Single-precision floating-point format

Single-precision floating-point format is a computer number format that occupies 4 bytes (32 bits) in computer memory and represents a wide dynamic range of values by using a floating point.

In IEEE 754-2008 the 32-bit base 2 format is officially referred to as **binary32**. It was called **single** in IEEE 754-1985. In older computers, other floating-point formats of 4 bytes were used.

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 the double float data type depended on the computer manufacturer and computer model.

Single-precision binary floating-point is used due to its wider range over fixed point (of the same bit-width), even if at the cost of precision.

Single precision is known as **REAL** in Fortran[1], as **float** in C, C++, C#, Java[2] and Haskell, and as **single** in Delphi (Pascal), Visual Basic, and MATLAB. However, **float** in Python, Ruby, PHP, and OCaml and **single** in versions of Octave prior to 3.2 refer to double-precision numbers.

IEEE 754 single-precision binary floating-point format: binary32

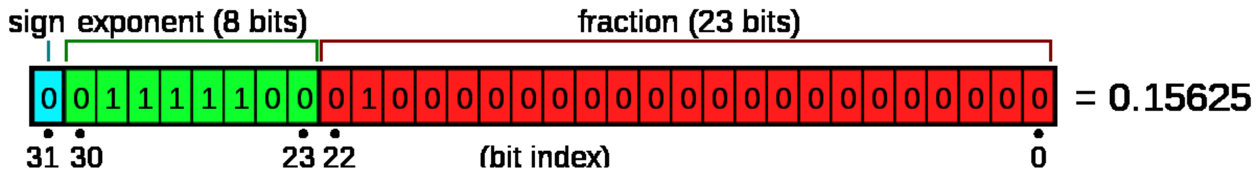
The IEEE 754 standard specifies a **binary32** as having:

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

This gives from 6 to 9 significant decimal digits precision (if a decimal string with at most 6 significant decimal is converted to IEEE 754 single precision and then converted back to the same number of significant decimal, then the final string should match the original; and if an IEEE 754 single precision is converted to a decimal string with at least 9 significant decimal and then converted back to single, then the final number must match the original [3]).

Sign bit determines the sign of the number, which is the sign of the significand as well. Exponent is either an 8 bit signed integer from −128 to 127 (2's Complement) or an 8 bit unsigned integer from 0 to 255 which is the accepted biased form in IEEE 754 binary32 definition. For this case an exponent value of 127 represents the actual zero.

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 biased exponent **e** and a **23 bit fraction** is $= (-1)^{\text{sign}}(1.b_{-1}b_{-2}...b_{-23})_2 \times 2^{e-127}$ where more precisely we have:

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

In this example:

- $sign = 0$
- $1 + \sum_{i=1}^{23} b_{23-i}2^{-i} = 1 + 2^{-2} = 1.25$

- $2^{(e-127)} = 2^{124-127} = 2^{-3}$

thus:

- $value = 1.25 \times 2^{-3} = 0.15625$

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	Significand zero	Significand non-zero	Equation
00_H	zero, −0	subnormal numbers	$(-1)^{\text{signbits}} \times 2^{-126} \times 0.\text{significandbits}$
01_H, ..., FE_H	normalized value		$(-1)^{\text{signbits}} \times 2^{\text{exponentbits}-127} \times 1.\text{significandbits}$
FF_H	±infinity	NaN (quiet, signalling)	

The minimum positive (subnormal) value is $2^{-149} \approx 1.4 \times 10^{-45}$. The minimum positive normal value is $2^{-126} \approx 1.18 \times 10^{-38}$. The maximum representable value is $(2-2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$.

Converting from decimal representation to binary32 format

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 re-multiply 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 x 2 = 0.750 = 0 + 0.750 => b_{-1} = 0, the integer part represents the binary fraction digit. Re-multiply 0.750 by 2 to proceed

0.750 x 2 = 1.500 = 1 + 0.500 => b_{-2} = 1

0.500 x 2 = 1.000 = 1 + 0.000 => b_{-3} = 1, fraction = 0.000, 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. So it is only approximated.

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

Also IEEE 754 binary32 format requires that you represent real values in $(1.x_1x_2...x_{23})_2 \times 2^e$ format, (see Normalized number, Denormalized number) so that 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 130 = 1000 0010)
- 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 as: 0-10000010-10001100000000000000000 = 41460000_H

Note: consider converting 68.123 into IEEE 754 binary32 format: Using the above procedure you expect to get $42883EF9_H$ with the last 4 bits being 1001 However due to the default rounding behaviour of IEEE 754 format what you get is $42883EFA_H$ whose last 4 bits are 1010 .

Ex 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 = 0111 1111)
- The fraction is 0 (looking to the right of the binary point in 1.0 is all 0 = 000...0)

From these we can form the resulting 32 bit IEEE 754 binary32 format representation of real number 1 as: 0-01111111-00000000000000000000000 = $3f800000_H$

Ex 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)= 125 = 0111 1101)
- The fraction is 0 (looking to the right of binary point in 1.0 is all zeros)

From these we can form the resulting 32 bit IEEE 754 binary32 format representation of real number 0.25 as: 0-01111101-00000000000000000000000 = $3e800000_H$

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

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)= 125 = 0111 1101)
- 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 as: 0-01111101-10000000000000000000000 = $3ec00000_H$

Single-precision examples

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

```
3f80 0000   = 1
c000 0000   = −2

7f7f ffff   ≈ 3.4028234 × 10^38  (max single precision)

0000 0000   = 0
8000 0000   = −0

7f80 0000   = infinity
```

```
ff80 0000   = −infinity

3eaa aaab   ≈ 1/3
```

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

Converting from single-precision binary to decimal

We start with the hexadecimal representation of the value, 41c80000, in this example, and 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
Exponent: 1000 0011$_{2}$ = 83$_{16}$ = 131
Significand: 100 1000 0000 0000 0000 0000$_{2}$ = 480000$_{16}$

We then add the implicit 24th bit to the significand

Significand: **1**100 1000 0000 0000 0000 0000$_{2}$ = C80000$_{16}$

and decode the exponent value by subtracting 127

Raw exponent: 83$_{16}$ = 131
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  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 = C80000/2^{23}

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

1.5625 × 2^{4} = **25**

Thus

```
41c8 0000   = 25
```

This is equivalent to:

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

External links

- Online calculator [4]
- Online converter for IEEE 754 numbers with single precision [5]
- C source code to convert between IEEE double, single, and half precision can be found here [6]

References

[1] http://scc.ustc.edu.cn/zlsc/sugon/intel/compiler_f/main_for/lref_for/source_files/rfreals.htm
[2] http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html
[3] William Kahan (1 October 1987). "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic" (http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF). .
[4] http://www.h-schmidt.net/FloatApplet/IEEE754.html
[5] http://www.binaryconvert.com/convert_float.html
[6] http://www.mathworks.com/matlabcentral/fileexchange/23173

Article Sources and Contributors

Single-precision floating-point format *Source*: http://en.wikipedia.org/w/index.php?oldid=541331000 *Contributors*: 198.207.223.xxx, Aklassyguy, Anthony Appleyard, B4hand, Brianbjparker, Cabyd, Conversion script, Coolant123, Cybercobra, Danilozf, Dicklyon, Dryguy, Dvaselaar, EncMstr, Etoombs, Foobaz, Goblin, Goodrone, Graham87, Happyuk, Harutsedo2, JLaTondre, JakeVortex, Jshadias, KeegY, Keka, KlappCK, Kuttipapu, Lamegeek8, Lone boatman, Mandarax, MattGiuca, Maury Markowitz, Mfc, Michael Angelkovich, Michael Hardy, Mortense, MrOllie, Qwerty112233, Radagast83, Rjstott, Sawak, ShashClp, Spacepotato, Stannered, StefanNL, Suruena, Theshadow27, UNV, Vadmium, Ylai, 105 anonymous edits

Image Sources, Licenses and Contributors

Image:Float example.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Float_example.svg *License*: GNU Free Documentation License *Contributors*: en:User:Fresheneesz, traced by User:Stannered

License