## 01.1 Memory

Computer memory is a collection of bistable devices—so they can represent only, say 0 or a 1 in each bit—organized as bytes: collections of 8 binary digits or bits. There are $2^8 = 256$ unique bytes. In more modern systems, each byte (n.b. not bit) of memory has a unique address—an identifying code. An important aspect of the C programming language is that it can deal directly with these memory addresses, a relatively low-level functionality.

Memory is not content-specific. It can be used to represent numbers (integers, floating point, signed numbers, etc.), codes (character codes, numeral codes, etc.), and instructions. We must keep track of the meaning of its contents. For instance, a single bit could represent the state of the union: 1 could mean "covfefe" and 0, "dumpsterfire." A less exciting example with two bits representing four directions:

11 = north
00 = south
01 = east
10 = west

### Things you can store in memory

#### Pure binary numbers

Non-negative integers of different magnitudes can be stored as pure binary in memory. Here is an example using one byte or two nibbles:

$$
\begin{aligned}
\texttt{0000 0000}_2 &= 0_{10} \\
\texttt{0000 0001}_2 &= 1_{10} \\
&\vdots \\
\texttt{1111 1110}_2 &= 254_{10} \\
\texttt{1111 1111}_2 &= 255_{10}.
\end{aligned}
$$

So the non-negative integers we can store in one byte are 0–255, of which there are $2^8 = 256$.

But we can use more than one byte to store a non-negative integer in pure binary. If multiple bytes are representing a number, the byte that occurs first (in terms of address) in memory is called the most significant byte (MSB), and the byte that occurs last is called the least significant byte (LSB). The MSB is usually represented as being to the left of the other bytes, and the LSB is typically represented as being to the right.

Here is a list of the total number of possible non-negative integers that can be stored in $n$ bits (formula: $2^n$) for typical values of $n$:

$$
\begin{aligned}
2^8 &= 256 \\
2^{16} &= 65,536 \\
2^{24} &= 16,777,216 \\
2^{32} &= 4,294,967,296.
\end{aligned}
$$

#### 8-bit two's complement signed binary

How can a negative number be stored in memory? A single byte can store 256 unique pieces of information. For decimal numbers, this can range 0 to 255 or (say) −128 to 127.

A very convenient binary representation is called two's complement. A number $x$ has two's complement in $n$ bits of $(2^n - x)_2$; that is, the number of unique numbers representable minus the number, represented in binary. For instance, the 8-bit two's complement of 0110 1000 is[1]

$$
\begin{array}{r}
\overset{11111}{\not 1}0000\,0000 \\
-\,0110\,1000 \\
\hline
1001\,1000
\end{array}
$$

1. The first borrow might seem strange, but it's simply $10_2 - 01_2 = 2_{10} - 1_{10} = 1_{10} = 01_2$.

$01101000_2 = 8_{10} + 32_{10} + 64_{10} = 104_{10}$

$10011000_2 \to 01100111_2 \to 01101000_2$

Below are listed some 8-bit two's complement decimal interpretations of binary numbers.

$$
\begin{aligned}
\texttt{0000 0000}_2 &= 0 \\
\texttt{0000 0001}_2 &= 1 \\
&\vdots \\
\texttt{0111 1111}_2 &= 127 \\
\texttt{1000 0000}_2 &= -128 \\
\texttt{1000 0001}_2 &= -127 \\
&\vdots \\
\texttt{1111 1110}_2 &= -2 \\
\texttt{1111 1111}_2 &= -1
\end{aligned}
$$

As if in Pac-Man, starting from the middle and exiting screen-right, only to appear screen-left—counting "up" loops one back down to negative numbers. Note that positive two's complements are the same as their pure binary counterparts.

There are two more-convenient ways to find the two's complement:

1. switching all bits ($0 \mapsto 1$ and $1 \mapsto 0$), then adding 1 or
2. starting from the right, copying all bits through the first 1 encountered, then switching all thereafter.

Both methods can be seen to always hold from the subtraction definition.

The two's complement of the two's complement of $x$ is $x$; that is, it is its own inverse.

**Example 01.1 −1** **re: two's complement**

Find the two's complement of 0000 0101.

1111 1010 → 1111 1011

If a binary number is interpreted as a two's complement binary number, it is negative if its most significant bit (msbit) is 1.

#### Binary coded decimal (BCD)

A binary coded decimal (BCD) represents each decimal digit with a nibble, so a series of nibbles can represent a decimal number. This leads to slightly less-dense storage, but is still useful for high-precision computation.

nibble 4 bits $2^4 = 16$

**Example 01.1 −2** **re: BCD for rounding error**

Recall that the number 421.73 had an infinitely long binary representation in Example 00.4 - 1. Represent this number in BCD. Let there be an "implied" decimal point, as some encodings define, between the third and fourth nibbles.

4 0100
2 0010
1 0001
7 0111
3 0011
BCD: 0100 0010 0001 0111 0011

#### Floating point

Floating point numbers can represent very large or very small numbers with limited space. It is for computer memory what scientific notation is for a small piece of paper: that is, it represents a number as a mantissa[2] $x$ and an exponent $n$; that is, $x2^n$, where we have used the conventional base of 2.

× 10^n

2. The mantissa is also called the significand or coefficient.

Consider the following illustration of a 32-bit (four-byte) floating point representation.

8-bit signed binary exponent | 24-bit fractional signed binary mantissa

We would interpret this as, for instance,

$$
\underbrace{.1011\cdots}_{\text{24-bit mantissa}} \times 2^{\overbrace{1011\,0110}^{\text{8-bit exponent}}}. \tag{1}
$$

#### Character codes

In addition to numbers, memory can store character codes: encoded alphabetic, special symbols, emojis, etc.

The most common character code is the American Standard Code for Information Interchange (ASCII). It's a 7-bit code, so there are 128 unique character codes.

$2^7 = 128$

It leaves the eighth bit of a byte, "bit seven," the parity bit, to be checked for transmission errors. It works as follows. Set (1) or reset (0) before transmission such that the total number of set (1) bits is either even or odd. If the system is using even parity, an even number of bits are set; or if it's using odd parity, an odd number of bits are set.

For instance, under odd parity, if the byte 1100 1101 is sent and the byte 1100 0101 is received, with its even number of set bits, the receiving system knows there has been a transmission error.

#### Instructions

Instructions are codes that direct the operation of a microprocessor. The myRIO has an ARM Cortex-A9 processor with 32-bit instructions.

**Example 01.1 −3** **re: memory interpretation**

Suppose the following is stored in a byte of memory: 1101 0101 or D5. How might this be interpreted?

$1101\ 0101_2 = 128_{10} + 64_{10} + 4_{10} + 1_{10} = 213_{10}$ unsigned int

$1101\ 0101_2 \to 0010\ 1010_2 \to 0010\ 1011_2 = 1_{10} + 2_{10} + 8_{10} + 32_{10} = 43_{10}$
$= -43_{10}$ 8 bit signed 2s complement

$1101_2 = D_{16}$ $0101_2 = 5_{16} = 5_{10}$

$1101\ 0101_2$ = 'U'

instruction

### Memory organization

In memory, bits are grouped into bytes of eight bits. Each byte is often considered as two nibbles, the contents of each represented by a hexadecimal numeral. For instance, a byte might be represented as follows.

$1101\ 0100_2$

$D4_{16}$

10 A
11 B
12 C
13 D
14 E
15 F

Each byte is given a unique positive integer address distinct from its contents.

$300_{10} = 0000\ 0001\ 0010\ 1100$

| address | contents |
|---|---|
| 0 | |
| 1 | |
| 2 | 0000 0001 |
| 3 | 0010 1100 |
| 4 | |
| ⋮ | |

When storing a multi-byte number, we use the bigendian convention: the MSB is stored at the lower address. The littleendian convention stores the MSB at the higher address.