WIKIPEDIA
The Free Encyclopedia

Single-precision floating-point format

Single-precision floating-point format (sometimes called **FP32** or **float32**) is a computer number format, usually occupying 32 bits in computer memory; it represents a wide dynamic range of numeric values by using a floating radix point.

A floating-point variable can represent a wider range of numbers than a fixed-point variable of the same bit width at the cost of precision. A signed 32-bit integer variable has a maximum value of $2^{31} - 1 = 2{,}147{,}483{,}647$, whereas an IEEE 754 32-bit base-2 floating-point variable has a maximum value of $(2 - 2^{-23}) \times 2^{127} \approx 3.4028235 \times 10^{38}$. All integers with 7 or fewer decimal digits, and any 2^n for a whole number $-149 \leq n \leq 127$, can be converted exactly into an IEEE 754 single-precision floating-point value.

In the IEEE 754-2008 standard, the 32-bit base-2 format is officially referred to as **binary32**; it was called **single** in IEEE 754-1985. IEEE 754 specifies additional floating-point types, such as 64-bit base-2 *double precision* and, more recently, base-10 representations.

One of the first programming languages to provide single- and double-precision floating-point data types was Fortran. Before the widespread adoption of IEEE 754-1985, the representation and properties of floating-point data types depended on the computer manufacturer and computer model, and upon decisions made by programming-language designers. E.g., GW-BASIC's single-precision data type was the 32-bit MBF floating-point format.

Single precision is termed *REAL* in Fortran,[1] *SINGLE-FLOAT* in Common Lisp,[2] *float* in C, C++, C#, Java,[3] *Float* in Haskell[4] and Swift,[5] and *Single* in Object Pascal (Delphi), Visual Basic, and MATLAB. However, *float* in Python, Ruby, PHP, and OCaml and *single* in versions of Octave before 3.2 refer to double-precision numbers. In most implementations of PostScript, and some embedded systems, the only supported precision is single.

IEEE 754 standard: binary32

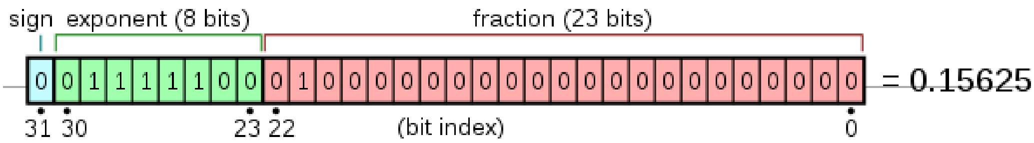
The IEEE 754 standard specifies a *binary32* as having:

- Sign bit: 1 bit
- Exponent width: 8 bits
- Significand precision: 24 bits (23 explicitly stored)

This gives from 6 to 9 significant decimal digits precision. If a decimal string with at most 6 significant digits is converted to the IEEE 754 single-precision format, giving a normal number, and then converted back to a decimal string with the same number of digits, the final result should match the original string. If an IEEE 754 single-precision number is converted to a decimal string with at least 9 significant digits, and then converted back to single-precision representation, the final result must match the original number.[6]

The sign bit determines the sign of the number, which is the sign of the significand as well. The exponent is an 8-bit unsigned integer from 0 to 255, in biased form: an exponent value of 127 represents the actual zero. Exponents range from −126 to +127 because exponents of −127 (all 0s) and +128 (all 1s) are reserved for special numbers.

The true significand includes 23 fraction bits to the right of the binary point and an *implicit leading bit* (to the left of the binary point) with value 1, unless the exponent is stored with all zeros. Thus only 23 fraction bits of the significand appear in the memory format, but the total precision is 24 bits (equivalent to $\log_{10}(2^{24}) \approx 7.225$ decimal digits). The bits are laid out as follows:



The real value assumed by a given 32-bit *binary32* data with a given *sign*, biased exponent *e* (the 8-bit unsigned integer), and a *23-bit fraction* is

$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\dots b_{23})_2 - 127} \times (1.b_{22}b_{21}\dots b_0)_2,$$

which yields

$$\text{value} = (-1)^{\text{sign}} \times 2^{(E-127)} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right).$$

In this example:

- $\text{sign} = b_{31} = 0$,
- $(-1)^{\text{sign}} = (-1)^0 = +1 \in \{-1, +1\}$,
- $E = (b_{30}b_{29}\dots b_{23})_2 = \sum_{i=0}^{7} b_{23+i} 2^{+i} = 124 \in \{1, \ldots, (2^8 - 1) - 1\} = \{1, \ldots, 254\}$,
- $2^{(E-127)} = 2^{124-127} = 2^{-3} \in \{2^{-126}, \ldots, 2^{127}\}$,
- $1.b_{22}b_{21}\dots b_0 = 1 + \sum_{i=1}^{23} b_{23-i} 2^{-i} = 1 + 1 \cdot 2^{-2} = 1.25 \in \{1, 1 + 2^{-23}, \ldots, 2 - 2^{-23}\} \subset [1; 2 - 2^{-23}] \subset [1; 2)$
  .

thus:

- $\text{value} = (+1) \times 2^{-3} \times 1.25 = +0.15625$.

Note:

- $1 + 2^{-23} \approx 1.000\,000\,119$,
- $2 - 2^{-23} \approx 1.999\,999\,881$,
- $2^{-126} \approx 1.175\,494\,35 \times 10^{-38}$,
- $2^{+127} \approx 1.701\,411\,83 \times 10^{+38}$.

Exponent encoding

The single-precision binary floating-point exponent is encoded using an offset-binary representation, with the zero offset being 127; also known as exponent bias in the IEEE 754 standard.

- $E_{min} = 01_H - 7F_H = -126$
- $E_{max} = FE_H - 7F_H = 127$
- Exponent bias = $7F_H = 127$

Thus, in order to get the true exponent as defined by the offset-binary representation, the offset of 127 has to be subtracted from the stored exponent.

The stored exponents 00_H and FF_H are interpreted specially.

Exponent	fraction = 0	fraction ≠ 0	Equation
$00_H = 00000000_2$	±zero	subnormal number	$(-1)^{\text{sign}} \times 2^{-126} \times 0.\text{fraction}$
01_H, ..., $FE_H = 00000001_2$, ..., 11111110_2	normal value		$(-1)^{\text{sign}} \times 2^{\text{exponent}-127} \times 1.\text{fraction}$
$FF_H = 11111111_2$	±infinity	NaN (quiet, signalling)	

The minimum positive normal value is $2^{-126} \approx 1.18 \times 10^{-38}$ and the minimum positive (subnormal) value is $2^{-149} \approx 1.4 \times 10^{-45}$.

Converting decimal to binary32

In general, refer to the IEEE 754 standard itself for the strict conversion (including the rounding behaviour) of a real number into its equivalent binary32 format.

Here we can show how to convert a base-10 real number into an IEEE 754 binary32 format using the following outline:

- Consider a real number with an integer and a fraction part such as 12.375
- Convert and normalize the integer part into binary
- Convert the fraction part using the following technique as shown here
- Add the two results and adjust them to produce a proper final conversion

Conversion of the fractional part: Consider 0.375, the fractional part of 12.375. To convert it into a binary fraction, multiply the fraction by 2, take the integer part and repeat with the new fraction by 2 until a fraction of zero is found or until the precision limit is reached which is 23 fraction digits for IEEE 754 binary32 format.

$0.375 \times 2 = 0.750 = 0 + 0.750 \Rightarrow b_{-1} = 0$, the integer part represents the binary fraction digit. Re-multiply 0.750 by 2 to proceed

$0.750 \times 2 = 1.500 = 1 + 0.500 \Rightarrow b_{-2} = 1$

$0.500 \times 2 = 1.000 = 1 + 0.000 \Rightarrow b_{-3} = 1$, fraction = 0.011, terminate

We see that $(0.375)_{10}$ can be exactly represented in binary as $(0.011)_2$. Not all decimal fractions can be represented in a finite digit binary fraction. For example, decimal 0.1 cannot be represented in binary exactly, only approximated. Therefore:

$$(12.375)_{10} = (12)_{10} + (0.375)_{10} = (1100)_2 + (0.011)_2 = (1100.011)_2$$

Since IEEE 754 binary32 format requires real values to be represented in $(1.x_1x_2\ldots x_{23})_2 \times 2^e$ format (see Normalized number, Denormalized number), 1100.011 is shifted to the right by 3 digits to become $(1.100011)_2 \times 2^3$

Finally we can see that: $(12.375)_{10} = (1.100011)_2 \times 2^3$

From which we deduce:

- The exponent is 3 (and in the biased form it is therefore $(127 + 3)_{10} = (130)_{10} = (1000\ 0010)_2$
- The fraction is 100011 (looking to the right of the binary point)

From these we can form the resulting 32-bit IEEE 754 binary32 format representation of 12.375:

$$(12.375)_{10} = (0\ 10000010\ 10001100000000000000000)_2 = (41460000)_{16}$$

Note: consider converting 68.123 into IEEE 754 binary32 format: Using the above procedure you expect to get $(42883\text{EF9})_{16}$ with the last 4 bits being 1001. However, due to the default rounding behaviour of IEEE 754 format, what you get is $(42883\text{EFA})_{16}$, whose last 4 bits are 1010.

Example 1: Consider decimal 1. We can see that: $(1)_{10} = (1.0)_2 \times 2^0$

From which we deduce:

- The exponent is 0 (and in the biased form it is therefore $(127 + 0)_{10} = (127)_{10} = (0111\ 1111)_2$
- The fraction is 0 (looking to the right of the binary point in 1.0 is all $0 = 000\ldots0$)

From these we can form the resulting 32-bit IEEE 754 binary32 format representation of real number 1:

$$(1)_{10} = (0\ 01111111\ 00000000000000000000000)_2 = (3\text{F}800000)_{16}$$

Example 2: Consider a value 0.25. We can see that: $(0.25)_{10} = (1.0)_2 \times 2^{-2}$

From which we deduce:

- The exponent is −2 (and in the biased form it is $(127 + (-2))_{10} = (125)_{10} = (0111\ 1101)_2$)
- The fraction is 0 (looking to the right of binary point in 1.0 is all zeroes)

From these we can form the resulting 32-bit IEEE 754 binary32 format representation of real number 0.25:

$$(0.25)_{10} = (0\ 01111101\ 00000000000000000000000)_2 = (3E800000)_{16}$$

Example 3: Consider a value of 0.375. We saw that $0.375 = (0.011)_2 = (1.1)_2 \times 2^{-2}$

Hence after determining a representation of 0.375 as $(1.1)_2 \times 2^{-2}$ we can proceed as above:

- The exponent is −2 (and in the biased form it is $(127 + (-2))_{10} = (125)_{10} = (0111\ 1101)_2$)
- The fraction is 1 (looking to the right of binary point in 1.1 is a single $1 = x_1$)

From these we can form the resulting 32-bit IEEE 754 binary32 format representation of real number 0.375:

$$(0.375)_{10} = (0\ 01111101\ 10000000000000000000000)_2 = (3EC00000)_{16}$$

Converting binary32 to decimal

If the binary32 value, `41C80000` in this example, is in hexadecimal we first convert it to binary:

$$41C8\ 0000_{16} = 0100\ 0001\ 1100\ 1000\ 0000\ 0000\ 0000\ 0000_2$$

then we break it down into three parts: sign bit, exponent, and significand.

- Sign bit: 0_2
- Exponent: $1000\ 0011_2 = 83_{16} = 131_{10}$
- Significand: $100\ 1000\ 0000\ 0000\ 0000\ 0000_2 = 480000_{16}$

We then add the implicit 24th bit to the significand:

- Significand: $\mathbf{1}100\ 1000\ 0000\ 0000\ 0000\ 0000_2 = C80000_{16}$

and decode the exponent value by subtracting 127:

- Raw exponent: $83_{16} = 131_{10}$
- Decoded exponent: $131 - 127 = 4$

Each of the 24 bits of the significand (including the implicit 24th bit), bit 23 to bit 0, represents a value, starting at 1 and halves for each bit, as follows:

```
bit 23 = 1
bit 22 = 0.5
bit 21 = 0.25
bit 20 = 0.125
bit 19 = 0.0625
bit 18 = 0.03125
bit 17 = 0.015625
.
.
bit 6 = 0.00000762939453125
bit 5 = 0.000003814697265625
bit 4 = 0.0000019073486328125
bit 3 = 0.00000095367431640625
bit 2 = 0.000000476837158203125
bit 1 = 0.0000002384185791015625
bit 0 = 0.00000011920928955078125
```

The significand in this example has three bits set: bit 23, bit 22, and bit 19. We can now decode the significand by adding the values represented by these bits.

- Decoded significand: $1 + 0.5 + 0.0625 = 1.5625 = \mathrm{C}80000/2^{23}$

Then we need to multiply with the base, 2, to the power of the exponent, to get the final result:

$$1.5625 \times 2^4 = 25$$

Thus

$$41\mathrm{C}8\ 0000 = 25$$

This is equivalent to:

$$n = (-1)^s \times (1 + m * 2^{-23}) \times 2^{x-127}$$

where s is the sign bit, x is the exponent, and m is the significand.

Precision limitations on decimal values (between 1 and 16777216)

- Decimals between 1 and 2: fixed interval 2^{-23} ($1+2^{-23}$ is the next largest float after 1)
- Decimals between 2 and 4: fixed interval 2^{-22}
- Decimals between 4 and 8: fixed interval 2^{-21}
- ...
- Decimals between 2^n and 2^{n+1}: fixed interval 2^{n-23}
- ...
- Decimals between 2^{22}=4194304 and 2^{23}=8388608: fixed interval 2^{-1}=0.5
- Decimals between 2^{23}=8388608 and 2^{24}=16777216: fixed interval 2^0=1

Precision limitations on integer values

- Integers between 0 and 16777216 can be exactly represented (also applies for negative integers between −16777216 and 0)
- Integers between 2^{24}=16777216 and 2^{25}=33554432 round to a multiple of 2 (even number)
- Integers between 2^{25} and 2^{26} round to a multiple of 4
- ...
- Integers between 2^n and 2^{n+1} round to a multiple of 2^{n-23}
- ...
- Integers between 2^{127} and 2^{128} round to a multiple of 2^{104}
- Integers greater than or equal to 2^{128} are rounded to "infinity".

Notable single-precision cases

These examples are given in bit *representation*, in hexadecimal and binary, of the floating-point value. This includes the sign, (biased) exponent, and significand.

$0\ 00000000\ 00000000000000000000001_2$ = 0000 0001$_{16}$ = $2^{-126} \times 2^{-23} = 2^{-149} \approx 1.4012984643 \times 10^{-45}$
(smallest positive subnormal number)

$0\ 00000000\ 11111111111111111111111_2$ = 007f ffff$_{16}$ = $2^{-126} \times (1 - 2^{-23}) \approx 1.1754942107 \times 10^{-38}$
(largest subnormal number)

```
0 00000001 00000000000000000000000₂ = 0080 0000₁₆ = 2^−126 ≈ 1.1754943508 × 10^−38
                                                   (smallest positive normal number)
```

```
0 11111110 11111111111111111111111₂ = 7f7f ffff₁₆ = 2^127 × (2 − 2^−23) ≈ 3.4028234664 × 10^38
                                                   (largest normal number)
```

```
0 01111110 11111111111111111111111₂ = 3f7f ffff₁₆ = 1 − 2^−24 ≈ 0.999999940395355225
                                                   (largest number less than one)
```

```
0 01111111 00000000000000000000000₂ = 3f80 0000₁₆ = 1 (one)
```

```
0 01111111 00000000000000000000001₂ = 3f80 0001₁₆ = 1 + 2^−23 ≈ 1.00000011920928955
                                                   (smallest number larger than one)
```

```
1 10000000 00000000000000000000000₂ = c000 0000₁₆ = −2
0 00000000 00000000000000000000000₂ = 0000 0000₁₆ = 0
1 00000000 00000000000000000000000₂ = 8000 0000₁₆ = −0

0 11111111 00000000000000000000000₂ = 7f80 0000₁₆ = infinity
1 11111111 00000000000000000000000₂ = ff80 0000₁₆ = −infinity

0 10000000 10010010000111111011011₂ = 4049 0fdb₁₆ ≈ 3.14159274101257324 ≈ π ( pi )
0 01111101 01010101010101010101011₂ = 3eaa aaab₁₆ ≈ 0.333333343267440796 ≈ 1/3

x 11111111 10000000000000000000001₂ = ffc0 0001₁₆ = qNaN (on x86 and ARM processors)
x 11111111 00000000000000000000001₂ = ff80 0001₁₆ = sNaN (on x86 and ARM processors)
```

By default, 1/3 rounds up, instead of down like double precision, because of the even number of bits in the significand. The bits of 1/3 beyond the rounding point are `1010...` which is more than 1/2 of a unit in the last place.

Encodings of qNaN and sNaN are not specified in IEEE 754 and implemented differently on different processors. The x86 family and the ARM family processors use the most significant bit of the significand field to indicate a quiet NaN. The PA-RISC processors use the bit to indicate a signalling NaN.

Optimizations

The design of floating-point format allows various optimisations, resulting from the easy generation of a base-2 logarithm approximation from an integer view of the raw bit pattern. Integer arithmetic and bit-shifting can yield an approximation to reciprocal square root (fast inverse square root), commonly required in computer graphics.

See also

- IEEE 754
- ISO/IEC 10967, language independent arithmetic
- Primitive data type
- Numerical stability
- Scientific notation

References

1. "REAL Statement" (https://web.archive.org/web/20210224045812/http://scc.ustc.edu.cn/zlsc/sugon/intel/compiler_f/main_for/lref_for/source_files/rfreals.htm). *scc.ustc.edu.cn*. Archived from the original (http://scc.ustc.edu.cn/zlsc/sugon/intel/compiler_f/main_for/lref_for/source_files/rfreals.htm) on 2021-02-24. Retrieved 2013-02-28.
2. "CLHS: Type SHORT-FLOAT, SINGLE-FLOAT, DOUBLE-FLOAT..." (http://www.lispworks.com/documentation/HyperSpec/Body/t_short_.htm)
3. "Primitive Data Types" (http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html). *Java Documentation*.

4. "6 Predefined Types and Classes" (https://www.haskell.org/onlinereport/haskell2010/haskellch6.html#x13-1350006.4). *haskell.org*. 20 July 2010.
5. "Float" (https://developer.apple.com/documentation/swift/float). *Apple Developer Documentation*.
6. William Kahan (1 October 1997). "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic" (http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF) (PDF). p. 4.

External links

- Live floating-point bit pattern editor (https://evanw.github.io/float-toy/)
- Online calculator (http://www.h-schmidt.net/FloatConverter/IEEE754.html)
- Online converter for IEEE 754 numbers with single precision (http://www.binaryconvert.com/convert_float.html)
- C source code to convert between IEEE double, single, and half precision (https://web.archive.org/web/20091031135212/http://www.mathworks.com/matlabcentral/fileexchange/23173)