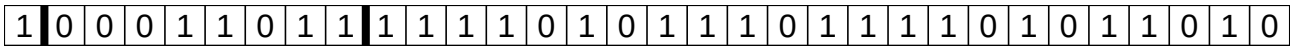
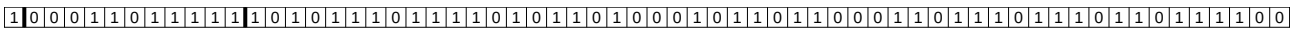
An Attempt to Explain PR #7757

32-bit floating point numbers look like this:

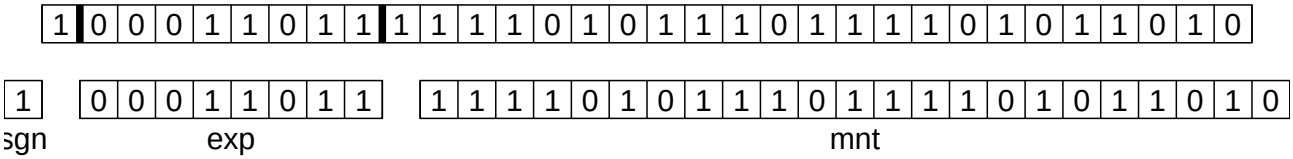


64-bit floating point numbers look quite similar, just twice as long:



Because there is no real difference, I'll explain the algorithm using 32-bit floating point numbers and just point out important differences.

In the function `printFloat` this bit pattern has already been split into three parts: A string called `sgn`, an `int` called `exp` and an `ulong` called `mnt`.



`sgn` has been translated into an empty string, or one of length one, containing either plus, minus or a space. We do not have to care about this for the time being. Let's concentrate on `exp` and `mnt` first. These two numbers are handed to `printFloatF`. At the invocation of `printFloatF` `exp` is a number from 0 to 254. Please note, that the value 255 has a special meaning, it's used for `nan`'s and `inf`'s. These have already been take care of in `printFloat`. (For doubles the value ranges von 0 to 1022.)

Now I'll go through the important code snippets from `printFloatF`, trying to explain everything in detail:

```
if (exp == 0 && mnt == 0)
    return printFloat0(buf, f, sgn, is_upper);
```

First, the algorithm checks, if the number to print is zero. In this case a specialized function for printing zero is called.

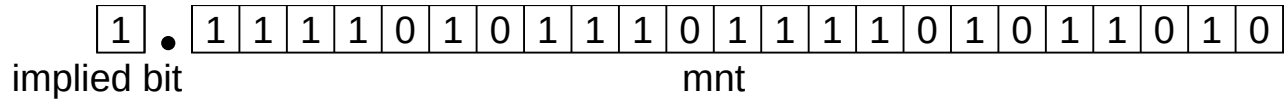
```
if (exp != 0)
    mnt |= 1L << (T.mant_dig - 1);
else
    exp = 1;
```

There is one more special case we need to cope with: Denormalized numbers, which are the numbers with `exp==0`. Normalized values get an implied 1 to the left of the mantissa (`mant_dig` is the number of digits of the mantissa including this implied 1), while denormalized values are considered to be equal to numbers with `exp==1`, but without this 1 bit set.

```
exp -= bias;
```

Next, we have to adjust the exponent. This is done by subtracting a so called bias, which is 127 for floats. (Formula for bias: $2^{r-1}-1$, where r is the number of digits in `exp`). So now `exp` is in the range from -126 to 127 (-1022 to 1023 for doubles).

Now we can express our number as $2^e \times m$, where e is `exp` and and m is `mnt` with a dot after the first (implied) digit, written in base 2.



Or with other words, we need to move the dot `exp` digits to the right. After that, left of the dot, we've got the integral part of our number and right of the dot, there is the fractional part.

Please skip now the next few lines in the source code. They prepare the buffer for the output and estimate the number of characters, which will be used in the end. For the time being, assume that buffer to be infinite large in both directions: To the left and to the right.

```
char[] buffer = ...
size_t start = max_left;
size_t left = max_left;
size_t right = max_left;
```

We'v got three pointers (`start`, `left`, `right` to that buffer, which at the beginning all point to the same place, it's the place of the dot between integral and fractional part. `left` will move to the left and `right` will move to the right, when we add more digits. Finally we end up with the integral part between `left` and `start` and the fractional part between `start` and `right`.

Fasten your seat-belts we are ready for take of…

```
if (exp >= T.mant_dig)
{
    ...
}
else if (exp + 61 < T.mant_dig)
{
    ...
}
else
{
    ...
}
```

We use three different algorithms to convert the digits from binary to decimal system. The first algorithm is used, when there are no fractional digits, that is, the dot is moved to a place somewhere right of the mantissa. The second algorithm is used, when there are only fractional digits. The third algorithm is used, when the integral digits and the fractional digits fit into an `ulong` each.

Sidenote: Some numbers fit into the third case and into one of the other two cases at the same time. Here is some possibility to do some speed up by choosing the better case. If havn't done that yet.

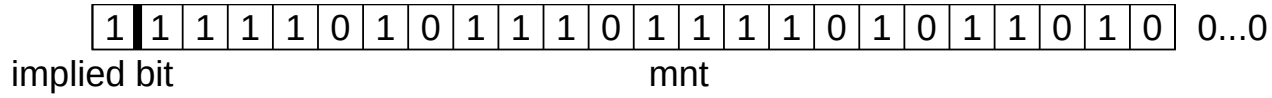
First algorithm, only integral bits.

Integers are converted from binary to decimal by repeated dividing of the number by 10 and writing down the reminder from right to left. As an example take the number 11001110_2:

$$\begin{aligned} 11001110 &= 1010 \times 10100 + 110 \\ 10100 &= 1010 \times 10 + 0 \\ 10 &= 1010 + 10 \end{aligned}$$

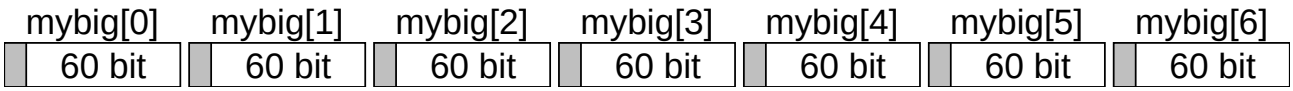
The reminders are 110, 0, 10, which are 6, 0 and 2 in decimal. So $11001110_2 = 206_{10}$ – remember, we have to write that digits from right to left.

Unfortunately, our number is very, very large. It looks like this:



Please note the 0...0 at the right end of the number. The number of zeros can be calculated by subtracting the number of bits of the mantissa (without implied leading bit) from `exp`.

We safe this long number in an array of `ulong`s. Theoretically, we could use 64 bits of every `ulong`, but I only use 60 bits from each. I use the other 4 bits for the carry number to simplify the large division to come. Our number looks now like this:




```
int count = exp / 60 + 1;

ulong[] mybig = bigbuf[0 .. count];
```

Compute the number of `ulong`s needed and use a slice from static array `bigbuf` to avoid GC usage. (If you worry about that `+1`: We have got [exp+1] digits and we need to do rounding up after the division. But `/` rounds down. So we subtract one before the division and add one after it. That's equivalent of rounding up.)

```
int lower = 60 - (exp - T.mant_dig + 1) % 60;
if (lower < T.mant_dig)
{
    mybig[0] = mnt >> lower;
    mybig[1] = (mnt & ((1L << lower) - 1)) << 60 - lower;
}
else
    mybig[0] = (mnt & ((1L << lower) - 1)) << 60 - lower;
```

Next we have to copy the mantissa-digits in our big number. There are two cases: Either it fits completely into the leftmost `ulong` or we have to split it between the leftmost and the second `ulong`. Ths depends on the number of zeros attached to the right, that is the value `exp`. `lower` holds the number of bits, that have to be put in the leftmost `ulong`.

Now we reached the heart of the first algorithm. In pseudo code it's:

```
while (number != 0)
    divide number by 10 and save the reminder
```

In D it looks like this:

```
int msu = 0;
while (msu < count - 1 || mybig[$-1] != 0)
{
    ulong mod = 0;
    foreach (i;msu .. count)
    {
        mybig[i] |= mod << 60;
        mod = mybig[i] % 10;
        mybig[i] /= 10;
    }
    if (mybig[msu] == 0)
        ++msu;

    buffer[--left] = cast(byte) ('0'+mod);
}
```

`msu` is the *most significant ulong*. It points to the leftmost `ulong`, that is not 0. At the beginning that's the `mybig[0]`. (For prove I note, that the implied bit is always set when there are integral parts of the number and this bit is always copied into `mybig[0]`.)

During the outer while loop, `msu` will move to the next `ulong` whenever the current `ulong` is 0. Eventually we reach the rightmost `ulong`. When we are there and this rightmost `ulong` is also zero, we know, that all `ulong`s are 0 and thereforr the whole number is 0.

Let's have a look at the inner loop:

```
    ulong mod = 0;
    foreach (i;msu .. count)
    {
        mybig[i] |= mod << 60; // (a)
        mod = mybig[i] % 10;   // (b)
        mybig[i] /= 10;        // (c)
    }
```

This is a division with reminder for our long number. To understand it, look at it, if it where a number in base 2^{60}: Every `ulong` represents a single digit of this number and we now do written division by 10: We first look at the leftmost digit, that's the one where `msu` points to. We divide that number by 10 (c). Before that we need to save the reminder (b) of that division.

Now we need to attach the reminder of that round to the left of the next `ulong`. Attaching to the left means: We multiply that value by 2^{60} and add it. That's what (a) does. Please note, that on the first invocation `mod==0` and therefore (a) does nothing. Here we see the benefit of only using 60 bits of the `ulong`s. We know, that `mod` can be at most 4 bits and that fits perfectly into these 4 reserved bits.

We continue with this loop, until we reach the right end of our chain of `ulong`s. After the inner loop finishes, `mod` contains the remainder we were looking for. Finally, we save it in `buffer` and advance `left` on step to the left.

After this loop is completed, we have got all integral digits of the number safed in `buffer`.

```
if (f.precision>0 || f.flHash) buffer[right++] = '.';

next = roundType.ZERO;
```

Finally, we add the dot at the right end of our series of digits. We don't do that, when `precision==0`, because that means, no fractional digits should be printed. But, if the hash-flag is set, we are asked to add that digit even in that case.

We allready know, that all digits right of the dot will be zeros, so no rounding will be needed.

Second algorithm, only fractional bits.

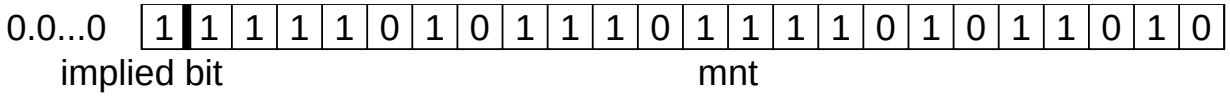
This algorithm is the opposite of the first algorithm, but quite similar. At the end it's more complicated though, because here we need to get the rounding right.

Fractions are converted from binary to decimal by repeated multiplication of the number by 10 and writing down the overflow. As an example take the number 0.10011_2:

$$\begin{aligned} 0.10011 \times 1010 &= 101.1111 \\ 0.1111 \times 1010 &= 1001.011 \\ 0.011 \times 1010 &= 11.11 \\ 0.11 \times 1010 &= 111.1 \\ 0.1 \times 1010 &= 101.0 \end{aligned}$$

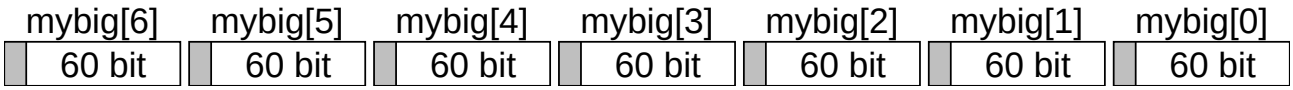
The overflows are 101, 1001, 11, 111, 101, which are 5, 9, 3, 7 and 5 in decimal. So $0.10011_2 = 0.59375_{10}$.

Again we've got a very large number. This time it looks like this:



The number of zeros is one less than `-exp`. The one, we have to subtract here is the implied bit.

For representing this number, we use again a chain of `ulong`s. And again we only use 60 digits of each. What's different: We fill up from left to right this time. Or in other words: This time we count from right to left. This might first look confusing, but later it makes the algorithm a little bit simpler.




```
int count = (T.mant_dig - exp - 2) / 60 + 1;

ulong[] mybig = bigbuf[0 .. count];

int upper = 60 - (-exp - 1) % 60;
if (upper < T.mant_dig)
{
    mybig[0] = (mnt & ((1L << (T.mant_dig - upper)) - 1)) << 60 - (T.mant_dig - upper);
```

```
        mybig[1] = mnt >> (T.mant_dig - upper);
    }
    else
        mybig[0] = mnt << (upper - T.mant_dig);
```

Again we first calculate the number of needed `ulong`s. It's the number of digits of the mantissa (including the leeding bit) plus the number of zeros (`-exp-1`). After this we put the mantissa into the appropriate `ulong`s.

```
    buffer[--left] = '0';

    if (f.precision>0 || f.flHash) buffer[right++] = '.';
```

Bevor calculating the fractional digits, we add the 0 left to the dot and the dot itself, at least, if we need it.

Now we start the loop which calculates the digits. It's quite similar to the loop in the first algorithm. We start, by analysing the inner loop:

```
        ulong over = 0;
        foreach (i;lsu .. count)
        {
            mybig[i] = mybig[i] * 10 + over;
            over = mybig[i] >> 60;
            mybig[i] &= (1L << 60) - 1;
        }
```

This time, it's straight forward: We multiply the number by 10 and add the carry called `over` here. The next carry are the leftmost four bits, which we safe. And finally we delete the old carry. At the end `over` contains the digit we are looking for.

```
    int lsu = 0;
    while ((lsu < count - 1 || mybig[$ - 1]!=0) && right-start - 1 < f.precision)
    {
        ulong over = 0;
        foreach (i;lsu .. count)
        {
            mybig[i] = mybig[i] * 10 + over;
            over = mybig[i] >> 60;
            mybig[i] &= (1L << 60) - 1;
        }
        if (mybig[lsu] == 0)
            ++lsu;

        buffer[right++] = cast(byte) ('0'+over);
    }
```

The outer loop is also similar: At the end, we add the digit to buffer, this time moving to the right. And whenever the rightmost `ulong` of our number get's zero, we move `lsu`, the least significant ulong, one step further to the left.

The condition for ending the loop is a little bit more complicated. Again, if our number consists only of zeros, we stop. But this time, we can stop earlier, if we got already all digits we need for the precision stated. Therefore we also have to check for that.

Unfortunately, this time we are not done yet: We need to find out, if we have to round up or down:

```
if (lsu >= count - 1 && mybig[count - 1] == 0)
    next = roundType.ZERO;
else if (lsu == count - 1 && mybig[lsu] == 1L << 59)
    next = roundType.FIVE;
else
{
    ulong over = 0;
    foreach (i;lsu .. count)
    {
        mybig[i] = mybig[i] * 10 + over;
        over = mybig[i] >> 60;
        mybig[i] &= (1L << 60) - 1;
    }
    next = over >= 5 ? roundType.UPPER : roundType.LOWER;
}
```

Rounding depends on the next digit. There are four cases, we need to distinguish: First, all remaining digits are zeros. Second, the next digit is a 5 and all other digits are zeros. Both cases can be directly checked.

If neighter holds, we need to calculate the next digit. If this digit is ≥ 5, we tend to round up (depending on the rounding mode) and else we tend to round down.

Note: The rounding is done later, for simplicity. We could omit this steps here sometimes, because, when we already know, that we will round down anyway, we do not have to distinguish these cases. Might be a slight improvement left for later.

Third algorithm, integer and fractional part fit into an ulong each.

This time, we just split the mantissa into two parts, the integer part and the fractional part:

```
ulong int_part = mnt >> (T.mant_dig - 1 - exp);
ulong frac_part = mnt & ((1L << (T.mant_dig - 1 - exp)) - 1);
```

Next, we first calculate the integer part. It's just divide by 10 and safe:

```
if (int_part == 0)
    buffer[--left] = '0';
else
    while (int_part>0)
    {
        buffer[--left] = '0'+(int_part%10);
        int_part /= 10;
    }
```

Next, we add the dot (if needed):

```
if (f.precision>0 || f.flHash)
    buffer[right++] = '.';
```

Now we continue with the fractional part. It's again multiplication by 10 and safe; stop when zero or we reached `precision`.

```
while (frac_part != 0 && right-start-1<f.precision)
{
    frac_part *= 10;
    buffer[right++] = cast(byte)('0'+(frac_part >> (T.mant_dig - 1 - exp)));
    frac_part &= ((1L << (T.mant_dig - 1 - exp)) - 1);
}
```

And finally we care about rounding, which is quite similar to the second algorithm:

```
if (frac_part == 0)
    next = roundType.ZERO;
else
{
    frac_part *= 10;
    auto nextDigit = frac_part >> (T.mant_dig - 1 - exp);
    frac_part &= ((1L << (T.mant_dig - 1 - exp)) - 1);

    if (nextDigit == 5 && frac_part == 0)
        next = roundType.FIVE;
    else if (nextDigit >= 5)
        next = roundType.UPPER;
    else
        next = roundType.LOWER;
}
```

Phew, we are through the hard part.

Now, we need to find out, if we have to round up or down. Only, if we have to round up, we have to do something. The decision on this depends on the rounding mode the user wishes, the rounding type we found above and in case we use the mode `toNearestTiesToEven` also from the last digit.

The modes up, down and toZero are simple:

```
if (rm == RoundingMode.up)
    roundUp = next != roundType.ZERO && sgn != "-";
else if (rm == RoundingMode.down)
    roundUp = next != roundType.ZERO && sgn == "-";
else if (rm == RoundingMode.toZero)
    roundUp = false;
else
    ...
```

When rounding up or down, we have to be aware if the number is negative. In this case we have to do the opposite of what we would do with a positiv number.

Rounding to nearest is a little bit more complicated, especially, because IEEE allowes two different implementations:

```
    roundUp = next == roundType.UPPER;

    if (next == roundType.FIVE)
    {
        if (rm == RoundingMode.toNearestTiesAwayFromZero)
            roundUp = true;
        else
        {
            auto last = buffer[right-1];
            if (last == '.') last = buffer[right-2];
            roundUp = last % 2 != 0;
        }
    }
```

First, we do the general case, that is, rounding up only, if we are in the upper half. In case we hit the middle, that ist we have roundType FIVE. We need to use the tie. Ties away from zero is simple. Ties to even needs us to peek upon the last digit and check if this digit is even. In some cases, the last symbol is a dot. In this case we need to jump over it, of course.

Before doing the rounding, we add the remaining fractional zeros to fill precision, if we didn't do so yet:

```
    if (f.precision>0 || f.flHash)
    {
        buffer[right .. f.precision + start + 1] = '0';
        right = f.precision + start + 1;
    }
```

Now we can do the rounding up if necessary. This is just done by adding 1 and continuing to the left if we hit an overflow. We also need to be aware of the dot here.

Finally, it might happen, that we have to add a 1 at the very left, wen rounding numbers like 999.999 up:

```
    foreach_reverse (i;left .. right)
    {
        if (buffer[i] == '.') continue;
        if (buffer[i] == '9')
            buffer[i] = '0';
        else
        {
            buffer[i]++;
            goto printFloat_done;
        }
    }
    buffer[--left] = '1';
printFloat_done:
```

Now we are almost done. What's still missing is the sign (if needed) and some padding with spaces or zeros. Again if needed. It's a little bit fuzzy, but straight forward. I'd like to point you back to the PR for this.

There is one thing, we skipped: The calculation of the sizes of the buffer. So ging back, what we do there is to calculate the amount of digits on the right of the dot (which is given in `precision`) plus 1 for the dot:

```
auto max_right = f.precision + 1;
```

Next, we do the same for the left. That's a little bit more tricky, because we need to estimate the number of dezimal digits. Luckily we know the number of binary digits is exactly `exp`. To get the number of dezimal digits, we just have to divide this by $\log_2(10)$, which is about 3.32. It's only an estimate and make sure with this, that we are on the safe side. It might happen, that we reserve one byte more, than needed, but that's fine. See the comments in the source code for some more details here.

```
auto max_left = exp > 0 ? to!int(exp / 3.32) + 3 : 2;
```

We are not completely done yet: Some of the flags influence the length of the buffer too. Because of this, we might have to adjust `max_left` and `max_right`:

```
if (!f.flDash)
    max_left = max(max_left, f.width - max_right + 2);

if (exp > 0 && f.flDash)
    max_right = max(max_right, f.width - to!int(exp / 3.33) - 1);
else if (f.flDash)
    max_right = max(max_right, f.width - 1);
```

Thanks for reading until here. :-) – Berni