# 20. How does the binary number system work?

**Brief summary:** The binary number system is used to represent numbers using only two digits - 0 and 1. For example, the number 13 (in the decimal number system) is 1101 in the binary number system. All data in a computer's memory is stored as sequences of bits, and so are all numbers.

The binary number system is used to represent numbers using only two digits - 0 and 1. For example, the number **13** (in the **decimal** number system) is **1101** in the **binary** number system.

As you probably know, every piece of data is stored in the computer's memory as a series of bits. Bit is the smallest unit of information and it can only have two values: 0 or 1. That means, every information we want to store in the computer's memory - a number, string, a complex object, or an entire program - is, in the end, stored as a series of zeros and ones. In this lecture, we will focus on numbers.

As we mentioned, the binary number system represents numbers as zeros and ones, so it fits perfectly how data is stored in a computer's memory. You may think that the binary number system is not something you need to understand, as it all happens under the hood. After all, there are programmers all around the world who have no idea how the binary number system works and they are doing fine.

But there are a lot of aspects of programming that are affected by how binary numbers work, and it's not possible to understand some of the programming caveats without knowing the binary system at all. If you are not convinced, let me give you a little spoiler from the next lecture: we will talk about how a banking application's client can lose all protection granted by daily transaction limits and in the end, have the account cleaned out by someone who accessed it illegally. It could be avoided if the programmer understood how operations on binary numbers work.

All right. Before we try to understand the binary number system, let's do so with the system we use on daily basis - the **decimal number system**. The base of this system is number 10. Actually, you can build a valid number system based on any number larger than 0, but 10 was probably most natural for the human race as we have 10 fingers, and we started our journey of understanding mathematics by counting them.

All right. Let’s consider the following decimal number:

**831**

You probably don’t need much explaining here - you simply know what this number is. You can imagine what it means to have 831 dollars (or any other currency you use), a folder with 831 pictures, or a book with 831 pages. We are so used to this system that we don’t even think about the numbers - we simply see them and know by instinct what they mean. But let’s break it down. Each digit has its place. The further to the left it is, the more significant it is - it means, it carries more “weight” of the number. 8 here means 800, 3 means 30 while 1 simply means 1. We could mark each digit with an index, counting from right to left:

$$\overset{2}{8}\;\overset{1}{3}\;\overset{0}{1}$$

Now, for each of the digits, we want to calculate 10 to the power of the index multiplied by the digit itself.

$$\overset{2}{8}\;\overset{1}{3}\;\overset{0}{1}$$

$$8*10^{2} \quad 3*10^{1} \quad 1*10^{0}$$

The sum of those numbers is the final number we want to represent. Let’s make sure of that. 8*100 + 3*10 + 1*1 is 831. (Remember that any number to the power of zero is 1).

Now it is clear why numbers most to the left are most significant - because we will multiply them by 10 to the largest power.

Great. We now understand exactly how the decimal number system works. Let's move on to the **binary number system**. It actually works almost the same. The only difference is the base of the system. It will not be 10, but 2.

Let's consider this number:

**1101**

This time you probably don't "feel" what the number means, but don't worry. We will figure it out in a second. Let's start the same as before - by marking each digit with its index, starting from the right.

3 2 1 0

**1101**

In the decimal number system, we calculated the powers of 10 and then multiplied them by the digit itself. Here it's the same, but we calculate the powers of 2.

3 2 1 0

**1101**

$1*2^3$ $1*2^2$ $0*2^1$ $1*2^0$

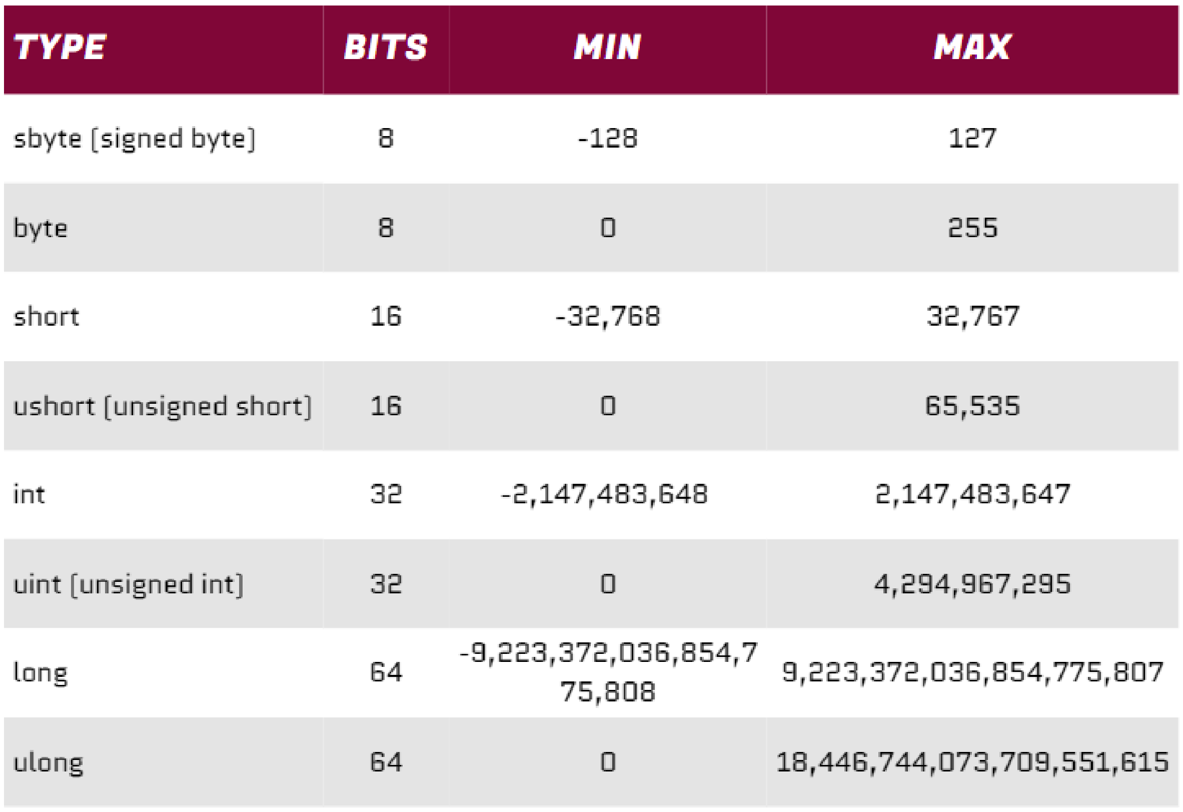
Let's calculate the sum. It's 8 + 4 + 0 + 1, which gives 13. That means, 1101 in the binary number system is 13 in the decimal number system.

Great. This gives us the basics that are needed to understand some operations related to programming.

The important thing to realize is that on a **limited number of bits we can store a limited number** (the same as in a decimal number system - for example the biggest number represented with 3 digits is 999). For example, with 4 bits the largest number that can be represented is 15 (because if each bit is set to one, then the number is 8 + 4 + 2 + 1 = 15). Each numeric type in C# occupies a certain number of bits in the memory. For example, an integer takes 32 bits. The largest number we can represent with int is 2147483648, which is a little over two billion.

And here is something interesting - this number is actually 2 to the power of 31, not 32! So what happened with one bit? Well, remember that with integers we can also represent negative numbers. This one bit is saved to store information whether the number is negative or not, which leaves us 31 bits for the actual number.

Here are sizes and ranges of the integral numeric types used in C#:



| TYPE | BITS | MIN | MAX |
|---|---|---|---|
| sbyte [signed byte] | 8 | -128 | 127 |
| byte | 8 | 0 | 255 |
| short | 16 | -32,768 | 32,767 |
| ushort [unsigned short] | 16 | 0 | 65,535 |
| int | 32 | -2,147,483,648 | 2,147,483,647 |
| uint [unsigned int] | 32 | 0 | 4,294,967,295 |
| long | 64 | -9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 |
| ulong | 64 | 0 | 18,446,744,073,709,551,615 |

All right. There is one more thing we must understand. Since each numeric type has its size limit and it simply can't represent a number that is larger, what happens when some arithmetic operation exceeds this limit?

Well, in such situations, something quite interesting happens. For example, if I add 2 billion to two billion when operating on ints, I will get the result of -294967296. In this lecture, I will only explain to you how it works. In the next one, we will learn how to handle such situations when programming.

Before we can understand what happens, we must understand how adding binary numbers work. But as before, let's start with decimal numbers for simplicity:

$$\begin{array}{r} 831 \\ +\ 461 \\ \hline 1292 \end{array}$$

You probably know this technique of adding numbers. If not, please read this article first:
[https://www.tutorialspoint.com/add_and_subtract_whole_numbers/addition_of_two_2digit_numbers_with_carry.htm](https://www.tutorialspoint.com/add_and_subtract_whole_numbers/addition_of_two_2digit_numbers_with_carry.htm)

With binary numbers, it works the same. Let's add binary 13 to binary 15.
Remember, 13 is 1101 and 15 is 1111:

$$\begin{array}{r} 1101 \\ +\ 1111 \\ \hline \end{array}$$

First, we add numbers from the first column from the right. 1+1 is 2, but we can't use 2 in the binary numbers system. That means, we need to carry it over to the next column. We will write 0 in the first column of the result because the modulo of the sum we calculated (2) and the base of the system (also 2) is 0.

$$\begin{array}{cccccc} & & & 1 & & \\ & 1 & 1 & 0 & 1 \\ + & 1 & 1 & 1 & 1 \\ \hline & & & & 0 \end{array}$$

Now the second column. Again, the sum is 2, so we carry over to the next column again.

$$\begin{array}{cccccc} & & 1 & & & \\ & 1 & 1 & 0 & 1 \\ + & 1 & 1 & 1 & 1 \\ \hline & & & 0 & 0 \end{array}$$

The third column. Now the sum is 3. We carry over 1 to the next column, and we leave 1 in the result. This is because the modulo of the sum we calculated (3) and the base of the system (2) is 1.

$$\begin{array}{r} 1\phantom{101} \\ 1101 \\ +\;1111 \\ \hline 100 \end{array}$$

Finally, the fourth column. The sum is 3 again, so we carry over 1, and we leave 1 in the result:

$$\begin{array}{r} 1\phantom{1101} \\ 1101 \\ +\;1111 \\ \hline 1100 \end{array}$$

It turned out that we actually need the fifth column to fit the 1 that we carried from the fourth column. This time it’s simple. The sum is 1 and we add it to the result:

$$\begin{array}{r} 1101 \\ +\;1111 \\ \hline 11100 \end{array}$$

All right! We have our result. It’s 16+8+4+0+0 = 28. This is correct because 13 + 15 is also 28.

But notice a very important thing - we needed to use one more digit to represent this number. Now, let’s go back to thinking about computers. If we had a numeric type that only has 4 bits, it would simply not be able to hold the result we had. So what would happen? Well, the last, most significant bit would just be discarded. And the actual result the computer could see would not be 11100 which is 28, but 1100 which is 12 - something completely different and simply wrong from the arithmetics point of view.

Now you know why adding two billion to two billion gave some weird number before. If you are curious why it was negative, remember that the most significant bit represents a sign, so if it happens to be 1, then C# will interpret the whole number as negative (0 means positive number, 1 means negative).

All right. That lecture was touching very low-level topics, but now you understand the basics of the binary number system. In the next lecture, we will talk about how it affects our everyday programming.

**Bonus questions:**

- "**What is the decimal representation of number 101?**"
  *It’s 5 because it’s 2 to the power of zero plus two to the power of 2, which gives 1 + 4 = 5.*

- "**Why arithmetic operations in programming can give unexpected results, like for example adding two large integers can give a negative number?**"
  *Because there is a limited number of bits reserved for each numeric type, for example for integer it’s 32 bits. If the result of the arithmetic operation is so large that it doesn’t fit on this amount of bits, some of the bits of the result will be trimmed, giving an unexpected result that is not valid.*