IEEE 754 Converter

Home
Utilities
IEEE754
IEEE754 (en)
IEEE754 (de)
Contact

This page allows you to convert between the decimal representation of numbers (like "1.02") and the binary format used by all modern CPUs (IEEE 754 floating point). The conversion is limited to single precision numbers (32 Bit). The purpose of this webpage is to help you understand floating point numbers.

IEEE 754 Converter (JavaScript), V0.13

Note: This JavaScript-based version is still under development, please report errors here.

	Sign	Exponent	Mantissa
Value:	+1	2^{62}	1.9004863500595093
Encoded as:	0	189	7553827
Binary:	☐	☑☐☑☑☑☑☐☐	☑☑☑☐☐☑☑☐☑☐☐☐☐☑☐☐☑☑

Decimal Representation	8.7644463E18
Binary Representation	01011110111100110100001100100011
Hexadecimal Representation	0x5ef34323
After casting to double precision	8.7644463287815373E18

Usage: You can either convert a number by choosing its binary representation in the button-bar, the other fields will be updated immediately. Or you can enter a binary number, a hexnumber or the decimal representation into the corresponding textfield and press return to update the other fields. To make it easier to spot eventual rounding errors, the selected float number is displayed after conversion to double precision.

Special Values: You can enter the words "Infinity", "-Infinity" or "NaN" to get the corresponding special values for IEEE-754. Please note there are two kinds of zero: +0 and -0.

Conversion: The value of a IEEE-754 number is computed as:

$$\texttt{sign} * 2^{\texttt{exponent}} * \texttt{mantissa}$$

The sign is stored in bit 32. The exponent can be computed from bits 24-31 by subtracting 127. The mantissa (also known as significand or fraction) is stored in bits 1-23. An invisible leading bit (i.e. it is not actually stored) with value 1.0 is placed in front, then bit 23 has a value of 1/2, bit 22 has value 1/4 etc. As a result, the mantissa has a value between 1.0 and 2. If the exponent reaches -127 (binary 00000000), the leading 1 is no longer used to enable gradual underflow.

Underflow: If the exponent has minimum value (all zero), special rules for denormalized values are followed. The exponent value is set to 2^{-126} while the "invisible" leading bit for the mantissa is no longer used. The range of the mantissa is now [0:1).
Note: The converter used to show denormalized exponents as 2^{-127} and a denormalized mantissa range [0:2). This is effectively identical to the values above, with a factor of two shifted between exponent and mantissa. However this confused people and was therefore changed (2015-09-26).

Rounding errors: Not every decimal number can be expressed exactly as a floating point number. This can be seen when entering "0.1" and examining

its binary representation which is either slightly smaller or larger, depending on the last bit.

Other representations: The hex representation is just the integer value of the bitstring printed as hex. Don't confuse this with true hexadecimal floating point values in the style of 0xab.12ef.

FAQ (Frequently Asked Questions):

Can you send me the source code? I need to convert format x to format y.
This source code for this converter doesn't contain any low level conversion routines. The conversion between a floating point number (i.e. a 32 bit area in memory) and the bit representation isn't actually a conversion, but just a reinterpretation of the same data in memory. This can be easily done with typecasts in C/C++ or with some bitfiddling via java.lang.Float.floatToIntBits in Java. The conversion between a string containing the textual form of a floating point number (e.g. "3.14159", a string of 7 characters) and a 32 bit floating point number is also performed by library routines. If you need to write such a routine yourself, you should have a look at the sourecode of a standard C library (e.g. GNU libc, uclibc or the FreeBSD C library - please have a look at the licenses before copying the code) - be aware, these conversions can be complicated.

I've converted a number to floating point by hand/some other method, and I get a different result. Your converter is wrong!
Possibly, but unlikely. The conversion routines are pretty accurate (see above). Until now, checking the results always proved the other conversion less accurate. In particular toggling the last bit can lead to the same result after rounding to a fixed number of decimal places. Please check the double precision result (bottom line in the converter) and compare the difference to the expected decimal result while toggling the last bit.