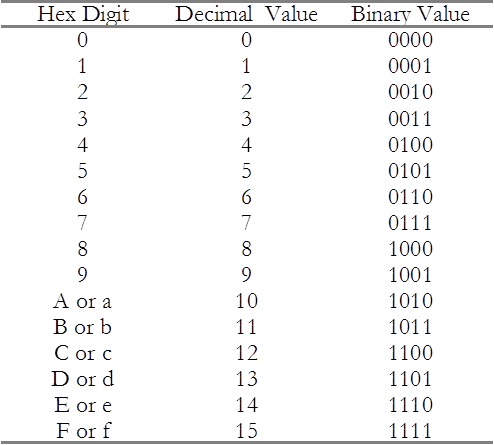
To eliminate confusion between decimal numbers and binary numbers, we will put a subscript 2 after the number to mean binary. Because the memory on most computers is 8 bits wide, most of the binary numbers stored in the computer will have 8, 16, or 32 bits. An 8-bit number is called a **byte**, and a 16-bit number is called a **halfword**. A 32-bit number is called a **word**. For example, the 8-bit binary number for decimal integer 106 is

011010102 = 0•27 + 1•26 + 1•25 + 0•24 + 1•23 + 0•22 + 1•21 + 0•20 = 64+32+8+2 = 106

Just like decimal and binary, each hexadecimal digit has a **place** and a **value**. In this case, the place is a power of 16 and the value is a digit selected from the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. On most computers, it doesn't matter if the hexadecimal digit is written as upper case or lower case {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f}.

A hexadecimal number is often abbreviated as "hex". The basis for a hexadecimal number  is 160, 161, 162, 163,... or 1, 16, 256, 4096,... Hexadecimal is a combination of its digits multiplied by powers of 16. To eliminate confusion between various formats, we will put a 0x or a $ before the number to mean hexadecimal. In C, we will use the 0x prefix to specify hexadecimal.

A **nibble**is defined as 4 binary bits, or one hexadecimal digit. Each value of the 4-bit nibble is mapped into a unique hex digit, as shown in Table 2.1.

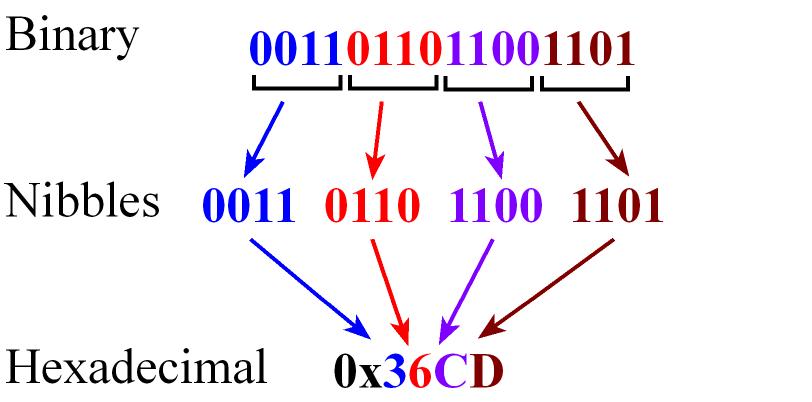


 There are 16 hexadecimal digits.

As illustrated in the figure below, to convert from binary to hexadecimal we can:

       1) Divide the binary number into right justified nibbles,

       2) Convert each nibble into its corresponding hexadecimal digit.

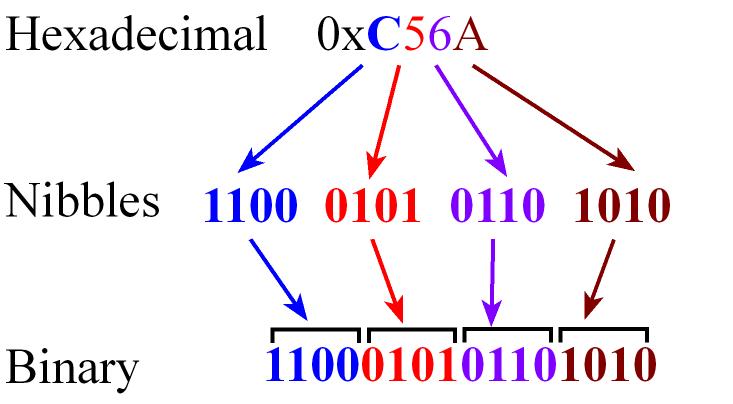


*Notice the mapping between 4 binary bits and one hex digit.*

As illustrated in the figure below, to convert from hexadecimal to binary we can:

       1) Convert each hexadecimal digit into its corresponding 4-bit binary nibble,

       2) Combine the nibbles into a single binary number.



*Notice that each hex digit maps into 4 binary bits.*

**Precision** is the number of distinct or different values. We express precision in alternatives, bytes, or binary bits. **Alternatives** are defined as the total number of possibilities. For example, an 8-bit number format can represent 256 different numbers. An 8-bit **digital to analog converter** (DAC) can generate 256 different analog outputs. An 8-bit **analog to digital converter** (ADC) can measure 256 different analog inputs.

A **byte**contains 8 bits as shown in the figure below, where each bit b7,...,b0 is binary and has the value 1 or 0. We specify b7 as the **most significant bit** or MSB, and b0 as the least significant bit or LSB.

https://courses.edx.org/asset-v1:UTAustinX+UT.6.03x+1T2016+type@asset+block/image003.gif

*A byte has 8 bits.*

If a byte is used to represent an unsigned number, then the value of the number is  
N = 128•b7 + 64•b6 + 32•b5 + 16•b4 + 8•b3 + 4•b2 + 2•b1 + b0

Numbers are used outside of computers. Sometimes numbers are used in a continuous fashion with an infinite or near infinite number of possibilities. Time, distance, temperature are examples of a continuous and infinite or near infinite parameters. However, sometimes we use a finite number of discrete integers to count or label objects. There are 12 months in a year, we have 10 fingers, there are 52 cards in a deck, and 5000 men in a Roman legion. For each of these discrete examples, we can think of **precision** as the number of alternatives from which we can select an object. For example, if we wish to select a month, we can choose from 12 possibilities.

|  |  |  |
| --- | --- | --- |
| Example | Precision in alternatives | Precision in bits |
| Fingers | 10 | a little more than 3 |
| Months | 12 | less than 4 |
| Human vertebrae | 33 | a little more than 5 |
| Cards in a deck | 52 | between 5 and 6 |
| Days in a year | 364-365 | between 8 and 9 |
| Minutes in a day | 1440 | between 11 and 12 |
| Men in a legion | 5000 | a little more than 12 |

**Observation:** If the least significant binary bit is zero, then the number is even.

**Observation:** If the right-most n bits (least sign.) are zero, then the number is divisible by 2n.

**Observation:** Consider an 8-bit unsigned number system. If bit 7 is low, then the number is between 0 and 127, and if bit 7 is high then the number is between 128 and 255.

One of the first schemes to represent signed numbers was called **one’s complement**. It was called one’s complement because to negate a number, we complement (logical not) each bit.

For example, if 25 equals 000110012 in binary, then –25 is 111001102. An 8-bit one’s complement number can vary from ‑127 to +127. The most significant bit is a sign bit, which is 1 if and only if the number is negative. The difficulty with this format is that there are two zeros +0 is 000000002, and –0 is 111111112. Another problem is that one’s complement numbers do not have basis elements. These limitations led to the use of two’s complement.

The **two’s complement** number system is the most common approach used to define signed numbers. It is called two’s complement because to negate a number, we complement each bit (like one’s complement), then add 1.

For example, if 25 equals 000110012 in binary, then –25 is 111001112. If a byte is used to represent a signed two’s complement number, then the value of the number is

N = -128•b7 + 64•b6 + 32•b5 + 16•b4 + 8•b3 + 4•b2 + 2•b1 + b0

The **basis** elements of an 8-bit signed number are {-128, 64, 32, 16, 8, 4, 2, 1}.

**Observation:** The most significant bit in a two’s complement signed number will specify the sign.

**Observation:** To take the negative of a two’s complement signed number we first complement (flip) all the bits, then add 1.

A **word** on the ARM Cortex M will have 32 bits. Consider an unsigned number with 32 bits, where each bit b31,...,b0 is binary and has the value 1 or 0. If a 32-bit number is used to represent an unsigned integer, then the value of the number is

N = 231 • b31 + 230 • b30 + ... + 2•b1 + b0 = sum(2i • bi) for i=0 to 31

There are 232 different unsigned 32-bit numbers. The smallest unsigned 32-bit number is 0 and the largest is 232-1. This range is 0 to about 4 billion.

A **halfword** or **double byte**contains 16 bits, where each bit b15,...,b0 is binary and has the value 1 or 0, as shown in Figure 2.4.

https://courses.edx.org/asset-v1:UTAustinX+UT.6.03x+1T2016+type@asset+block/image004.gif

*FA halfword has 16 bits.*

Similar to the unsigned algorithm, we can use the basis to convert a decimal number into signed binary. We will work through the algorithm with the example of converting –100 to 8‑bit binary: We start with the most significant bit (in this case –128) and decide do we need to include it to make –100? Yes (without –128, we would be unable to add the other basis elements together to get any negative result), so we set bit 7 and subtract the basis element from our value. Our new value equals –100 minus –128, which is 28. We go the next largest basis element, 64 and ask, “do we need it?” We do not need 64 to generate our 28, so bit 6 is zero. Next we go the next basis element, 32 and ask, “do we need it?” We do not need 32 to generate our 28, so bit 5 is zero. Now we need the basis element 16, so we set bit 4, and subtract 16 from our number 28 (28-16=12). Continuing along, we need basis elements 8 and 4 but not 2, 1. Putting it together we get 100111002 (which means -128+16+8+4).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number | Basis | Need it | bit | Operation |
| -100 | -128 | **yes** | bit 7=1 | subtract -100 - -128 |
| 28 | 64 | no | bit 6=0 | none |
| 28 | 32 | no | bit 5=0 | none |
| 28 | 16 | **yes** | bit 4=1 | subtract 28-16 |
| 12 | 8 | **yes** | bit 3=1 | subtract 12-8 |
| 4 | 4 | **yes** | bit 2=1 | subtract 4-4 |
| 0 | 2 | no | bit 1=0 | none |
| 0 | 1 | no | bit 0=0 | none |

Example conversion from decimal to signed 8-bit binary.

A second way to convert negative numbers into binary is to first convert them into unsigned binary, then do a two’s complement negate. For example, we earlier found that +100 is 011001002. The two’s complement negate is a two-step process. First we do a logic complement (flip all bits) to get 100110112. Then add one to the result to get 100111002.

A third way to convert negative numbers into binary uses the number wheel. Let *n* be the number of bits in the binary representation. We specify **precision**, *M*=2^*n*, as the number of distinct values that can be represented. To convert negative numbers into binary is to first add *M* to the number, then convert the unsigned result to binary using the unsigned method. This works because binary numbers with a finite *n* are like the minute-hand on a clock. If we add 60 minutes, the minute-hand is in the same position. Similarly if we add *M* to or subtract *M* from an *n*-bit number, we go around the number wheel and arrive at the same place. This is one of the beautiful properties of 2's complement: unsigned and signed addition/subtraction are same operation. In this example we have an 8-bit number so the precision is 256. So, first we add 256 to the number, then convert the unsigned result to binary using the unsigned method. For example, to find –100, we add 256 plus –100 to get 156. Then we convert 156 to binary resulting in 100111002. This method works because in 8-bit binary math adding 256 to number does not change the value. E.g., 256-100 has the same 8-bit binary value as –100.

When dealing with numbers on the computer, it will be convenient to memorize some **Powers of 2** as shown in the table below.

|  |  |
| --- | --- |
| exponent | decimal |
| 20 | 1 |
| 21 | 2 |
| 22 | 4 |
| 23 | 8 |
| 24 | 16 |
| 25 | 32 |
| 26 | 64 |
| 27 | 128 |
| 28 | 256 |
| 29 | 512 |
| 210 | 1024 about a thousand |
| 211 | 2048 |
| 212 | 4096 |
| 213 | 8192 |
| 214 | 16384 |
| 215 | 32768 |
| 216 | 65536 |
| 220 | about a million |
| 230 | about a billion |
| 240 | about a trillion |

Some powers of two that will be useful to memorize.

We will use **fixed-point** numbers when we wish to express values in our computer that have noninteger values. A fixed-point number contains two parts. The first part is a variable integer, called *I*. The variable integer will be stored on the computer. The second part of a fixed-point number is a fixed constant, called the**resolution** *Δ*.

The fixed constant will NOT be stored on the computer. The fixed constant is something we keep track of while designing the software operations. The value of the number is the product of the variable integer times the fixed constant. The integer may be signed or unsigned. An unsigned fixed-point number is one that has an unsigned variable integer. A signed fixed-point number is one that has a signed variable integer.

*Value* = *VariableInteger* \* *FixedConstant* = *I*\**Δ*

The **precision** of a number system is the total number of distinguishable values that can be represented. The precision of a fixed-point number is determined by the number of bits used to store the variable integer. On most microcontrollers, we can use 8, 16, or 32 bits for the integer. With **binary fixed point** the fixed constant is a power of 2.

For example, consider a binary fixed-point number system where the resolution is 2-4. The resolution is not stored on the computer, just the integer *I*.

|  |  |
| --- | --- |
| Integer *I* | Value |
| 0 | 0/16 = 0 |
| 1 | 1/16 = 0.0625 |
| 2 | 2/16 = 0.125 |
| 3 | 3/16 = 0.1875 |

For example, consider a decimal fixed-point number system where the resolution is 10-2. The resolution is not stored on the computer, just the integer *I*.

|  |  |
| --- | --- |
| Integer *I* | Value |
| 0 | 0/100 = 0 |
| 1 | 1/100 = 0.01 |
| 2 | 2/100 = 0.02 |
| 3 | 3/100 = 0.03 |