

# Module 1. Intro to Number Representations

## Topics

- How do we store (or “encode”) information in digital systems?
- Specifically: how do we store numbers?

## First things first: Remembering Digital Logic

Before we can talk about how computing systems are built, we first need to talk about their basic building block: digital logic. In digital logic, information is represented in binary *bits*.

Digital logic defines how we can process information using bits:

*First things first:*  $n$  bits differentiate among  $2^n$  things.

**Terminology:** 1 byte = 8 binary digits = 8 bits (e.g. 10010011)

½ byte = 1 nibble = 4 bits

1 word = 2 (or more) bytes --> MSP430 word = 2 bytes

1 double word = 2 words (4 bytes on MSP430)

In computers, *information* and *memory space* is organized in to multiples of bytes.  
But what do the bytes mean?

## The meaning of bits and bytes assigned by convention!

>> Under a given coding convention, a byte can represent up to  $2^8 = 256$  things

For example, 1 byte (8 bits) could encode:

- A letter in an alphabet
- One or more decimal numbers
- The state of eight individual things (one per bit)
- An *instruction* that tells the CPU to do something:

We call these conventions **encoding formats**. They represent a kind of contract on how data will be stored and used. As programmers, it is up to us to assign meaning to those bits—which defines what operations we perform on them.

## **Conversion between Bases and Formats: Binary**

### Positional Number Systems

We write numbers in a positional system, which can be defined as:

For binary numbers, we can write this definition as:

Unsigned integers = All bits used to convey magnitude (whole numbers  $\geq 0$ )

### Decimal to Binary Conversion – Successive Division

Note: To differentiate numbers in different formats, we use notation to denote the radix used we write it. For binary:  $1010_2$  or 1010b; decimal:  $1010_{10}$  or 1010d (or just 1010)

## Hexadecimal: A common way to write binary numbers

Since working in binary can be cumbersome, we often write numbers in *hexadecimal*, which is base 16.

Simple rule for conversion:

--> 1 Hex character represents values from 0 to 15d using digits 0 – Fh

DEC	BIN	HEX		DEC	BIN	HEX
0	0000	0		8	1000	8
1	0001	1		9	1001	9
2	0010	2		10	1010	A
3	0011	3		11	1011	B
4	0100	4		12	1100	C
5	0101	5		13	1101	D
6	0110	6		14	1110	E
7	0111	7		15	1111	F

*Ex. 158d =*

**If you memorize anything in this class, memorize these!**

**Notation:** Numbers in hex are written as 1010h or 0x1010

***Conversion between hex and binary is piece of cake!*** Just convert each hex digit to a binary nibble...

$$1001\ 1110b = 158d$$

Or vice versa:

$$8AC4h = \quad 8 \quad A \quad C \quad 4$$

**Note:** A modern computer always stores information in binary form. Writing in hex is just a faster way for us to read and write these numbers—the machine's representation is still binary.

**How do we store negative numbers?**

**One way:** Sign Magnitude integers =  $n-1$  bits used to convey magnitude with “most significant bit” or MSB used for sign. Convention: 0 = +, 1 = -

**Note:** This format has 2 representations of 0 = +0 and -0 !

**Another way:** Two's Complement integers = More common format for signed integers. For  $n$  bits, values range from  $-2^{(n-1)}$  to  $2^{(n-1)}-1$

*How it works:*

**Positive numbers:** Follow same format as unsigned numbers

$$1026 = 0000\ 0100\ 0000\ 0010b = 0402h$$

**Negative numbers:** *Write magnitude, Complement each bit, Add 1*

-15 =

Range of Values for 2's Complement

0111 1111 1111 1111b

0111 1111 1111 1110b

. . .

0000 0000 0000 0000b

. . .

1000 0000 0000 0001b

1000 0000 0000 0000b

*Ex:* Find the 8-bit two's complement representation of 104 and -80

*Ex:* What are the decimal equivalent values of these 2's complement values

0010 0011b

1000 0011b

*Ex:* What decimal value does 8008h represent as an...

(a) unsigned integer (b) 2's comp integer (c) sign-magnitude integer





## What about things that aren't integers?

### Characters

To handle letters and other displayable characters, we need an encoding format to describe how we can represent these values in binary. One very common format for this is ASCII (American Standard Code for Information Exchange), which defines a table of binary codes that represent various characters.

Note: Other formats exist for representing different types of characters (alphabets and character sets for all human languages, emoji, etc.). For information on this, see "Unicode".

### Unicode Examples

Unicode Name	Bit representation	Character
U+00FC LATIN SMALL LETTER U WITH DIAERESIS	C3 BC	ü
U+1F602 FACE WITH TEARS OF JOY	F0 9f 98 82	
U+1F363 SUSHI	F0 9F 8D 83	

### Non-integer Numbers

In future lectures we will talk about representing non-integer data. These are called fixed-point and floating-point data types, which we will cover soon!

## Preview: How is data actually stored in a program?

C defines a set of standard data types to store information. Each datatype has a specific representation, which depends on the compiler and the CPU architecture.

For the MSP430 architecture, the standard datatypes are defined as follows:

```
int a;           // 16-bit two's-complement signed (2 bytes)
unsigned int b;  // 16-bit unsigned integer (2 bytes)
long int c;     // 32-bit signed integer (two's complement) (4 bytes)
char d;        // 8-bit unsigned integer (1 byte)

float e;       // 32-bit IEEE754 single-precision floating point value (4 bytes)
double f;     // 64-bit IEEE754 double-precision floating point value (8 bytes)
```

Note that the types `char`, `float`, and `double` have the same size on all architectures—these are part of the C standard.

We can use these standard datatypes to hold different kinds of information (signed/unsigned numbers, characters, decimal values), or compose more complex types (like arrays or structs).

**Important: The size and type of a variable define the range of values they can represent!**

- The value of a variable CANNOT exceed the fixed size of the variable
- Variables will "overflow" or "roll over" if the value exceeds the variable size!