Lecture 23 — Floating Point Numbers

Table of contents

Goals . . . 2
The problem: how many molecules in a cup? . . . 2
Fractions in binary: fixed point . . . 3
Letting the point move . . . 6
Why "mantissa" . . . 7
IEEE 754 . . . 8
Conversion steps . . . 10
Finite precision . . . 11
Comparing float sizes . . . 12
Special values . . . 13
Not a number (NaN) . . . 15
Floating point in C and on the AVR . . . 15
See also . . . 16

How a computer stores fractions, and numbers far too large or too small for any sensible number of integer bits. The note starts from why that is needed, with the number of air molecules in a coffee cup; covers fixed point, where the binary point sits at an agreed place, and the trade-off between range and precision it forces; then lets the point float, by way of scientific notation, normalisation and the log-table origin of the word "mantissa"; and then works through the IEEE 754 single and double precision formats, converting in both directions, finite precision and what it does to arithmetic, the formats from quarter to octuple precision, the special values zero, subnormal, infinity and NaN, and floating point in C and on the AVR. The reference notes are fixed-point-numbers[1] and floating-point-numbers[2]. Whole-number formats, including the excess notation the exponent uses, are in binary-number-representations[3].

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

Goals

1. See why a fixed number of integer bits cannot hold both very large and very small quantities.
2. Represent fractions in fixed point: convert a value to an $m.n$ format and read a bit pattern in one, unsigned and two's complement, and state the range and precision of every 8-bit format.
3. Write numbers in normalised scientific notation, in decimal and in binary, and say where the word "mantissa" comes from.
4. Describe the IEEE 754 single and double precision formats: the fields, the biased exponent, the implicit leading 1, and why the fields are in that order.
5. Convert a decimal number to IEEE single precision in hexadecimal, and back.
6. Explain the limits of floating point: finite precision and its effect on arithmetic, the special values, and how the formats from quarter to octuple precision compare.
7. Know how C and the AVR handle floating point, and why integer arithmetic is preferred on the AVR.

The problem: how many molecules in a cup?

Worked: air molecules in an empty coffee cup

How many air molecules are in an empty 12 fl oz coffee cup at 1 atm and 26.85 °C?

The ideal gas law, $pV = nRT$, gives the amount of gas n in moles. In SI units, rounded to keep the arithmetic simple:

Quantity	Given	SI, rounded
pressure p	1 atm	100 000 Pa
volume V	12 fl oz	0.00036 m^3
gas constant R		8 $J/(K \cdot mol)$
temperature T	26.85 °C	300 K

$$n = \frac{pV}{RT} = \frac{100\,000 \times 0.00036}{8 \times 300} = \frac{36}{2400} = 0.015 \text{ mol}.$$

Molecules $= n\times$ Avogadro's number, $N_A = 6.02214076\times10^{23} =$ `602214076000000000000000`:

$$0.015 \times 602\,214\,076\,000\,000\,000\,000\,000 = 9\,033\,211\,140\,000\,000\,000\,000 \approx 9.03 \times 10^{21}.$$

(The roundings are generous: 1 atm is 101 325 Pa, 12 US fl oz is 0.000355 m^3 and R is 8.314. With those, $n \approx 0.0144$ mol and the count is about 8.7×10^{21}. Either way only the first two or three digits mean anything.)

What this says about storing numbers. Avogadro's number needs 79 bits as an integer, and the answer 73. The volume, 0.00036, has its first 1 only in the 12th place after the binary point ($2^{-12} \approx 0.000244$), so it needs at least 12 fractional bits to register at all. A format that held all three exactly would need about 80 integer bits and a few dozen fractional bits, almost all of them zero in any one value, and every digit past the third would be noise anyway. What is worth storing is **the significant digits and the scale**, separately. That is floating point, and the stepping stone to it is fixed point.

Fractions in binary: fixed point

Place values continue past the binary point by halving: $2^{-1} = \frac{1}{2}$, $2^{-2} = \frac{1}{4}$, $2^{-3} = \frac{1}{8}$, $2^{-4} = \frac{1}{16}$. **Fixed-point notation** chooses a place for the point, arbitrarily, and keeps it there. The point is never stored; it is an agreement about how to read the bits.

2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}
8	4	2	1	.	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$

A format with m integer bits and n fractional bits is called $m.n$. The full treatment is fixed-point-numbers[4].

Worked: representing 4.375

How is 4.375 represented in 8-bit fixed point, with 4, 5 or 3 integer bits?

Whole part: $4 = 100_2$.

Fraction, by repeated doubling: $0.375 \times 2 = 0.75$ (bit 0), $0.75 \times 2 = 1.5$ (bit 1), $0.5 \times 2 = 1.0$ (bit 1), and the fraction is now 0. Reading the bits in the order they came: $0.375 = 0.011_2$. As a check by subtraction, $0.375 - 0.25 = 0.125 = \frac{1}{8}$, so the $\frac{1}{4}$ and $\frac{1}{8}$ bits are set.

Or all at once: with 3 fractional bits, every value is a whole number of eighths. $4.375 \times 8 = 35 = 100011_2$; put the point back three places from the right: 100.011_2.

So $4.375 = 100.011_2$, and in each format:

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

Format	Bits
4.4	`0100.0110`
5.3	`00100.011`
3.5	`100.01100`

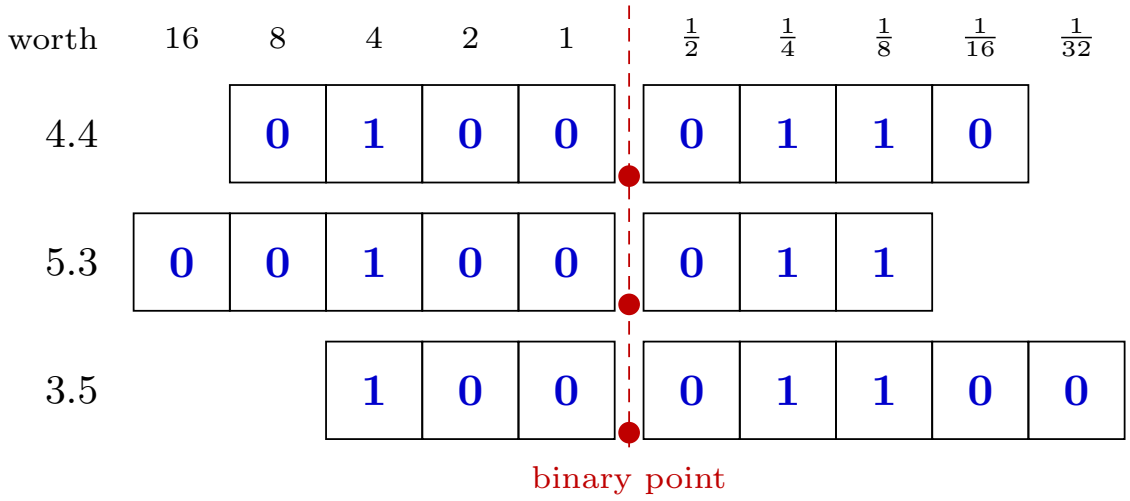


Figure 1: The value 4.375 in three 8-bit fixed-point formats. The significant bits 1, 0, 0, 0, 1, 1 are the same every time; each lines up with the same place values, 4, 1/4 and 1/8, wherever the point is put. Moving the point one place left therefore shifts the bits one place left in the byte, a 0 dropping off the left end and a 0 joining on the right. The format decides which values fit: 4.375 needs at least 3 integer bits and 3 fractional bits, and all three formats here have that.

Worked: `10110101` as unsigned 5.3

What does the 8-bit number `10110101` represent in unsigned fixed-point notation with 5 integer bits and 3 fractional bits?

Split it 5 and 3: `10110.101`.

- Integer bits `10110` $= 16 + 4 + 2 = 22$.
- Fraction bits `.101` $= \frac{1}{2} + \frac{1}{8} = 0.625$.

Answer: 22.625. Check: read as a whole number the byte is 181, and $181/2^3 = 22.625$.

Worked: `10110101` as two's complement 5.3

> What does the same `10110101` represent in two's complement fixed-point notation with 5 integer bits and 3 fractional bits? Choices: -9.375, -10.625, -9.625.

In two's complement the MSB's weight is negative (binary-number-representations[5]). In 5.3 the MSB is worth $-2^4 = -16$, and every other bit keeps its positive weight, **fraction bits included**:

$$\underbrace{-16}_{1} + \underbrace{0}_{0} + \underbrace{4}_{1} + \underbrace{2}_{1} + \underbrace{0}_{0} \;.\; \underbrace{\tfrac{1}{2}}_{1} + \underbrace{0}_{0} + \underbrace{\tfrac{1}{8}}_{1} = -10 + 0.625 = -9.375.$$

Answer: -9.375**.** Two checks:

- Unsigned it was 22.625. A set MSB in two's complement subtracts $2^5 = 32$ from the unsigned reading: $22.625 - 32 = -9.375$.
- Negate it: invert every bit, `01001.010` $= 9.25$, and add 1 in the last place, $\frac{1}{8}$: 9.375. So the original is -9.375.

Where the other two come from:

- -10.625 reads the integer bits correctly as -10 and then attaches the fraction as if it shared the sign, $-(10+0.625)$. But the fraction bits have positive weight: $-10+0.625 = -9.375$.
- -9.625 inverts only the integer bits (`10110` → `01001` $= 9$) and keeps `.101` as it is. That is neither rule: negating needs every bit inverted and then a 1 added in the last place.

bits	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**1**	
unsigned 5.3	16	8	4	2	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$16+4+2+\frac{1}{2}+\frac{1}{8} = \mathbf{22.625}$
two's complement 5.3	-16	8	4	2	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$-16+4+2+\frac{1}{2}+\frac{1}{8} = \mathbf{-9.375}$
									difference $32 = 2 \times 16$

Figure 2: One byte, 10110101, read in two 5.3 formats. The weights are identical except for the MSB, which is worth +16 unsigned and -16 in two's complement. Summing the weights of the 1 bits (the weights of the 0 bits are greyed out) gives 22.625 unsigned and -9.375 two's complement, and the two differ by exactly 32, twice the MSB's weight. The fraction bits add 1/2 and 1/8 in both readings: a two's complement fraction is never subtracted.

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

Range against precision

Every 8-bit format has the same 256 patterns; the format only spaces them. For unsigned $m.n$ the **precision** is the last bit's weight, 2^{-n}, and the **maximum** is all ones, 255×2^{-n}:

Format	Maximum	Precision	Format	Maximum	Precision
8.0	255	1	4.4	15.9375	0.0625
7.1	127.5	0.5	3.5	7.96875	0.03125
6.2	63.75	0.25	2.6	3.984375	0.015625
5.3	31.875	0.125	1.7	1.9921875	0.0078125
“9.$-1"|510|2||1020|0.99609375|4|0.00390625||"10.-$2”					

8.0 is `xxxxxxxx.` (an ordinary byte), 7.1 is `xxxxxxx.x`, 6.2 is `xxxxxx.xx`, and so on. **Each place the point moves left halves both the maximum and the step.** The formats in quotes go the other way: in “9.$-1"*thepointsitsoneplace* $*$ *beyond* $*$ *thebyte*,` *xxxxxxxx*0.`, *sothelowestoredbitisworth*2*andonlyevennumbersupto*510*exist*; "10.-$2” is `xxxxxxxx00.`, multiples of 4 up to 1020. No choice of point gets range and precision together. Two's complement ranges, and a drawing of the point sliding, are in fixed-point-numbers[6].

Letting the point move

Natural language and scientific notation

People handle the same problem with a **variable decimal point**. In words, **SI prefixes** move it: kilometre, milligram, femtometre, yoctogram, each prefix a power of 10. In symbols, **scientific notation** writes significant digits times a power of 10: Avogadro's number is 6.02×10^{23}, and the cup's 0.00036 m^3 is 3.6×10^{-4} m^3. A few digits plus an exponent covers any size.

Normalisation

The same value can be written 602×10^{21}, 60.2×10^{22}, 6.02×10^{23} or 0.602×10^{24}. **Normalised** means one non-zero digit before the point, the digits in $[1, 10)$, so 6.02×10^{23} is the normalised one: 602×10^{21} and 60.2×10^{22} have too many digits before the point, and 0.602×10^{24} has a zero there.

In binary the range is $[1, 2)$: the only non-zero binary digit is 1, so a normalised binary number is always $1.xxx_2 \times 2^e$.

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

Worked: which two are correctly normalised in binary?

Which two of these values have the correct binary normalisation?

A. 15.940×2^{75} B. 7.970×2^{76} C. 3.985×2^{77} D. 1.993×2^{78} E. 0.996×2^{79}

F. 11.796×2^{-15} G. 5.898×2^{-14} H. 2.949×2^{-13} I. 1.475×2^{-12} J. 0.737×2^{-11}

Answer: D and I, the only two with the multiplier in $[1, 2)$.

The two rows are each one number written five ways: halving the multiplier and adding 1 to the exponent leaves the value unchanged. A to E are all $\approx 6.02 \times 10^{23}$, Avogadro's number; F to J are all $\approx 3.60 \times 10^{-4}$, the volume of the cup in m^3 (to the three decimals shown). Exactly one spelling of each is normalised.

Option	Multiplier	Normalised in binary?	Why not
A	15.940	×	≥ 2: shift three more places
B	7.970	×	≥ 2: shift two more places
C	3.985	×	≥ 2: shift one more place
D	**1.993**	✓	
E	0.996	×	< 1: a leading 0
F	11.796	×	≥ 2
G	5.898	×	≥ 2
H	2.949	×	≥ 2
I	**1.475**	✓	
J	0.737	×	< 1: a leading 0

The trap is the decimal rule. B, C, G and H have one non-zero digit before the point, so they look normalised by the $[1, 10)$ rule. But the base here is 2, and with base 2 the multiplier must be in $[1, 2)$. E and J fail the other way: their leading digit is 0.

Why "mantissa"

The digits part of a floating-point number is called the **mantissa**, a name taken from **logarithm tables**. Logarithms turn multiplication into addition, $\log(a \times b) = \log a + \log b$, and before calculators that was done with printed tables. A standard four-figure table gives the logarithms of 1.00 to 9.99: row 12, column 3 reads `0899`, meaning $\log_{10} 1.23 \approx 0.0899$.

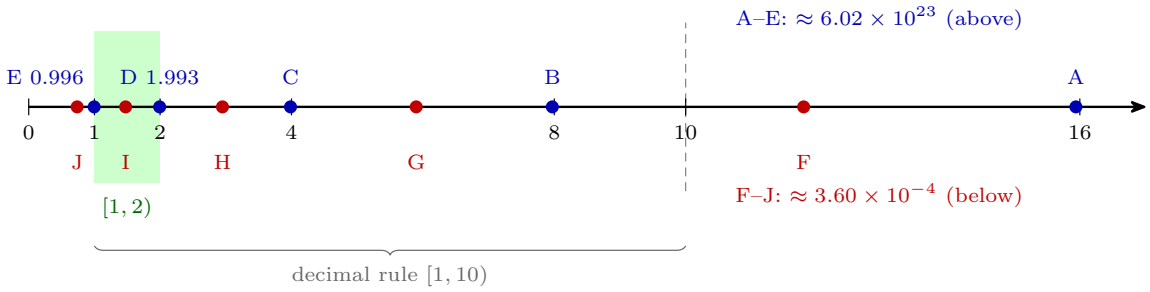


Figure 3: The ten multipliers on a number line, Avogadro's number written five ways above the line and the cup's volume five ways below. Each step left halves the multiplier and adds 1 to the power of 2, so the value stays the same. A binary normalised multiplier must lie in [1, 2), the shaded window, and exactly one spelling of each number lands in it: D, 1.993, and I, 1.475. E, at 0.996, sits just outside the left edge, and D just inside the right one. The wider window [1, 10) of decimal normalisation would wrongly admit B, C, G and H as well.

Worked: three logarithms from one table entry

Given $\log_{10} 1.23 \approx 0.0899$, what are $\log_{10} 12.3$, $\log_{10} 123$ and $\log_{10} 0.123$?

Each is 1.23 times a power of 10, and the log of a product is the sum of the logs:

- $\log_{10} 12.3 = \log_{10}(1.23 \times 10) = 0.0899 + 1 = \mathbf{1.0899}$
- $\log_{10} 123 = \log_{10}(1.23 \times 10^2) = 0.0899 + 2 = \mathbf{2.0899}$
- $\log_{10} 0.123 = \log_{10}(1.23 \times 10^{-1}) = 0.0899 - 1 = \mathbf{-0.9101}$

The fractional part, 0.0899, is the same every time and depends only on the digits 1, 2, 3: that was the **mantissa**. The integer part, 1, 2 or −1, only places the point: that was the **characteristic**. (For 0.123, tables kept the mantissa visible by writing $\bar{1}.0899$, meaning $-1 + 0.0899$.) So the table only needs mantissas, and the power of 10 is handled by adding a whole number. A floating-point number splits the same way, digits and an exponent, and the digits took the old name.

IEEE 754

The standard formats store a sign bit S, an exponent field E and a mantissa field M, and mean

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

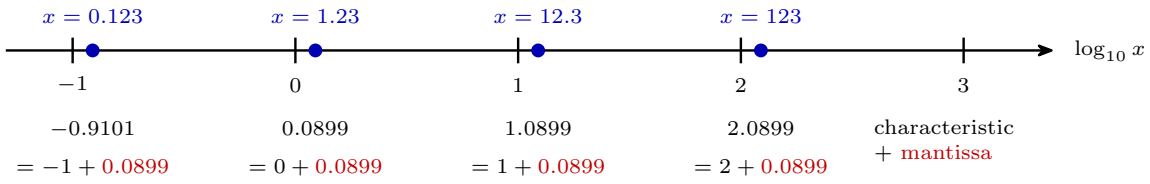


Figure 4: Logarithms of 0.123, 1.23, 12.3 and 123 on a log scale. Each is ten times the one before, so each sits exactly one unit further along. Every logarithm splits into a whole number, the characteristic, plus the same fractional part, 0.0899, which depends only on the digits 1, 2, 3; that fractional part was called the mantissa. The characteristic only records the power of 10, which is to say where the point goes.

- **Single precision, 32 bits**: 1 sign bit, 8 exponent bits in **excess-127** format, 23 mantissa bits.
- **Double precision, 64 bits**: 1 sign bit, 11 exponent bits in **excess-1023** format, 52 mantissa bits.
- **The mantissa is normalised**, so it begins with 1, and that 1 is **implicit**: it is not stored, which buys a 24th (or 53rd) significant bit. The exception is the subnormal numbers (below), where the hidden bit is 0.

Excess-127 means the field holds the true exponent plus 127. It is the excess notation of binary-number-representations[7], but with a bias one less than $2^7 = 128$. The field layouts are drawn in floating-point-numbers[8].

Worked: why sign, then exponent, then mantissa?

Why are the fields in the order sign, exponent, mantissa?

So that floats compare like integers. Take 1×2^{100} and 3×2^{20}. The first is far bigger despite the smaller leading number, because the exponent fixes the scale and the mantissa only refines it within that scale. Normalised, they are $1.0_2 \times 2^{100}$ and $1.1_2 \times 2^{21}$, with exponent fields $100 + 127 = 227$ and $21 + 127 = 148$. With the exponent in the high bits, comparing the two 32-bit patterns as unsigned integers compares 227 with 148 first and gets the right answer without looking at the mantissas. The mantissa only decides between numbers with equal exponents, which is exactly when it should.

The bias helps: in excess form larger exponents, negative ones included, are always larger unsigned fields. And the sign at the very top separates negatives from positives first. So the integer comparison circuits can sort non-negative floats directly.

[7] courses/csse2010/binary-number-representations.pdf
[8] courses/csse2010/floating-point-numbers.pdf

Conversion steps

1. Convert to **binary scientific notation**, normalised.
2. Store the **sign**: $+ \to 0$, $- \to 1$.
3. Store the **biased exponent**: true exponent $+127$.
4. Store the **mantissa**: drop the implicit 1 and **right-pad with zeros** to 23 bits.

Then group into nibbles for hex. And **beware of special values**: zero, infinity and NaN have reserved patterns.

Worked: -23.25 to IEEE single precision

> Convert -23.25 to an IEEE single-precision floating-point number, expressed in hexadecimal.

1. Binary scientific notation. $23 = 10111_2$ and $0.25 = 0.01_2$, so $23.25 = 10111.01_2$. Move the point four places left: $10111.01_2 = 1.011101_2 \times 2^4$.

2. Sign. Negative, so $S = 1$.

3. Exponent. $4 + 127 = 131 = 10000011_2$.

4. Mantissa. Drop the leading 1: `011101`, then pad with zeros on the right to 23 bits: `01110100000000000000000`.

Assemble and group:

S	Exponent	Mantissa
1	10000011	01110100000000000000000

`1100 0001 1011 1010 0000 0000 0000 0000` = `C 1 B A 0 0 0 0`, so -23.25 = `0xC1BA0000`.

Checks: $+23.25$ differs only in the sign bit, `0x41BA0000`. In double precision the exponent is $4 + 1023 = 1027$ and the pattern `0xC037400000000000`; in half precision (5-bit exponent, bias 15) it is `0xCDD0`.

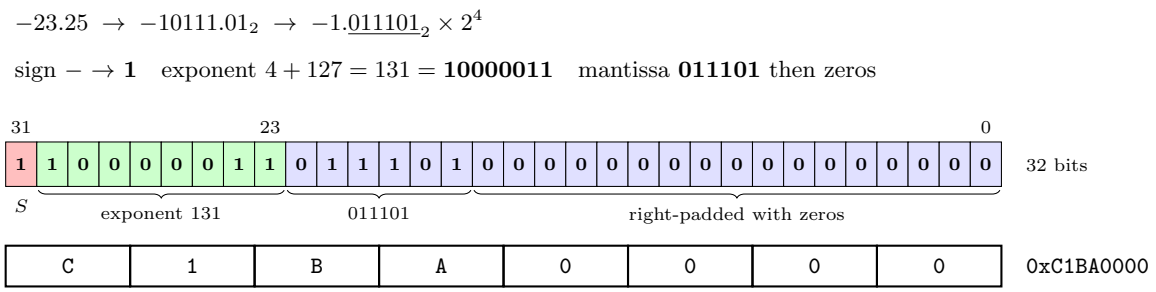


Figure 5: Encoding -23.25 in single precision. The minus sign becomes a 1 in the sign bit. The value 10111.01 in binary is normalised to 1.011101 times 2 to the 4, the 4 is stored as 4 + 127 = 131, and the bits after the leading 1 fill the mantissa field from the left, with zeros padding it out on the right. The 32 bits then group into nibbles from the left, giving C1BA0000 in hexadecimal. The nibble boundaries do not line up with the field boundaries: the C takes the sign and the first three exponent bits.

Worked: `0x422F0000` to decimal

Convert `0x422F0000` from an IEEE floating-point number to a decimal value.

Bits. `4 2 2 F 0 0 0 0` = `0100 0010 0010 1111 0000 0000 0000 0000`.

Split 1 / 8 / 23.

S	Exponent	Mantissa
0	10000100	01011110000000000000000

- **Sign** 0: positive.
- **Exponent** $10000100_2 = 132$, not all 0s or all 1s, so normal; true exponent $132 - 127 = 5$.
- **Mantissa**: put back the hidden 1: 1.0101111_2.

Value. $+1.0101111_2 \times 2^5$: move the point five places right, $101011.11_2 = 32+8+2+1+\frac{1}{2}+\frac{1}{4} =$ **43.75**.

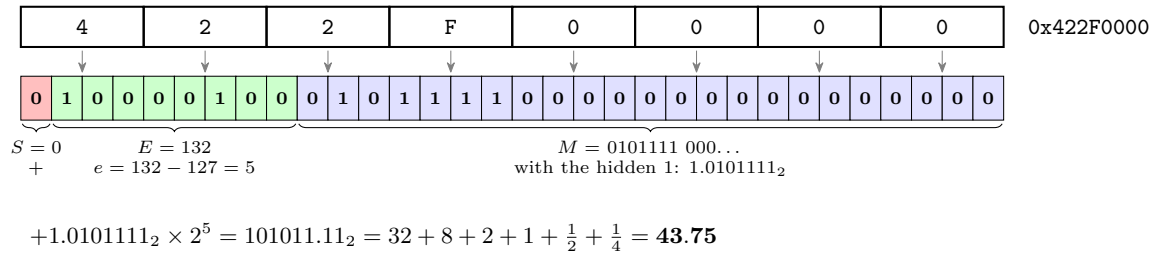


Figure 6: Decoding 0x422F0000. Each hex digit expands to four bits, and the 32 bits are then cut at the field boundaries, which fall in the middle of the first three hex digits. The sign bit 0 means positive. The exponent field 10000100 is 132, so the true exponent is 132 - 127 = 5. The mantissa field gets its hidden 1 back in front, giving 1.0101111, and shifting the point five places right gives 101011.11, which is 43.75.

Finite precision

Not every number can be represented precisely. A float has a fixed number of mantissa bits, so a value that needs more is rounded, and rounding happens again after every operation.

$10 \div 3 = 3.333\ldots = 11.010101\ldots_2$, with `01` repeating for ever, so the stored quotient is cut short and is not exactly $\frac{10}{3}$. In 3-digit decimal the effect is plain: $10.0 \div 3 = 3.33$, and $3.33 \times 3 = 9.99$.

In binary it depends on how the two roundings fall. For this particular case they cancel, and single and double precision both give exactly 10.0. But nothing guarantees it: in single precision $(1 \div 41) \times 41 = 0.99999994$, in double $(1 \div 49) \times 49 = 0.9999999999999999$, and in quarter precision $(13 \div 3) \times 3 = 14$ (floating-point-numbers[9]). So $(10.0 \div 3) \times 3$ **may not equal** 10.0, and floats should be compared with a tolerance, never with `==` after arithmetic.

Worked: does the order of addition matter?

> Can $(A + B) + C$ differ from $A + (B + C)$?

Yes. The order in which operations are done can change the answer, which is often summed up loosely as floating point being **occasionally non-commutative**. Strictly, the law that fails is **associativity**: $A + B$ always equals $B + A$, but grouping the terms differently rounds different intermediate sums. In double precision:

- $(0.1 + 0.2) + 0.3 = 0.6000000000000001$
- $0.1 + (0.2 + 0.3) = 0.6$

Worked: when is $(X + \delta) - X = 0$?

> If $X \gg \delta$, what is $(X + \delta) - X$?

It **can be zero**. Near X the representable values are about $X \times 2^{-23}$ apart in single precision. If δ is under half that gap, $X + \delta$ rounds back to X and δ is lost; at exactly half it is a tie, which goes to whichever neighbour has a 0 as its last mantissa bit. With $X = 2^{24} = 16\,777\,216$, where the gap is 2, and $\delta = 1$: $X + 1$ is a tie between X and $X + 2$, goes to X (the even one), and $(X + 1) - X = 0$. It also gives a second non-associative sum: $(X + 1) + 1 = X$, but $X + (1 + 1) = X + 2$. The same thing in quarter precision is drawn in floating-point-numbers[10].

Comparing float sizes

Every IEEE binary format has the same structure; only the widths change. With m mantissa bits the next value after 1 is $1 + 2^{-m}$, and the largest finite value is $(2 - 2^{-m}) \times 2^{\text{bias}}$.

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

	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
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 a standard format; it follows the same rules at 8 bits (bias 7) so that every value can be listed. The exponent width sets the range and the mantissa width the precision, and as the formats grow almost all the extra bits go to the mantissa. Each extra exponent bit roughly squares the maximum: $240 \approx 2^8$ with 4 bits, $65504 \approx 2^{16}$ with 5, $3.4 \times 10^{38} \approx 2^{128}$ with 8, $1.8 \times 10^{308} \approx 2^{1024}$ with 11. A to-scale drawing of the six layouts is in floating-point-numbers[11].

Special values

Two exponent fields are reserved. The exponent in this table is the field exactly as stored, with no bias taken off: 0 means the bits are all 0s, and the maximum, all 1s, is 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 precision)	± 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.

- **Zero** is zero: all exponent and mantissa bits 0. It cannot be normalised, so it needs its own pattern, and with the separate sign bit there is a +0 and a −0.

[11] courses/csse2010/floating-point-numbers.pdf

- **Subnormal numbers** have an **implicit leading 0** instead of 1, which breaks the scientific-notation convention on purpose so that values below the smallest normal number can be held. They use the **same exponent as the smallest normal numbers**, 2^{-126} in single precision, not the $2^{0-127} = 2^{-127}$ the bias would suggest; that way they continue the smallest normal binade at the same spacing down to 0.
- **Infinity**: numbers too big to store (past about 3.4×10^{38} in single), and results such as $1 \div 0$, $-\log 0$ and $\tan(\pi \div 2)$.
- **NaN**: "bad" calculations with no meaningful result.

Worked: every positive quarter-precision float

> List every positive value of the quarter-precision format (1 sign, 4 exponent, 3 mantissa bits, bias 7).

Rows are the sign bit and exponent field, columns the mantissa field. A normal row E holds $1.M_2 \times 2^{E-7}$; row `0000` (italics) holds the subnormals, $0.M_2 \times 2^{-6}$; row `1111` holds infinity and NaN.

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
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

Three cells worked by hand:

- `0 0111 000`: $E = 7$, $1.000_2 \times 2^0 = 1$. Exponent field 7 is the bias, so this row runs from 1 to 1.875 in steps of $\frac{1}{8}$, which is where "next after 1 = 1.125" comes from.
- `0 1110 111`: $1.111_2 \times 2^{14-7} = 1.875 \times 128 = 240$, the maximum.

- `0 0000 001`: subnormal, $0.001_2 \times 2^{-6} = \frac{1}{8} \times \frac{1}{64} = \frac{1}{512} = 0.001953125$, the smallest positive value. The step along this row, $\frac{1}{512}$, is the same as the step along row `0001`.

Each normal row is twice the one above it, so the gap between neighbours doubles every row while the relative precision stays the same; and counting up through the bit patterns counts up through the values. The number-line picture is in floating-point-numbers[12].

Not a number (NaN)

NaN represents the result of an operation that **has no solution**:

- $0 \div 0$, $0 \times \infty$, $\infty - \infty$;
- $\sqrt{-1}$, $\log(-1)$, $\sin^{-1} 20$;
- 0^0, an indeterminate form (IEEE's `powr(0, 0)` gives NaN, though C's `pow(0, 0)` returns 1).

Once a NaN appears it **propagates as a NaN through subsequent operations**: NaN + 1, NaN × 0 and $\sqrt{\text{NaN}}$ are all NaN, so the final result shows that something went wrong earlier. The exceptions are the few functions whose answer does not depend on the missing value, such as `pow(NaN, 0) = 1`, and comparisons, which return false instead: `x == x` is false for a NaN, which is how one is detected.

Floating point in C and on the AVR

- **`float` and `double` are usually single and double precision**, but C only sets minimums. Simplified, `float` must represent numbers up to 10^{37} with **6 decimal digits** of accuracy, and `double` numbers up to 10^{37} with **10 decimal digits**, and not less accurately than `float`.
- **In AVR C, `double` is the same as `float`**, 32-bit single precision. That falls short of the standard, since single precision gives only about 7 digits.
- **GCC provides `_Float16` and `__float128`** for half and quadruple precision. There is no type for quarter or octuple precision.
- **Integer calculations with equivalent results are preferable.** The AVR has no floating-point hardware, so every float operation is a library routine.

```
PORTB = PINA / 3;        /* 8 branchless instructions          */
PORTB = PINA * 0.334f;   /* 300+ instructions, with branches   */
```

[12] courses/csse2010/floating-point-numbers.pdf

Both give the same `PORTB` for all 256 values of `PINA` (checked exhaustively). The first divides by a constant, which a compiler can do with an integer multiply and a shift, for example $(x \times 171) \gg 9$, which equals $\lfloor x/3 \rfloor$ for every byte. The second converts `PINA` to a `float`, runs the software multiply, and converts back, each step a routine that unpacks fields, normalises, rounds and tests for special cases. More in floating-point-numbers[13] and c-language-basics[14].

See also

- fixed-point-numbers[15] — the binary point, $m.n$ formats, two's complement fixed point, range against precision
- floating-point-numbers[16] — IEEE 754 in full, encoding and decoding, special values, finite precision, the quarter-precision number line, floats on the AVR
- binary-number-representations[17] — place values, two's complement, excess notation, hex grouping
- c-language-basics[18] — `float` and `double` among C's types
- instruction-set-architecture[19] — the AVR's only hardware data type, the 8-bit integer
- 2026-10-15-compilation-and-linking[20] — Lecture 22 · 2026-10-22-memory-disks-and-files[21] — Lecture 24

[13] courses/csse2010/floating-point-numbers.pdf
[14] courses/csse2010/c-language-basics.pdf
[15] courses/csse2010/fixed-point-numbers.pdf
[16] courses/csse2010/floating-point-numbers.pdf
[17] courses/csse2010/binary-number-representations.pdf
[18] courses/csse2010/c-language-basics.pdf
[19] courses/csse2010/instruction-set-architecture.pdf
[20] courses/csse2010/2026-10-15-compilation-and-linking.pdf
[21] courses/csse2010/2026-10-22-memory-disks-and-files.pdf