APPENDIX C

Bits, bytes, and data types

Here, I show you basic representations of bytes and briefly characterize how they comprise data types. Additionally, I show a few novel features of C# 7.0, which can be useful for IoT development.

Binary encoding: integral types

A byte can be interpreted as an array of bits—that is, an array of logical values. Formally, each entry of this array encodes the byte value V as the weighted sum:

$$V = \sum_{i=0}^{N-1} b_i 2^i$$

where b_i denotes the bit value (0 or 1), i stands for the bit index, and N is the number of bits. The base (radix) of 2 comes from the fact that a single bit can possess two values. The summation weights increase with a bit index. For this reason, a bit of index 0 is called the least significant bit (LSB), contrary to the most significant bit (MSB) occupying the largest index. Table C-1 shows several examples of bit representations of the following bytes (N = 8): 39, 127, 168, and 255.

TABLE C-1 Binary representation of the selected integers: 39, 127, 168, and 255

	MSB							**LSB**	**V**
Index	7	6	5	4	3	2	1	0	-
Logical value	0	0	1	0	0	1	1	1	-
Numerical value	0	0	32	0	0	4	2	1	39
Logical value	0	1	1	1	1	1	1	1	-
Numerical value	0	64	32	16	8	4	2	1	127
Logical value	1	0	1	0	1	0	0	0	-
Numerical value	128	0	32	0	8	0	0	0	168
Logical value	1	1	1	1	1	1	1	1	-
Numerical value	128	64	32	16	8	4	2	1	255

You see from Table C-1 that V cannot be larger than $2^8 - 1 = 255$ or smaller than 0. Thus, the byte can encode only positive (unsigned) values. How to cope with negative values? The most popular ways are as follows:

- **Signed magnitude representation** One bit (typically the MSB) stores the sign information. However, this approach includes two ways of representing a zero value.
- **One's complement** Inverting bits (changing 1 to 0 and *vice versa*) of the positive number produces negative values. This solution requires the carry flag.
- **Two's complement** The preceding problems are overcome in the two's complement system, where inverting bits and adding a 1 to the resulting number produces negative vales. In other words, the value encoded by the most significant bit is multiplied by –1 and added to the sum of values encoded using other bits:

$$V_s = \sum_{i=0}^{N-1} b_i 2^i - b_{N-1} 2^{N-1}$$

Thus, a single byte can encode integers ranging from $-2^7 = -128$ to $2^7 - 1 = 127$. Table C-2 shows a few examples of signed bytes encoded using two's complement.

TABLE C-2 Binary representation of the selected signed integers: –128, –74, –39, and 127

	MSB							LSB	V_s
Index	7	6	5	4	3	2	1	0	-
Logical value	1	0	0	0	0	0	0	0	-
Numerical value	1	0	0	0	0	0	0	0	–128
Logical value	1	0	1	1	0	1	1	0	-
Numerical value	–128	0	32	16	0	4	2	0	–74
Logical value	1	1	0	1	1	0	0	1	-
Numerical value	–128	64	0	16	8	0	0	1	–39
Logical value	0	1	1	1	1	1	1	1	-
Numerical value	0	64	32	16	8	4	2	1	127

To encode larger integer values, the bit array is extended to $N = 16$ (`short` and `ushort` datatype), $N = 32$ (`uint` and `int`), and $N = 64$ (`long` and `ulong`). Such data types can store values of ranges specified in Table C-3.

TABLE C-3 Integer ranges for two's complement encoding system (the minimum value of unsigned integers is 0)

N	16	32	64
Minimum value (signed)	–32678	–2147483648	–9223372036854775808
Maximum value (signed)	32767	2147483647	9223372036854775807
Maximum value (unsigned)	65535	4294967295	18446744073709551615

Naturally, you do not need to remember exact limits of integer data types. The specific ranges can be obtained from `MinValue` and `MaxValue` constants of the integral data types. To illustrate this, I wrote a simple UWP app; see the companion code at Appendix C/DataRanges. I implement it as follows:

- I create a custom control, `DataTypeRangesControl` (DataRanges/DataTypeRangesControl), which consists of three `TextBlock` controls that display the data type label along with the minimum and maximum values that the particular type can handle.
- I write the `RangesViewModel` class; see Listing C-1. This class serves as the `ViewModel` for the main view of the DataRanges app. Particular fields of the `RangesViewModel` are bound to the UI, so I do not need to manually rewrite properties.

LISTING C-1 Retrieving the smallest and largest possible values of integer data types

```
public class RangesViewModel
{
    // N=8
    public byte UInt8MinValue { get; set; } = byte.MinValue;
    public byte UInt8MaxValue { get; set; } = byte.MaxValue;
    public sbyte Int8MinValue { get; set; } = sbyte.MinValue;
    public sbyte Int8MaxValue { get; set; } = sbyte.MaxValue;

    // N=16
    public ushort UInt16MinValue { get; set; } = ushort.MinValue;
    public ushort UInt16MaxValue { get; set; } = ushort.MaxValue;
    public short Int16MinValue { get; set; } = short.MinValue;
    public short Int16MaxValue { get; set; } = short.MaxValue;

    // N=32
    public uint UInt32MinValue { get; set; } = uint.MinValue;
    public uint UInt32MaxValue { get; set; } = uint.MaxValue;
    public int Int32MinValue { get; set; } = int.MinValue;
    public int Int32MaxValue { get; set; } = int.MaxValue;

    // N=64
    public ulong UInt64MinValue { get; set; } = ulong.MinValue;
    public ulong UInt64MaxValue { get; set; } = ulong.MaxValue;
    public long Int64MinValue { get; set; } = long.MinValue;
    public long Int64MaxValue { get; set; } = long.MaxValue;
}
```

After you compile and run the DataTypeRanges app on the IoT device or local computer, the data type ranges will be depicted as shown in Figure C-1.

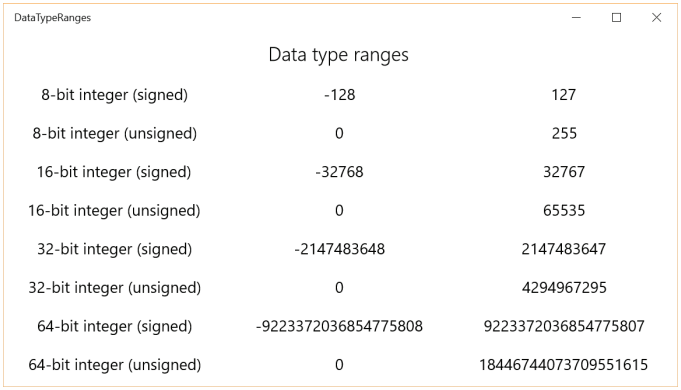

FIGURE C-1 Data type ranges retrieved from the constants fields of `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, and `ulong` data types.

Binary encoding: floating numbers

Typically, floating numbers are represented in a scientific format as follows:

$$V_f = s \times m \times 2^e$$

where s is the sign, m stands for the mantissa (or significand), and e denotes the exponent. Similarly, as in the integral data types, a single bit is used to encode the sign, while the specific amount of bits is used to encode the significand and the exponent. The exponent is stored as an unsigned 8-bit integer with a fixed bias added to it. According to the IEEE 754 standard, this is in a 32-bit precision floating number: $0 < m < 2^{24}$ and $-126 \leq e \leq 127$. This means that 23 bits are used to encode the mantissa, while 8 bits encode the exponent with a bias of 127 (a byte with all zeros is reserved).

In a 64-bit precision floating number, 52 bits encode the mantissa ($0 < m < 2^{53}$) and the remaining 11 bits encode the exponent, which can take values in the range $-1022 < e < 1023$ with a bias of 1023.

If you are new to these concepts, I encourage you to extend the DataTypeRanges app to present minimum and maximum values of the `float` and `double` data types.

Hexadecimal numeral system

Very often in IoT programming, values such as register addresses are represented using a hexadecimal numeral system (HEX). This system uses a base of 16, composed of 10 digits (0,1,2,...) and 6 letters (A=10, B=11, C=12, D=13, E=14 and F=15). Accordingly, the decimal (DEC) integer can be represented in the HEX system as:

$$V_D = \sum_{i=0}^{N-1} h_i 16^i$$

where h_i is one of the available HEX symbols at a position i. Table C-4 contains decimal and HEX representations of the following unsigned integers: 127, 255, 1024, and 56506. LS and MS represent least and most significant elements, respectively.

Hexadecimal values are preceded by the 0x symbol, e.g. 0xFF.

TABLE C-4 Hexadecimal and decimal representations of selected integers: 127, 255, 1024, and 56506

	MS			LS	V_D
Index	3	2	1	0	-
HEX value	0	0	7	F	-
DEC value	0	0	112	15	127
HEX value	0	0	F	F	-
DEC value	0	0	240	15	255
HEX value	0	4	0	0	-
DEC value	0	1024	0	0	1024
HEX value	D	C	B	A	-
DEC value	53428	3072	176	10	56506

Numeral values formatting

Fortunately, you can quickly write an app to perform the conversion between various numeral systems. To this end you use the static `System.Convert.ToString` method with the appropriate number defining the numeral system radix. You can use one of the following values: 2 (binary), 8 (octet), 10 (decimal), or 16 (hexadecimal). The code snippet from Listing C-2 shows how to display an integer in binary, decimal, and hexadecimal format.

LISTING C-2 Presenting the integer value in different numeral systems

```
private static void ValueFormatting()
{
    var myInteger = 12345;

    Debug.Write("BIN: ");
    Debug.WriteLine(Convert.ToString(myInteger, 2));

    Debug.Write("DEC: ");
    Debug.WriteLine(Convert.ToString(myInteger, 10));

    Debug.Write("HEX: ");
    Debug.WriteLine(Convert.ToString(myInteger, 16));
}
```

The above code will produce in the Debug Output the following values:

```
BIN: 11000000111001
DEC: 12345
HEX: 3039
```

Binary literals and digit separator

C# 7.0 includes two more features that are helpful for low-level programming. These are binary literals and the digit separator. Binary literals let you define binary constants, while the digit separator (_) helps you format literals. A sample code snippet utilizing these features appears in Listing C-3, while the corresponding output appears in Listing C-4.

LISTING C-3 Binary literals

```
private static void BinaryLiterals()
{
    var a = 0b0001_1110;
    var b = 0b1000_0110;

    DebugValue(a);
    DebugValue(b);

    DebugValue(b & a);
    DebugValue(a | b);
    DebugValue(a ^ b);
}

private static void DebugValue(int value)
{
    Debug.Write(value.ToString().PadLeft(5));
    Debug.Write(", BIN: " + Convert.ToString(value, 2).PadLeft(8));
    Debug.WriteLine(", HEX: " + Convert.ToString(value, 16).PadLeft(2));
}
```

LISTING C-4 Sample output of the BinaryLiterals method from Listing C-3

```
   30, BIN:    11110, HEX: 1e

  134, BIN: 10000110, HEX: 86

    6, BIN:      110, HEX: 6

  158, BIN: 10011110, HEX: 9e

  152, BIN: 10011000, HEX: 98
```

Note that you will need Visual Studio 2017 to test the above examples. Appendix F, "Setting up Visual Studio 2017 for IoT development," describes how to set up this development environment for C# and IoT programming.