Real numbers in binary

Decimal fraction to Binary

*Convert $0.8125_{10}$ into binary*

Continually multiply the number by 2 until the fractional part of the result = 0 or the required precision has been reached. The integers form the solution:

$$
\begin{align*}
0.8125 \times 2 &= 1.625 \\
0.625 \times 2 &= 1.25 \\
0.25 \times 2 &= 0.5 \\
0.5 \times 2 &= 1.0 \\
0.0 \times 2 &= 0.0 \\
0.0 \times 2 &= 0.0
\end{align*}
$$

Decimal fraction to Hex

*Convert $0.00927734375_{10}$ into hex*

Continually multiply the number by 16 until the fractional part of the result = 0 or the required precision has been reached. The integers form the solution:

$$
\begin{align*}
0.00927734375 \times 16 &= 0.1484375 \\
0.1484375 \times 16 &= 2.375 \\
0.375 \times 16 &= 6.0 \\
0.0 \times 16 &= 0.0
\end{align*}
$$
Real numbers in binary

Floating point
By implementing a floating point format, an enormous range of numbers is possible (at the expense of accuracy). There are several formats but standards are IEEE short, IEEE long and IEEE extended real formats.

IEEE real number ranges

| Format   | Bits | Range of | |X| |<|  |
|----------|------|----------|-----|-----|-----|
| float    | 32   | \(1.18 \times 10^{-38}\) \(<\ |X| < 3.40 \times 10^{38}\) |
| double   | 64   | \(2.23 \times 10^{-308}\) \(<\ |X| < 1.79 \times 10^{308}\) |
| long double | 80  | \(3.37 \times 10^{-4932}\) \(<\ |X| < 1.18 \times 10^{4932}\) |

Short real format (32 bits)
The number is stored in the following format:

S=1 bit, Exponent=8 bits, Mantissa=23 bits

\[
\begin{array}{c|c|c}
S & \text{Exponent} & \text{Mantissa} \\
\hline
& & \\
\end{array}
\]

First the number needs to be normalised. This is similar to normalising a decimal number, e.g.

1234.5678 normalised is \(1.2345678 \times 10^3\)

This number has an exponent of 3, a mantissa of 12345678 and is positive (s=0 for +ve and 1 for –ve).
Real numbers in binary

A binary example:
0.0000101 normalised is $1.01 \times 2^{-5}$

To cope with signed exponents, the exponent is "biased" as opposed to using 2's complement. This means that the exponent has 127 added to it.

Because a normalised binary number always starts with a 1 there is no point in storing it so it is dumped and the mantissa shifted left 1 place in order to fit another digit into memory.

Example:

Show how the decimal number 0.0390625 would be stored in IEEE short real format.

$0.0390625_{10} = 0.0000101_2$

Normalise the binary number to $1.01 \times 2^{-5}$

Sign=0 as num is +ve
Exponent = $-5+127=122_{10}$ or $01111010_2$
Mantissa = $010000000000000000000000_2$

So stored number is:

0 01111010 010000000000000000000000_2
0011 1101 0010 0000 0000 0000 0000 0000_2

or $3D200000_{16}$
Real numbers in binary

Reading a short format number from computer memory:

What is the decimal value of:

10111110 1110100 00000000 00000000₂

or BEF40000₁₆

Sign=1 so –ve
Exp=01111101₂ or 125₁₀ so subtract 127 to give -2
Mantissa=11110100000000000000000₂ when the hidden bit is replaced.

So the normalised value is
1.11101000000000000000000₂ x 2⁻² or
0.01111010000000000000000₂ or 0.4765625₁₀

So the actual decimal value is -0.4765625₁₀

The following program produced:

```c
void main()
{
    static float x;
    x=-0.4765625;
}
```

C7 06 8E 02 F4 BE  mov word ptr [028E],BEF4
C7 06 8C 02 00 00  mov word ptr [028C],0000
Real numbers in binary

The integer and three floating point types supported in Borland C++ builder are:

32-bit integers

- **short int**
  - `s` (sign bit) = 0 (positive), 1 (negative)
  - `b3` (2's complement)
  - `m` (magnitude)

- **int, long int**
  - `s` (sign bit) = 0 (positive), 1 (negative)
  - `b3` (2's complement)
  - `m` (magnitude)

Floating point types, always

- **float**
  - `s` (sign bit) = 0 (positive), 1 (negative)
  - `b5` (biased exponent)
  - `f6` (significand)

- **double**
  - `s` (sign bit) = 0 (positive), 1 (negative)
  - `b10` (biased exponent)
  - `f15` (significand)

- **long double**
  - `s` (sign bit) = 0 (positive), 1 (negative)
  - `b16` (biased exponent)
  - `f34` (significand)

- `i` = Position of implicit binary point
- `x` = Integer bit of significance
- `l` = Integer bit of significance

**Figure 1 - Internal representation of numeric types**