mantissa bits	3	10	23	52	112	236
Bias	7	15	127	1023	16383	262143
Next after 1	$1 + \frac{1}{8} = 1.125$	$1 + \frac{1}{1024} \approx 1.001$	$1 + 2^{-23} \approx 1 + 1.2 \times 10^{-7}$	$1 + 2^{-52} \approx 1 + 2.2 \times 10^{-16}$	$1 + 2^{-112} \approx 1 + 1.9 \times 10^{-34}$	$1 + 2^{-236} \approx 1 + 9.1 \times 10^{-72}$
Maximum	240	65504	$\approx 3.4 \times 10^{38}$	$\approx 1.8 \times 10^{308}$	$\approx 1.2 \times 10^{4932}$	$\approx 1.6 \times 10^{78913}$
Use	ultra rare	rare	common	common	rare	ultra rare

Quarter precision is not one of the standard's formats. It is an 8-bit format built by the same rules (1 sign, 4 exponent, 3 mantissa bits, bias 7), small enough to list every value (below). Half precision appears in graphics and machine learning, where memory matters more than digits; single and double are what ordinary code uses.

The maximum grows with the **exponent** width and the precision with the **mantissa** width. Each extra exponent bit doubles the bias and so roughly squares the maximum: $2^8 \to 2^{16} \to 2^{128} \to 2^{1024}$ as the exponent goes 4, 5, 8, 11 bits. The quarter maximum is $1.875 \times 2^7 = 240$ and the half maximum $(2 - 2^{-10}) \times 2^{15} = 65504$.

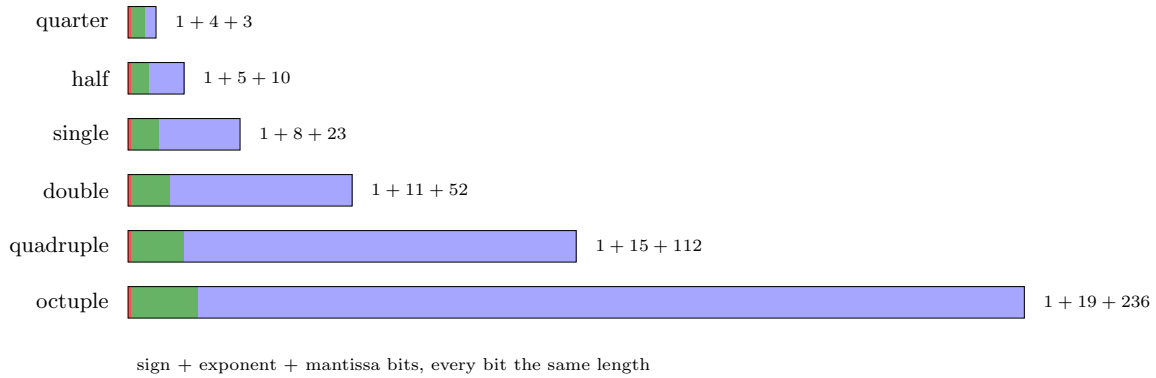


Figure 6: The six formats drawn to one scale, one bit per unit of length, with the sign in red, the exponent in green and the mantissa in blue. As the formats double in size, nearly all of the new bits go to the mantissa: the exponent only grows from 4 to 19 bits while the mantissa grows from 3 to 236. A few exponent bits already give an enormous range, because each one doubles the largest power of 2; the precision needs many more bits, because each mantissa bit only halves the gap between neighbouring values.

Every quarter-precision value

Quarter precision has only 256 patterns, so every value can be listed. Each row is a sign bit (0 here) and a 4-bit exponent field; each column is the 3-bit mantissa field. A normal row with stored exponent E holds $1.M_2 \times 2^{E-7}$. The first row (in italics) is the subnormals, $0.M_2 \times 2^{-6}$, and the last row is infinity and NaN. The negative values are the same table with the sign bit set.

Sign, exponent	…000	…001	…010	…011	…100	…101	…110	…111
0 0000	*0*	*0.001953125*	*0.00390625*	*0.005859375*	*0.0078125*	*0.009765625*	*0.01171875*	*0.013671875*
0 0001	0.015625	0.017578125	0.01953125	0.021484375	0.0234375	0.025390625	0.02734375	0.029296875
0 0010	0.03125	0.03515625	0.0390625	0.04296875	0.046875	0.05078125	0.0546875	0.05859375
0 0011	0.0625	0.0703125	0.078125	0.0859375	0.09375	0.1015625	0.109375	0.1171875
0 0100	0.125	0.140625	0.15625	0.171875	0.1875	0.203125	0.21875	0.234375
0 0101	0.25	0.28125	0.3125	0.34375	0.375	0.40625	0.4375	0.46875

Sign, exponent	…000	…001	…010	…011	…100	…101	…110	…111
0 0110	0.5	0.5625	0.625	0.6875	0.75	0.8125	0.875	0.9375
0 0111	1	1.125	1.25	1.375	1.5	1.625	1.75	1.875
0 1000	2	2.25	2.5	2.75	3	3.25	3.5	3.75
0 1001	4	4.5	5	5.5	6	6.5	7	7.5
0 1010	8	9	10	11	12	13	14	15
0 1011	16	18	20	22	24	26	28	30
0 1100	32	36	40	44	48	52	56	60
0 1101	64	72	80	88	96	104	112	120
0 1110	128	144	160	176	192	208	224	240
0 1111	Inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN

What the table shows:

- **Each row is a binade**, the values from one power of 2 up to the next, and it holds **eight equally spaced values**. The spacing doubles from each row to the next: $\frac{1}{8}$ between 1 and 2, $\frac{1}{4}$ between 2 and 4, and 16 between 128 and 240. The relative precision is the same in every row, which is the whole point of floating point.
- **Exponent field 0111 is the row of 1**, because 7 is the bias: $2^{7-7} = 1$.
- **Counting up through the patterns counts up through the values**, row by row (why this order).
- **The subnormal row has the same spacing as the row after it**, $\frac{1}{512}$, and reaches down to 0. That is the effect of giving subnormals the exponent 2^{-6} rather than 2^{-7}.
- **The largest finite value is 240**, and the next pattern up is infinity. Anything at or above 248, halfway from 240 to the next step, overflows.
- **Not every whole number is here.** 17, 19 and 33 are missing; between 16 and 32 only the even numbers can be held.

Worked: rounding in quarter precision

In quarter precision, what are $16 + 1$, $13 \div 3$, and then that quotient times 3?

- $16 + 1$**.** Between 16 and 32 the values are 2 apart. 17 is exactly halfway between 16 (mantissa 000) and 18 (mantissa 001); the tie goes to the even mantissa, **16**. The 1 has been absorbed.
- $13 \div 3 = 4.333\ldots$ Between 4 and 8 the values are 0.5 apart, and 4.333 is nearer 4.5 than 4. So the quotient is **4.5**.
- $4.5 \times 3 = 13.5$**.** Between 8 and 16 the values are 1 apart, and 13.5 is a tie between 13 (101) and 14 (110); it goes to the even mantissa, **14**. So $(13 \div 3) \times 3 = 14$, not 13.

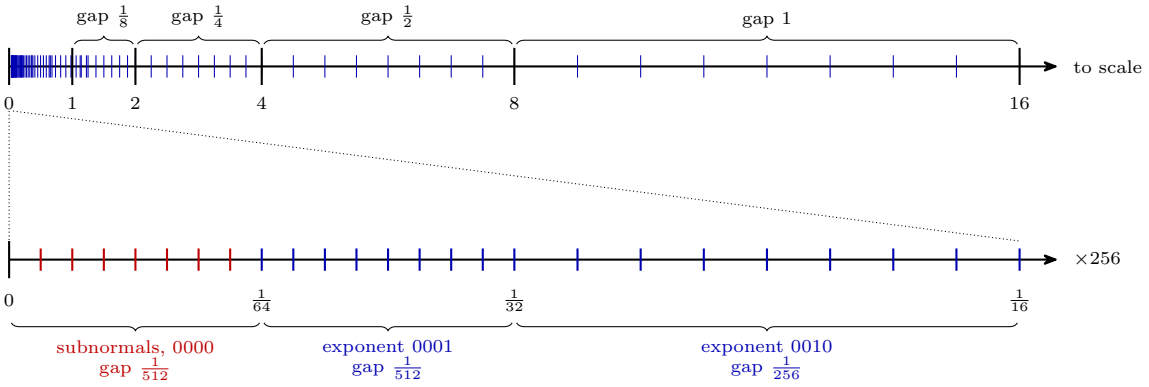


Figure 7: The positive quarter-precision values on a number line. Top: 0 to 16 to scale. Every binade from one power of 2 to the next holds eight evenly spaced values, so the gap doubles at each power of 2, from 1/8 between 1 and 2 to 1 between 8 and 16, and the values crowd together towards 0. Bottom: the stretch from 0 to 1/16 magnified 256 times, so that 1/16 lands directly under 16. The subnormals (exponent field 0000) fill the space below the smallest normal value, 1/64, with the same gap of 1/512 as the first normal binade, so the spacing carries on evenly down to 0 instead of leaving a hole.

Floating point in C and on the AVR

C has two floating-point types, `float` and `double`. They are **usually** IEEE single and double precision, but the C standard only sets minimums. Simplified:

- `float` must represent numbers up to 10^{37} while keeping **6 decimal digits** of accuracy;
- `double` must represent numbers up to 10^{37} while keeping **10 decimal digits**, and must not be less accurate than `float`.

Single precision meets the first easily (about 7 digits, up to 3.4×10^{38}), and double the second (about 16 digits).

On the AVR, avr-gcc makes `double` the same as `float`: 32-bit single precision. That is strictly against the standard, since single precision has about 7 digits, not 10. A program that moves to the AVR from a desktop and relies on `double` silently loses half its digits (c-language-basics[10]).

GCC also offers `_Float16` and `__float128`, for half and quadruple precision, on targets that support them. There is no C type for quarter or octuple precision.

[10] courses/csse2010/c-language-basics.pdf

Prefer integer arithmetic

The AVR has **no floating-point hardware**. Its only data type in hardware is the 8-bit integer (instruction-set-architecture[11]), so every `float` operation is a call to a library routine that unpacks the fields, aligns, adds or multiplies the mantissas with integer instructions, renormalises, rounds and checks for special values. Compare two lines that give the same result for every input:

```
PORTB = PINA / 3;        /* integer division:  about 8 instructions, no branches */
PORTB = PINA * 0.334f;   /* floating point:    300+ instructions, with branches   */
```

The second converts the byte to a `float`, calls the software multiply, and converts the result back, and each of those steps is a routine full of tests and loops. Its running time also varies with the input, which matters in an interrupt handler or a timing loop. The first needs no library at all: a compiler turns division by a constant into a multiplication by a fixed-point reciprocal and a shift. One way for a byte: $x \div 3 = (x \times 171) \gg 9$, since $171/512 = 0.33398\ldots$ is just above $\frac{1}{3}$, and this gives exactly $\lfloor x/3 \rfloor$ for every x from 0 to 255. (Checking all 256 inputs also confirms that truncating `PINA * 0.334f` gives the same answers.)

So on a small microcontroller, **work in integers or fixed-point-numbers[12] wherever the result can be the same**: scale a value so that the fraction becomes whole numbers (millivolts instead of volts, tenths of a degree), and divide or shift at the end.

Common mistakes

- **Normalising binary to $[1, 10)$.** In binary the mantissa must be in $[1, 2)$: 3.985×2^{77} is not normalised; 1.993×2^{78} is.
- **Storing the hidden 1.** The mantissa field holds only the bits *after* the point.
- **Using a bias of 128.** IEEE single is excess-127 and double excess-1023, one less than the 2^{k-1} of plain excess notation.
- **Forgetting the bias when decoding.** An exponent field of 132 means 2^5, not 2^{132}.
- **Padding the mantissa on the left.** The zeros go on the right; they are fraction bits.
- **Decoding special patterns as normal numbers.** Check the exponent for all 0s or all 1s first: `0x7F800000` is infinity, not 2^{128}.
- **Treating a subnormal's exponent as 2^{-127}.** It is 2^{-126}, with a hidden 0.
- **Testing floats for equality.** After any arithmetic, compare with a tolerance; `0.1` is not exactly 0.1 to begin with.
- **Assuming addition is associative.** The order of a sum can change its result, and a small term can vanish into a large one.
- **Expecting 64 bits from `double` on the AVR.** It is 32.

[11] courses/csse2010/instruction-set-architecture.pdf
[12] courses/csse2010/fixed-point-numbers.pdf

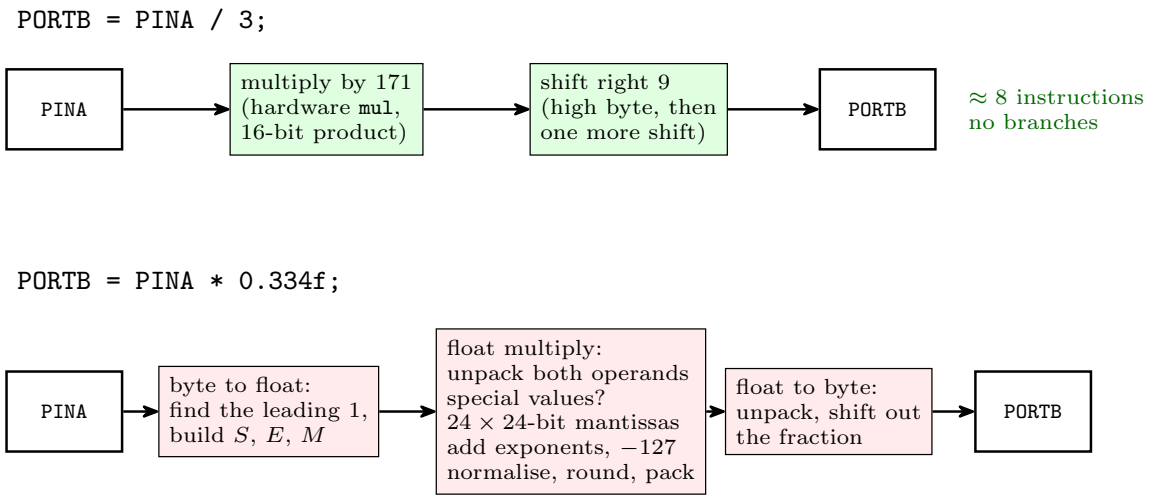


Figure 8: Why the two lines differ so much in cost on the AVR. Dividing a byte by 3 can be done with the hardware 8-by-8 multiply and a shift: multiply by 171 and shift right 9 places, which is the same as taking the high byte of the product and shifting it once more. That is a few instructions with no branches, taking the same time for every input. The float version has no hardware to use. The byte is first converted to a float, which means finding its leading 1 and building the sign, exponent and mantissa fields. The multiply is a library routine that unpacks both operands, tests for special values, multiplies the 24-bit mantissas with 8-bit multiplies, adds the exponents and takes off one bias, then normalises, rounds and packs the result. A third routine converts back to an integer. Each step is a loop or a chain of tests, so the time also depends on the value.

- **Using floats on the AVR where integers would do.** It costs hundreds of instructions per operation.

See also

- fixed-point-numbers[13] — the fixed binary point, $m.n$ formats, and the range/precision trade-off this escapes
- binary-number-representations[14] — excess notation, sign and magnitude, hex grouping
- c-language-basics[15] — `float` and `double` among C's types, and floating-point constants
- instruction-set-architecture[16] — the AVR's hardware data types, which do not include floating point
- 2026-10-19-floating-point-numbers[17] — Lecture 23

[13] courses/csse2010/fixed-point-numbers.pdf
[14] courses/csse2010/binary-number-representations.pdf
[15] courses/csse2010/c-language-basics.pdf
[16] courses/csse2010/instruction-set-architecture.pdf
[17] courses/csse2010/2026-10-19-floating-point-numbers.pdf