Gulliver

Sandia National Laboratories

Jun 26, 2023

GETTING STARTED

Gulliver is a C# utility package and library engineered for the manipulation of arbitrary sized byte arrays accounting for appropriate endianness and jagged byte length. Functionality includes the as previously unsupported in .NET standard set of boolean algebraic operations, bitwise shifting, and unsigned endian aware mathematical addition, subtraction, and comparison. Gulliver exists to free developers from managing byte ordering and operations at the low level as was previously required by the standard C# language distributions.

Gulliver was of course named for the titular character in Gulliver's Travels, a.k.a. "**Travels into Several Remote Nations of the World. In Four Parts. By Lemuel Gulliver, First a Surgeon, and then a Captain of Several Ships**" by Jonathan Swift, a book that the library author has admittedly not yet read but was pulled from the Computer Science zeitgeist referring to the big-endian versus little-endian nature of byte ordering.

Gulliver originally came to be for the sake of Arcus, a C# library for calculating, parsing, formatting, converting and comparing both IPv4 and IPv6 addresses and subnets. Inherently, by its nature, Arcus needed to do a great deal of byte manipulation. Eventually Gulliver came into a life of its own and it was decided that it should be broken off into its own library.

The API surfaced by Gulliver is not outwardly complex, and the intention is to keep the most common functionality as simple as possible. That simplicity is Gulliver's great strength. Each simple block can be put together to achieve more complex and powerful functionality. We did it so other developers didn't have to. Of note, while we don't believe Gulliver is inefficient, it is certainly not necessarily the most efficient code solution depending on the task at hand. We opted for readability and understanding when developing Gulliver, and elected not to write code that was enigmatic for the sake of efficiency. For example we decided check for invalid method input in an overly optimistic defensive manner, chose enumerable abstractions for the ease of iteration, and chose not to work with C# spans as the added speed would cost us compatibility for older language versions.

CHAPTER

ONE

WHAT IS ENDIANNESS

Endianness[1], at least as far as computing is concerned, is the ordering of bytes within a binary representation of data. The most common representations of endianness are *Big-Endian* and *Little-Endian*. There exist other forms, such as Middle-Endian, but those are beyond the scope of this document.

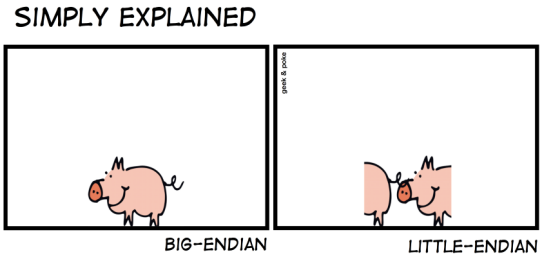

Fig. 1: "Simply Explained"
Comic by Oliver Widder Released under Attribution 3.0 Unported (CC BY 3.0).

1.1 Etymology

The Computer Science terms Big-Endian and Little-Endian were introduced by Danny Cohen[2] in 1980. The key term *endian* has its roots in the novel Gulliver's Travels[3] by Jonathan Swift[4] where within a war occurs between two factions who are fighting over which end of a boiled egg should be opened for eating. The big end or the little end. Unsurprisingly, the same said book was the inspiration for the naming of the Gulliver library.

1.2 Big-Endian

Big-Endian, often referred to as *Network Byte Order*, ordering is left-to-right. Given the representation of an unsigned number in bytes the further to the left that a byte exists the more significant it is.

For example, the decimal value of the unsigned integer 8675309_{10} may be represented as $0x845FED_{16}$ in hexadecimal. This hexadecimal value is composed of the three bytes $0x84_{16}$, $0x5F_{16}$, and $0xED_{16}$. As such the value 8675309_{10} may be represented in Big-Endian as a byte stream of $[0x5C_{16}, 0x7B_{16}, 0x2A_{16}]$.

[1] Wikipedia contributors. (2019, October 10). Endianness. In Wikipedia, The Free Encyclopedia. Retrieved 18:31, October 13, 2019, from https://en.wikipedia.org/w/index.php?title=Endianness&oldid=920566520.
[2] Cohen, Danny the computer scientist that termed the phrase "endian" referring to byte order.
[3] Gulliver's Travels a book.
[4] Swift, Jonathan, author of Gulliver's Travels.

Big-Endian integer representation likely comes as second nature to developers familiar with right-to-left Arabic numerals[5] representation.

1.3 Little-Endian

Little-Endian ordering is right-to-left. Given the representation of an unsigned number in bytes the further to the right a byte exists the more significant it is.

For example, the decimal value of the unsigned integer 8675309_{10} may be represented as $0x845FED_{16}$ in hexadecimal. This hexadecimal value is composed of the three bytes $0x84_{16}$, $0x5F_{16}$, and $0xED_{16}$. But because little-endian byte order is left to right 8675309_{10} may be represented in Little-Endian as a byte stream of $[0xED_{16}, 0x5F_{16}, 0x84_{16}]$.

To developers most affiliated with right-to-left natural languages Little-Endian may seem backwards.

[5] Arabic numerals are the typical digital number format used in the English language.

CHAPTER

TWO

FREQUENTLY ASKED QUESTIONS

Note: No questions have been asked, frequently or otherwise.

CHAPTER

THREE

GENERAL BYTE ARRAY OPERATIONS

There exist a number of operations one may want to do on an array of bytes that don't directly relate to explicit higher order mathematical operations or typical bitwise operations, baring any other name space we're considering these general operations. Typically their about building, transforming, mutating, or gathering meta data.

Unless otherwise stated the following methods are statically defined in the `ByteArrayUtils` class and do not modify their input.

3.1 Byte Array Creation and Population

It is usually easy enough to new up a new byte array, however sometimes something a little more exotic than an array of `0x00` bytes are desired.

3.1.1 Create

It can be necessary to create a byte array filled with a known value. In this case `ByteArrayUtils.CreateByteArray` can be used to create a byte array of a given `length` filled with an optional `element` value.

```
public static byte[] ByteArrayUtils.CreateByteArray(int length, byte element = 0x00)
```

In the following example a byte array of length `10` is filled with the the repeated byte value of `0x42`:

Listing 1: Create Byte Array Example

```
public static void CreateByteArrayExample()
{
   // Setup
   const int length = 10;
   const byte element = 0x42; // optional, defaults to 0x00

   // Act
   // creates a byte array of length 10, filled with bytes of 0x42
   var result = ByteArrayUtils.CreateByteArray(length, element);

  // Conclusion
  Console.WriteLine("CreateByteArray example");
  Console.WriteLine($"length:\t{length}");
  Console.WriteLine($"element:\t{element}");
  Console.WriteLine(\$"result:\t{result.ToString("H")}");
}
```

```
CreateByteArray example
length: 10
element: 66
result: 42 42 42 42 42 42 42 42 42 42
```

Creates a byte array `resultBytes` with a value of `[0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42]`.

3.2 Byte Array Mutation

Byte arrays often need to be altered in some way to process them, the addition of needing to be concerned with endiness can make this a bit less straightforward.

3.2.1 Trimming

Leading zero byte trimming works similarly for both big and little endian byte arrays. In both cases leading, or most significant, zero value bytes are removed. For big endian those bytes starting at the 0th index are removed, whereas for little endian zero bytes are removed from the tail of the array.

If a byte array has no most significant zero valued bytes then a copy of the original will be returned.

Big Endian

To trim all `0x00` bytes starting at the 0th index of the byte array

```
public static byte[] ByteArrayUtils.TrimBigEndianLeadingZeroBytes(this byte[] input)
```

The following example trims the array `[0x00, 0x00, 0x2A, 0x00]` returning `[0x2A, 0x00]`:

Listing 2: Trim Big-Endian Leading Zero Bytes

```
public static void TrimBigEndianLeadingZeroBytes()
{
   // Setup
   var input = new byte[] { 0x00, 0x00, 0x2A, 0x00 };

   // Act
   var result = input.TrimBigEndianLeadingZeroBytes();

   // Conclusion
   Console.WriteLine("TrimBigEndianLeadingZeroBytes example");
   Console.WriteLine($"input:\t{input.ToString("H")}");
   Console.WriteLine($"result:\t{result.ToString("H")}");
}
```

```
TrimBigEndianLeadingZeroBytes example
input: 00 00 2A 00
result: 2A 00
```

Note that the final `0x00` value was not removed as we were only trimming most significant zero values.

Little Endian

To trim all `0x00` bytes starting at the end of the byte array

```
public static byte[] ByteArrayUtils.TrimLittleEndianLeadingZeroBytes(this byte[] input)
```

Listing 3: Trim Little-Endian Leading Zero Bytes

```
public static void TrimLittleEndianLeadingZeroBytes()
{
   // Setup
   var input = new byte[] { 0x2A, 0xFF, 0x2A, 0x00 };

   // Act
   var result = input.TrimLittleEndianLeadingZeroBytes();

   // Conclusion
   Console.WriteLine("TrimLittleEndianLeadingZeroBytes");
   Console.WriteLine($"input:\t{input.ToString("H")}");
   Console.WriteLine($"result:\t{result.ToString("H")}");
}
```

```
TrimLittleEndianLeadingZeroBytes
input: 2A FF 2A 00
result: 2A FF 2A
```

3.2.2 Padding

When padding a byte array, if the the given array length is equal to or larger than `finalLength` a copy of the original array will be returned. Otherwise bytes with the value of `element` will be padded in the most significant value place.

Big Endian

```
public static byte[] ByteArrayUtils.PadBigEndianMostSignificantBytes(this byte[] source,␣
↪int finalLength, byte element = 0x00)
```

Listing 4: Pad Big-Endian Most Significant Bytes Example

```
public static void PadBigEndianMostSignificantBytesExample()
{
   // Setup
   var bytes = new byte[] { 0xDE, 0xFA, 0xCE, 0xC0, 0xDE };
   const int finalLength = 6;

   // Act
   var result = bytes.PadBigEndianMostSignificantBytes(finalLength);

   // Conclusion
   Console.WriteLine("PadBigEndianMostSignificantBytes Short Example");
   Console.WriteLine($"input:\t{bytes.ToString("H")}");
```

(continues on next page)

(continued from previous page)

```
   Console.WriteLine($"result:\t{result.ToString("H")}");
   Console.WriteLine(string.Empty);
}
```

```
PadBigEndianMostSignificantBytes Short Example
input: DE FA CE C0 DE
result: 00 DE FA CE C0 DE
```

Little Endian

```
public static byte[] ByteArrayUtils.PadLittleEndianMostSignificantBytes(this byte[]␣
↪source, int finalLength, byte element = 0x00)
```

Listing 5: Pad Little-Endian Most Significant Bytes Example

```
public static void PadLittleEndianMostSignificantBytesExample()
{
   // Setup
   var input = new byte[] { 0xDE, 0xFA, 0xCE, 0xC0, 0xDE };
   const int finalLength = 6;

   // Act
   var result = input.PadLittleEndianMostSignificantBytes(finalLength);

   // Conclusion
   Console.WriteLine("PadLittleEndianMostSignificantBytes Example");
   Console.WriteLine($"input:\t{input.ToString("H")}");
   Console.WriteLine($"result:\t{result.ToString("H")}");
   Console.WriteLine(string.Empty);
}
```

```
PadLittleEndianMostSignificantBytes Example
input: DE FA CE C0 DE
result: DE FA CE C0 DE 00
```

3.2.3 Appending

Appending operations are endian agnostic, new byte values will appear after the highest order index of the input array.

Append Bytes

The `ByteArrayUtils.AppendBytes` operation simply adds `count` bytes to the end of the value provided by the `source` array. The optional `element` parameter may be provided to use a byte value other than the default `0x00`.

```
public static byte[] ByteArrayUtils.AppendBytes(this byte[] source, int count, byte␣
↪element = 0x00)
```

Listing 6: Append Bytes Example

```
public static void AppendBytesExample()
{
   // Setup
   var input = new byte[] { 0xC0, 0xC0, 0xCA, 0xFE };
   const int count = 4;

   // Act
   var result = input.AppendBytes(count);

   // Conclusion
   Console.WriteLine("AppendBytes Example");
   Console.WriteLine($"input:\t{input.ToString("H")}");
   Console.WriteLine($"result:\t{result.ToString("H")}");
   Console.WriteLine(string.Empty);
}
```

```
AppendBytes Example
input: C0 C0 CA FE
result: C0 C0 CA FE 00 00 00 00
```

Append Shortest

`ByteArrayUtils.AppendShortest` works much like `ByteArrayUtils.AppendBytes`, except instead of providing a desired byte count, the two input arrays lengths are compared and the shortest array is returned, along with the the longest array, with enough `0x00` bytes such that both byte arrays are now the same length.

Effectively this adds most significant `0x00` bytes to the shortest little endian byte array, but may be useful for big endian arrays as well.

```
public static (byte[] left, byte[] right) ByteArrayUtils.AppendShortest(byte[] left,␣
↪byte[] right)
```

Listing 7: Append Shortest Example

```
public static void AppendShortestExample()
{
   // Setup
   var lhs = new byte[] { 0xDE, 0xCA, 0xF0 };
```

(continues on next page)

(continued from previous page)

```
    var rhs = new byte[] { 0xCA, 0xFE, 0xC0, 0xFF, 0xEE };

    // Act
    var (lhsResult, rhsResult) = ByteArrayUtils.AppendShortest(lhs, rhs);

    // Conclusion
    Console.WriteLine("AppendShortest Example");
    Console.WriteLine($"lhs:\t{lhs.ToString("H")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("H")}");
    Console.WriteLine($"lhsResult:\t{lhsResult.ToString("H")}");
    Console.WriteLine($"lhsResult:\t{rhsResult.ToString("H")}");
    Console.WriteLine(string.Empty);
}
```

```
AppendShortest Example
lhs: DE CA F0
rhs: CA FE C0 FF EE
lhsResult: DE CA F0 00 00
lhsResult: CA FE C0 FF EE
```

3.2.4 Prepend

Prepending operations are endian agnostic, new byte values will appear after the lowest order index of the input array.

Prepend Bytes

The `ByteArrayUtils.PrependBytes` operation simply adds `count` bytes to the start of the value provided by the `source` array. The optional `element` parameter may be provided to use a byte value other than the default `0x00`. This is essentially the inverse of `ByteArrayUtils.AppendBytes` operation.

```
public static byte[] ByteArrayUtils.PrependBytes(this byte[] source, int count, byte␣
↪element = 0x00)
```

Listing 8: Prepend Bytes Example

```
public static void PrependBytesExample()
{
    // Setup
    var input = new byte[] { 0xC0, 0xC0, 0xCA, 0xFE };
    const int count = 4;

    // Act
    var result = input.PrependBytes(count);

    // Conclusion
    Console.WriteLine("PrependBytes Example");
    Console.WriteLine($"input:\t{input.ToString("H")}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
}
```

```
PrependBytes Example
input: C0 C0 CA FE
result: 00 00 00 00 C0 C0 CA FE
```

Prepend Shortest

`ByteArrayUtils.PrependShortest` works much like `ByteArrayUtils.PrependBytes`, except instead of providing a desired byte count, the two input arrays lengths are compared and the shortest array is returned, along with the the longest array, with enough `0x00` bytes such that both byte arrays are now the same length.

Effectively this adds most significant `0x00` bytes to the shortest big endian byte array, but may be useful for little endian arrays as well.

```
public static (byte[] left, byte[] right) ByteArrayUtils.PrependShortest(byte[] left,␣
↪byte[] right)
```

Listing 9: Prepend Shortest Example

```
public static void PrependShortestExample()
{
   // Setup
   var lhs = new byte[] { 0xDE, 0xCA, 0xF0 };
   var rhs = new byte[] { 0xCA, 0xFE, 0xC0, 0xFF, 0xEE };

   // Act
   var (lhsResult, rhsResult) = ByteArrayUtils.PrependShortest(lhs, rhs);

   // Conclusion
   Console.WriteLine("PrependShortest Example");
   Console.WriteLine($"lhs:\t{lhs.ToString("H")}");
   Console.WriteLine($"rhs:\t{rhs.ToString("H")}");
   Console.WriteLine($"lhsResult:\t{lhsResult.ToString("H")}");
   Console.WriteLine($"lhsResult:\t{rhsResult.ToString("H")}");
   Console.WriteLine(string.Empty);
}
```

```
PrependShortest Example
lhs: DE CA F0
rhs: CA FE C0 FF EE
lhsResult: 00 00 DE CA F0
lhsResult: CA FE C0 FF EE
```

3.2.5 Reversing

Unsurprisingly, hopefully, `ByteArrayUtils.ReverseBytes` returns the reverse of the provided `bytes` byte array.

Note: The `ReverseBytes` operation is endian agnostic.

```
public static byte[] ByteArrayUtils.ReverseBytes(this byte[] bytes)
```

Listing 10: Reverse Bytes Example

```
public static void ReverseBytesExample()
{
   // Setup
   var input = new byte[] { 0xC0, 0x1D, 0xC0, 0xFF, 0xEE };

   // Act
   var result = input.ReverseBytes();

   // Conclusion
   Console.WriteLine("ReverseBytes example");
   Console.WriteLine($"input:\t{input.ToString("H")}");
   Console.WriteLine($"result:\t{result.ToString("H")}");
   Console.WriteLine(string.Empty);
}
```

```
ReverseBytes example
input: C0 1D C0 FF EE
result: EE FF C0 1D C0
```

3.3 Effective Length

Effective length provides the ability to count the number of non-most significant bytes within a byte array. Eg. the length of meaningful bytes within the array.

3.3.1 Big Endian

`ByteArrayUtils.BigEndianEffectiveLength` returns an `int` representing the byte length of the given `input` disregarding the `0x00` bytes at the beginning of the array.

```
public static int ByteArrayUtils.BigEndianEffectiveLength(this byte[] input)
```

Listing 11: Big-Endian Effective Length Example

```
public static void BigEndianEffectiveLengthExample()
{
   // Setup
   var input = new byte[] { 0x00, 0x00, 0x00, 0xDA, 0xBD, 0xAD };
```

(continues on next page)

(continued from previous page)

```
    // Act
    var result = input.BigEndianEffectiveLength();

    // Conclusion
    Console.WriteLine("BigEndianEffectiveLength Example");
    Console.WriteLine($"input:\t{input.ToString("H")}");
    Console.WriteLine($"result:\t{result}");
    Console.WriteLine(string.Empty);
}
```

```
BigEndianEffectiveLength Example
input: 00 00 00 DA BD AD
result: 3
```

3.3.2 Little Endian

`ByteArrayUtils.LittleEndianEffectiveLength` returns an `int` representing the byte length of the given `input` disregarding the `0x00` bytes at the end of the array.

```
public static int ByteArrayUtils.LittleEndianEffectiveLength(this byte[] input)
```

Listing 12: Little-Endian Effective Length Example

```
public static void LittleEndianEffectiveLengthExample()
{
    // Setup
    var input = new byte[] { 0xDA, 0xB0, 0x00, 0x00, 0x00, 0x00 };

    // Act
    var result = input.LittleEndianEffectiveLength();

    // Conclusion
    Console.WriteLine("LittleEndianEffectiveLength Example");
    Console.WriteLine($"input:\t{input.ToString("H")}");
    Console.WriteLine($"result:\t{result}");
    Console.WriteLine(string.Empty);
}
```

```
LittleEndianEffectiveLength Example
input: DA B0 00 00 00 00
result: 2
```

CHAPTER

FOUR

STRINGIFICATION

What can be better than making byte arrays slightly more human readable!? We provide a number of ways to format a byte array that will hopefully meet your needs. Because `byte[]` doesn't implement `IFormattable` you have to be explicit about calling `ByteArrayUtils.ToString` and cannot rely on string interpolation or `string.Format` and the typical format provider.

```
public static string ByteArrayUtils.ToString(this byte[] bytes, string format = "g",␣
↪IFormatProvider formatProvider = null)
```

Implemented `format` values

Table 1: Implemented format values

Format	Name	Description	Example
`g` (default), `G`, `H`, or empty string	General Format Hexadecimal (base 16) bytes	Formats as uppercase hexadecimal digits with individual bytes delimited by a single space character	`C0 FF EE C0 DE`
`HC`	Uppercase Hexadecimal (base 16) bytes Contiguous	Formats bytes as uppercase contiguous hexadecimal digits	`C0FFEEC0DE`
`h`	Lowercase Hexadecimal (base 16) bytes	Formats bytes as lowercase hexadecimal digits with individual bytes delimited by a single space character	`c0 ff ee c0 de`
`hc`	Lowercase Hexadecimal (base 16) bytes Contiguous	Formats bytes as lowercase contiguous hexadecimal digits	`c0ffeec0de`
`b`, or `B`	Binary (base 2) bytes	Formats bytes as binary digits with individual bytes delimited by a single space character	`11000000 11111111 11101110 11000000 11011110`
`bc`, or `BC`	Binary (base 2) bytes Contiguous	Formats bytes as contiguous binary digits	`11000000111111111110111011000000011`
`d`, or `D`	Decimal (base 10) bytes	Formats bytes as decimal (base 10) digits with individual bytes delimited by a single space character	`192 255 238 192 222`
`I`, or `IBE`	Integer (Big Endian)	Formats bytes as big endian unsigned decimal (base 10) integer	`828927557854`
`ILE`	Integer (Little Endian)	Formats bytes as little endian unsigned decimal (base 10) integer	`956719628224`

Listing 1: Stringification Example

```
public static void StringificationExample()
{
   // Setup
   var input = new byte[] { 0xC0, 0xFF, 0xEE, 0xC0, 0xDE };

   // Conclusion
   Console.WriteLine("Stringification Example");
   Console.WriteLine($"input:\t{input.ToString("H")}");

   Console.WriteLine("Hexadecimal Formats");
   Console.WriteLine($"H:\t\"{input.ToString("H")}\"");
   Console.WriteLine($"h:\t\"{input.ToString("h")}\"");
   Console.WriteLine($"HC:\t\"{input.ToString("HC")}\"");
   Console.WriteLine($"hc:\t\"{input.ToString("hc")}\"");

   Console.WriteLine("Binary Formats");
   Console.WriteLine($"b:\t\"{input.ToString("b")}\"");
   Console.WriteLine($"bc:\t\"{input.ToString("bc")}\"");

   Console.WriteLine("Integer Formats");
   Console.WriteLine($"d:\t\"{input.ToString("d")}\"");
   Console.WriteLine($"IBE:\t\"{input.ToString("IBE")}\"");
   Console.WriteLine($"ILE:\t\"{input.ToString("ILE")}\"");

   Console.WriteLine(string.Empty);
}
```

```
Stringification Example
input:  C0 FF EE C0 DE
Hexadecimal Formats
H:      "C0 FF EE C0 DE"
h:      "c0 ff ee c0 de"
HC:     "C0FFEEC0DE"
hc:     "c0ffeec0de"
Binary Formats
b:      "11000000 11111111 11101110 11000000 11011110"
bc:     "11000000111111111110111011000000011011110"
Integer Formats
d:      "192 255 238 192 222"
IBE:    "828927557854"
ILE:    "956719628224"
```

CHAPTER

FIVE

BITWISE BYTE ARRAY OPERATIONS

The various Bitwise byte array operations provided by Gulliver implement the standard expected bitwise operations that should fit the needs of most developers. In some cases these methods are endian aware such that byte arrays of differing lengths may be appropriately lined up for operations.

Unless otherwise stated the following methods are statically defined in the `ByteArrayUtils` class and do not modify their input.

5.1 Addressing

The various addressing methods allow for the easy retrieval of individual byte, or bit data within the given byte array.

5.1.1 Byte Array as Bit Array

Byte arrays are great, but sometimes what we really need are bit (or `boolean`) arrays instead. `ByteArrayUtils.AddressBit` method takes an array of `byte` treats it as if it were an array of bits instead returning the bit value at the given bit index `index`.

Keep in mind there are 8 bits in a byte.

```
public static bool ByteArrayUtils.AddressBit(this byte[] bytes, int index)
```

Listing 1: AddressBit Example

```
public static class Addressing
{
    public static void AddressBitExample()
    {
        // Setup
        var input = new byte[] { 0xC0, 0x1D };
        var bitLength = input.Length * 8;

        // Act
        IEnumerable<string> result = Enumerable.Range(0, bitLength - 1)
                                               .Select(i =>
                                               {
                                                   var bit = input.AddressBit(i);
                                                   return (i, b: bit ? 1 : 0);
                                               })
                                               .Select(x => $"[{x.i}]:{x.b}")
```

(continues on next page)

(continued from previous page)

```
                                                .Skip(4)
                                                .Take(bitLength - 8)
                                                .ToList();

        // Conclusion
        Console.WriteLine("AddressBit Example");
        Console.WriteLine($"input:\t{input.ToString("H")}");
        Console.WriteLine($"input:\t{input.ToString("b")}");
        Console.WriteLine($"result:\t{string.Join(", ", result)}");
        Console.WriteLine(string.Empty);
    }
}
```

```
AddressBit Example
input:  C0 1D
input:  11000000 00011101
result: [4]:0, [5]:0, [6]:1, [7]:1, [8]:1, [9]:0, [10]:1, [11]:1
```

5.2 Boolean Operations

Boolean operations include the standard **NOT**, **AND**, **OR**, and **XOR**.

XNOR, and the remaining 11 truth functions were deemed unnecessary. But remember, as an exercise for the developer, a **XNOR** may be created by using a **NOT** on the result of an **OR** operation, and given the principal of Functional Completeness each of the 16 truth functions can be built using your newly created gate.

5.2.1 NOT

`ByteArrayUtils.BitwiseNot` will return the inverse of the provided `bytes`

Note: Due to its unary nature the `ByteArrayUtils.BitwiseNot` operation is endian agnostic.

```
public static byte[] ByteArrayUtils.BitwiseNot(byte[] bytes)
```

Listing 2: Bitwise NOT Example

```
public static void BitwiseNotExample()
{
    // Setup
    var input = new byte[] { 0x00, 0x11, 0xAC, 0xFF };
    // Act

    var result = ByteArrayUtils.BitwiseNot(input);

    // Conclusion
    Console.WriteLine("BitwiseNot Example");
    Console.WriteLine($"input:\t{input.ToString("H")}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
```

(continues on next page)

(continued from previous page)

```
    Console.WriteLine(string.Empty);
    Console.WriteLine($"input:\t{input.ToString("b")}");
    Console.WriteLine($"result:\t{result.ToString("b")}");
    Console.WriteLine(string.Empty);
}
```

```
BitwiseNot Example
input:  00 11 AC FF
result: FF EE 53 00

input:  00000000 00010001 10101100 11111111
result: 11111111 11101110 01010011 00000000
```

5.2.2 AND

`ByteArrayUtils.BitwiseAndBigEndian` and `ByteArrayUtils.BitwiseAndLittleEndian` will return the logical AND of the `left` and `right` byte arrays. In the case where the input byte arrays are not of the same length the shortest array will be padded by the appropriate count of `0x00` most significant bytes so that comparisons may appropriately take place.

Big Endian

```
public static byte[] ByteArrayUtils.BitwiseAndBigEndian(byte[] left, byte[] right)
```

Listing 3: Bitwise AND Big-Endian Example

```
public static void BitwiseAndBigEndianExample()
{
    // Setup
    var lhs = new byte[] { 0xC0, 0xDE };
    var rhs = new byte[] { 0xC0, 0xFF, 0xEE };

    // Act
    var result = ByteArrayUtils.BitwiseAndBigEndian(lhs, rhs);

    // Conclusion
    Console.WriteLine("BitwiseAndBigEndian Example");
    Console.WriteLine($"lhs:\t{lhs.ToString("H")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("H")}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"lhs:\t{lhs.ToString("b")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("b")}");
    Console.WriteLine($"result:\t{result.ToString("b")}");
    Console.WriteLine(string.Empty);
}
```

```
BitwiseAndBigEndian Example
lhs:    C0 DE
```

(continues on next page)

(continued from previous page)

```
rhs:    C0 FF EE
result: 00 C0 CE

lhs:    11000000 11011110
rhs:    11000000 11111111 11101110
result: 00000000 11000000 11001110
```

Little Endian

```
public static byte[] ByteArrayUtils.BitwiseAndLittleEndian(byte[] left, byte[] right)
```

Listing 4: Bitwise AND Little-Endian Example

```
public static void BitwiseAndLittleEndianExample()
{
    // Setup
    var lhs = new byte[] { 0xC0, 0xDE };
    var rhs = new byte[] { 0xC0, 0xFF, 0xEE };

    // Act
    var result = ByteArrayUtils.BitwiseAndLittleEndian(lhs, rhs);

    // Conclusion
    Console.WriteLine("BitwiseAndLittleEndian Example");
    Console.WriteLine($"lhs:\t{lhs.ToString("H")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("H")}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"lhs:\t{lhs.ToString("b")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("b")}");
    Console.WriteLine($"result:\t{result.ToString("b")}");
    Console.WriteLine(string.Empty);
}
```

```
BitwiseAndLittleEndian Example
lhs:    C0 DE
rhs:    C0 FF EE
result: C0 DE 00

lhs:    11000000 11011110
rhs:    11000000 11111111 11101110
result: 11000000 11011110 00000000
```

5.2.3 OR

`ByteArrayUtils.BitwiseOrBigEndian` and `ByteArrayUtils.BitwiseOrLittleEndian``will return the logical OR of the ``left` and `right` byte arrays. In the case where the input byte arrays are not of the same length the shortest array will be padded by the appropriate count of `0x00` most significant bytes so that comparisons may appropriately take place.

Big Endian

```
public static byte[] ByteArrayUtils.BitwiseOrBigEndian(byte[] left, byte[] right)
```

Listing 5: Bitwise OR Big-Endian Example

```
public static void BitwiseOrBigEndianExample()
{
    // Setup
    var lhs = new byte[] { 0xC0, 0xDE };
    var rhs = new byte[] { 0xC0, 0xFF, 0xEE };

    // Act
    var result = ByteArrayUtils.BitwiseOrBigEndian(lhs, rhs);

    // Conclusion
    Console.WriteLine("BitwiseOrBigEndian Example");
    Console.WriteLine($"lhs:\t{lhs.ToString("H")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("H")}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"lhs:\t{lhs.ToString("b")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("b")}");
    Console.WriteLine($"result:\t{result.ToString("b")}");
    Console.WriteLine(string.Empty);
}
```

```
BitwiseOrBigEndian Example
lhs:    C0 DE
rhs:    C0 FF EE
result: C0 FF FE

lhs:    11000000 11011110
rhs:    11000000 11111111 11101110
result: 11000000 11111111 11111110
```

Little Endian

```
public static byte[] ByteArrayUtils.BitwiseOrLittleEndian(byte[] left, byte[] right)
```

Listing 6: Bitwise OR Little-Endian Example

```
public static void BitwiseOrLittleEndianExample()
{
    // Setup
    var lhs = new byte[] { 0xC0, 0xDE };
    var rhs = new byte[] { 0xC0, 0xFF, 0xEE };

    // Act
    var result = ByteArrayUtils.BitwiseOrLittleEndian(lhs, rhs);

    // Conclusion
    Console.WriteLine("BitwiseOrLittleEndian Example");
    Console.WriteLine($"lhs:\t{lhs.ToString("H")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("H")}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"lhs:\t{lhs.ToString("b")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("b")}");
    Console.WriteLine($"result:\t{result.ToString("b")}");
    Console.WriteLine(string.Empty);
}
```

```
BitwiseOrLittleEndian Example
lhs:    C0 DE
rhs:    C0 FF EE
result: C0 FF EE

lhs:    11000000 11011110
rhs:    11000000 11111111 11101110
result: 11000000 11111111 11101110
```

5.2.4 XOR

`ByteArrayUtils.BitwiseXorBigEndian` and `ByteArrayUtils.BitwiseXorLittleEndian` will return the logical Exclusive Or of the `left` and `right` byte arrays. In the case where the input byte arrays are not of the same length the shortest array will be padded by the appropriate count of `0x00` most significant bytes so that comparisons may appropriately take place.

Big Endian

```
public static byte[] ByteArrayUtils.BitwiseXorBigEndian(byte[] left, byte[] right)
```

Listing 7: Bitwise XOR Big-Endian Example

```
public static void BitwiseXorBigEndianExample()
{
    // Setup
    var lhs = new byte[] { 0xC0, 0xDE };
    var rhs = new byte[] { 0xC0, 0xFF, 0xEE };

    // Act
    var result = ByteArrayUtils.BitwiseXorBigEndian(lhs, rhs);

    // Conclusion
    Console.WriteLine("BitwiseXorBigEndian Example");
    Console.WriteLine($"lhs:\t{lhs.ToString("H")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("H")}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"lhs:\t{lhs.ToString("b")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("b")}");
    Console.WriteLine($"result:\t{result.ToString("b")}");
    Console.WriteLine(string.Empty);
}
```

```
BitwiseXorBigEndian Example
lhs:    C0 DE
rhs:    C0 FF EE
result: C0 3F 30

lhs:    11000000 11011110
rhs:    11000000 11111111 11101110
result: 11000000 00111111 00110000
```

Little Endian

```
public static byte[] ByteArrayUtils.BitwiseXorLittleEndian(byte[] left, byte[] right)
```

Listing 8: Bitwise XOR Little-Endian Example

```
public static void BitwiseXorLittleEndianExample()
{
    // Setup
    var lhs = new byte[] { 0xC0, 0xDE };
    var rhs = new byte[] { 0xC0, 0xFF, 0xEE };

    // Act
    var result = ByteArrayUtils.BitwiseXorLittleEndian(lhs, rhs);

```

(continues on next page)

(continued from previous page)

```
    // Conclusion
    Console.WriteLine("BitwiseXorLittleEndian Example");
    Console.WriteLine($"lhs:\t{lhs.ToString("H")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("H")}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"lhs:\t{lhs.ToString("b")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("b")}");
    Console.WriteLine($"result:\t{result.ToString("b")}");
    Console.WriteLine(string.Empty);
}
```

```
BitwiseXorLittleEndian Example
lhs:    C0 DE
rhs:    C0 FF EE
result: 00 21 EE

lhs:    11000000 11011110
rhs:    11000000 11111111 11101110
result: 00000000 00100001 11101110
```

5.3 Bitshifting

Bitshifting allows for the shifting of the underlying bit values of bytes in the desired direction.

Bitshifting operations are endian agnostic.

5.3.1 Shift Right

`ByteArrayUtils.ShiftBitsRight` is an arithmetic bit shift that returns the value of `bytes` with its underlying bits shifted `shift` indexes to the right. If the `carry` value is desired there exists an overload, shown below, that outs the result.

```
public static byte[] ByteArrayUtils.ShiftBitsRight(this byte[] bytes, int shift)
```

Listing 9: Shift Bits Right Example

```
public static void ShiftBitsRightExample()
{
    // Setup
    var input = new byte[] { 0xAD, 0x0B, 0xEC, 0x0F, 0xFE, 0xE0 };
    const int shift = 5;

    // Act
    var result = input.ShiftBitsRight(shift);

    // Conclusion
    Console.WriteLine("ShiftBitsRight Example");
    Console.WriteLine($"shift:\t{shift}");
```

(continues on next page)

(continued from previous page)

```
    Console.WriteLine($"input:\t{input.ToString("H")}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"input:\t{input.ToString("b")}");
    Console.WriteLine($"result:\t{result.ToString("b")}");
    Console.WriteLine(string.Empty);
}
```

```
ShiftBitsRight Example
shift:  5
input:  AD 0B EC 0F FE E0
result: 05 68 5F 60 7F F7

input:  10101101 00001011 11101100 00001111 11111110 11100000
result: 00000101 01101000 01011111 01100000 01111111 11110111
```

With Carry

An overload to the above `ByteArrayUtils.ShiftBitsRight` that provides the `carry` result of the operation.

```
public static byte[] ByteArrayUtils.ShiftBitsRight(this byte[] bytes, int shift, out␣
↪byte[] carry)
```

Listing 10: Shift Bits Right Carry Example

```
public static void ShiftBitsRightCarryExample()
{
    // Setup
    var input = new byte[] { 0xAD, 0x0B, 0xEC, 0x0F, 0xFE, 0xE0 };
    const int shift = 5;

    // Act
    var result = input.ShiftBitsRight(shift, out var carry);

    // Conclusion
    Console.WriteLine("ShiftBitsRight Carry Example");
    Console.WriteLine($"input:\t{input.ToString("H")}");
    Console.WriteLine($"shift:\t{shift}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine($"carry:\t{carry.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"input:\t{input.ToString("b")}");
    Console.WriteLine($"result:\t{result.ToString("b")}");
    Console.WriteLine($"carry:\t{carry.ToString("b")}");
    Console.WriteLine(string.Empty);
}
```

```
ShiftBitsRight Carry Example
input:  AD 0B EC 0F FE E0
shift:  5
```

(continues on next page)

(continued from previous page)

```
result: 05 68 5F 60 7F F7
carry:  00

input:  10101101 00001011 11101100 00001111 11111110 11100000
result: 00000101 01101000 01011111 01100000 01111111 11110111
carry:  00000000
```

5.3.2 Shift Left

`ByteArrayUtils.ShiftBitsLeft` is an arithmetic bit shift that returns the value of `bytes` with its underlying bits shifted `shift` indexes to the left. If the `carry` value is desired there exists an overload, shown below, that outs the result.

```
public static byte[] ByteArrayUtils.ShiftBitsLeft(this byte[] bytes, int shift)
```

Listing 11: Shift Bits Left Example

```
public static void ShiftBitsLeftExample()
{
    // Setup
    var input = new byte[] { 0xAD, 0x0B, 0xEC, 0x0F, 0xFE, 0xE0 };
    const int shift = 5;

    // Act
    var result = input.ShiftBitsLeft(shift);

    // Conclusion
    Console.WriteLine("ShiftBitsLeft Example");
    Console.WriteLine($"input:\t{input.ToString("H")}");
    Console.WriteLine($"shift:\t{shift}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"input:\t{input.ToString("b")}");
    Console.WriteLine($"result:\t{result.ToString("b")}");
    Console.WriteLine(string.Empty);
}
```

```
ShiftBitsLeft Example
input:  AD 0B EC 0F FE E0
shift:  5
result: A1 7D 81 FF DC 00

input:  10101101 00001011 11101100 00001111 11111110 11100000
result: 10100001 01111101 10000001 11111111 11011100 00000000
```

With Carry

An overload to the above `ByteArrayUtils.ShiftBitsLeft` that provides the `carry` result of the operation.

```
public static byte[] ByteArrayUtils.ShiftBitsLeft(this byte[] bytes, int shift, out␣
↪byte[] carry)
```

Listing 12: Shift Bits Left Carry Example

```
public static void ShiftBitsLeftCarryExample()
{
    // Setup
    var input = new byte[] { 0xAD, 0x0B, 0xEC, 0x0F, 0xFE, 0xE0 };
    const int shift = 5;

    // Act
    var result = input.ShiftBitsLeft(shift, out var carry);

    // Conclusion
    Console.WriteLine("ShiftBitsLeft Carry Example");
    Console.WriteLine($"input:\t{input.ToString("H")}");
    Console.WriteLine($"shift:\t{shift}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine($"carry:\t{carry.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"input:\t{input.ToString("b")}");
    Console.WriteLine($"result:\t{result.ToString("b")}");
    Console.WriteLine($"carry:\t{carry.ToString("b")}");
    Console.WriteLine(string.Empty);
}
```

```
ShiftBitsLeft Carry Example
input:  AD 0B EC 0F FE E0
shift:  5
result: A1 7D 81 FF DC 00
carry:  15

input:  10101101 00001011 11101100 00001111 11111110 11100000
result: 10100001 01111101 10000001 11111111 11011100 00000000
carry:  00010101
```

CHAPTER

SIX

UNSIGNED ARITHMETIC OPERATIONS

For now arithmetic operations within Gulliver have been limited to those involving the treatment of byte arrays as unsigned integer values. Negative numbers and floating point values are (for the moment) out of scope for the needs of this library.

Note: Unless otherwise stated the following methods are statically defined in the `ByteArrayUtils` class and do not modify their input.

6.1 Addition

Both the `ByteArrayUtils.AddUnsignedBigEndian` and `ByteArrayUtils.AddUnsignedLittleEndian` methods return the addition result of the provided `right` and `left` byte arrays accounting for the appropriate endiness of the input.

6.1.1 Big Endian

```
public static byte[] ByteArrayUtils.AddUnsignedBigEndian(byte[] right, byte[] left)
```

Listing 1: Add Unsigned Big-Endian Example

```
public static void AddUnsignedBigEndianExample()
{
    // Setup
    var lhs = new byte[] { 0xAD, 0xDE, 0xD0 };
    var rhs = new byte[] { 0xC0, 0xDE };

    // Act
    var result = ByteArrayUtils.AddUnsignedBigEndian(lhs, rhs);

    // Conclusion
    Console.WriteLine("AddUnsignedBigEndian Example");
    Console.WriteLine($"lhs:\t{lhs.ToString("H")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("H")}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"lhs:\t{lhs.ToString("IBE")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("IBE")}");
```

(continues on next page)

(continued from previous page)

```
    Console.WriteLine($"result:\t{result.ToString("IBE")}");
    Console.WriteLine(string.Empty);
}
```

```
AddUnsignedBigEndian Example
lhs:    AD DE D0
rhs:    C0 DE
result: AE 9F AE

lhs:    11394768
rhs:    49374
result: 11444142
```

6.1.2 Little Endian

```
public static byte[] ByteArrayUtils.AddUnsignedLittleEndian(byte[] left, byte[] right)
```

Listing 2: Add Unsigned-Little Endian Example

```
public static void AddUnsignedLittleEndianExample()
{
    // Setup
    var lhs = new byte[] { 0xAD, 0xDE, 0xD0 };
    var rhs = new byte[] { 0xC0, 0xDE };

    // Act
    var result = ByteArrayUtils.AddUnsignedLittleEndian(lhs, rhs);

    // Conclusion
    Console.WriteLine("AddUnsignedLittleEndian Example");
    Console.WriteLine($"lhs:\t{lhs.ToString("H")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("H")}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"lhs:\t{lhs.ToString("ILE")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("ILE")}");
    Console.WriteLine($"result:\t{result.ToString("ILE")}");
    Console.WriteLine(string.Empty);
}
```

```
AddUnsignedLittleEndian Example
lhs:    AD DE D0
rhs:    C0 DE
result: 6D BD D1

lhs:    13688493
rhs:    57024
result: 13745517
```

6.2 Subtraction

Both the `ByteArrayUtils.SubtractUnsignedBigEndian` and `ByteArrayUtils.SubtractUnsignedLittleEndian` methods return the subtraction result of the provided `right` (minuend) and `left` (subtrahend) byte arrays accounting for the appropriate endiness of the input.

> **Warning:** If the operation would result in a negative value, given we're only dealing with unsigned integer operations, the execution will throw an `InvalidOperationException`.

6.2.1 Big Endian

```
public static byte[] ByteArrayUtils.SubtractUnsignedBigEndian(byte[] left, byte[] right)
```

Listing 3: Subtract Unsigned Big-Endian Example

```
public static void SubtractUnsignedBigEndianExample()
{
    // Setup
    var lhs = new byte[] { 0xDE, 0x1E, 0x7E, 0xD0 };
    var rhs = new byte[] { 0xC0, 0xDE };

    // Act
    var result = ByteArrayUtils.SubtractUnsignedBigEndian(lhs, rhs);

    // Conclusion
    Console.WriteLine("SubtractUnsignedBigEndian Example");
    Console.WriteLine($"lhs:\t{lhs.ToString("H")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("H")}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"lhs:\t{lhs.ToString("IBE")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("IBE")}");
    Console.WriteLine($"result:\t{result.ToString("IBE")}");
    Console.WriteLine(string.Empty);
}
```

```
SubtractUnsignedBigEndian Example
lhs:    DE 1E 7E D0
rhs:    C0 DE
result: DE 1D BD F2

lhs:    3726540496
rhs:    49374
result: 3726491122
```

6.2.2 Little Endian

```
public static byte[] ByteArrayUtils.SubtractUnsignedLittleEndian(byte[] left, byte[]␣
↪right)
```

Listing 4: Subtract Unsigned Little-Endian Example

```
public static void SubtractUnsignedLittleEndianExample()
{
    // Setup
    var lhs = new byte[] { 0xDE, 0x1E, 0x7E, 0xD0 };
    var rhs = new byte[] { 0xC0, 0xDE };

    // Act
    var result = ByteArrayUtils.SubtractUnsignedLittleEndian(lhs, rhs);

    // Conclusion
    Console.WriteLine("SubtractUnsignedLittleEndian Example");
    Console.WriteLine($"lhs:\t{lhs.ToString("H")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("H")}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"lhs:\t{lhs.ToString("ILE")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("ILE")}");
    Console.WriteLine($"result:\t{result.ToString("ILE")}");
    Console.WriteLine(string.Empty);
}
```

```
SubtractUnsignedLittleEndian Example
lhs:    DE 1E 7E D0
rhs:    C0 DE
result: 1E 40 7D D0

lhs:    3497926366
rhs:    57024
result: 3497869342
```

6.3 Safe Summation

Safe summation allows for the the safe addition or subtraction of a `long` values `delta` from the given `source` byte array input. This is useful for iterating or decrementing byte arrays.

Both the `ByteArrayUtils.TrySumBigEndian` and `ByteArrayUtils.TrySumLittleEndian` methods return a `bool` stating if the operation was successful, and will out a non-null value of `result` on success.

6.3.1 Big Endian

```
public static bool ByteArrayUtils.TrySumBigEndian(byte[] source, long delta, out byte[]␣
↪result)
```

Listing 5: Try Sum Big-Endian Example

```
public static void TrySumBigEndianExample()
{
    // Setup
    var bytes = new byte[] { 0xAD, 0xDE, 0xD0 };
    const long delta = 42L;

    // Act
    var success = ByteArrayUtils.TrySumBigEndian(bytes, delta, out var result);

    // Conclusion
    Console.WriteLine("TrySumBigEndian Example");
    Console.WriteLine($"bytes:\t{bytes.ToString("H")}");
    Console.WriteLine($"delta:\t{delta}");
    Console.WriteLine($"success:\t{success}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"bytes:\t{bytes.ToString("IBE")}");
    Console.WriteLine($"result:\t{result.ToString("IBE")}");
    Console.WriteLine(string.Empty);
}
```

```
TrySumBigEndian Example
bytes:  AD DE D0
delta:  42
success:        True
result: AD DE FA

bytes:  11394768
result: 11394810
```

6.3.2 Little Endian

```
public static bool ByteArrayUtils.TrySumLittleEndian(byte[] source, long delta, out␣
↪byte[] result)
```

Listing 6: Try Sum Little-Endian Example

```
public static void TrySumLittleEndianExample()
{
    // Setup
    var bytes = new byte[] { 0xAD, 0xDE, 0xD0 };
    const long delta = 42L;

    // Act
```

(continues on next page)

(continued from previous page)

```
    var success = ByteArrayUtils.TrySumLittleEndian(bytes, delta, out var result);

    // Conclusion
    Console.WriteLine("TryLittleEndian Subtraction Example");
    Console.WriteLine($"bytes:\t{bytes.ToString("H")}");
    Console.WriteLine($"delta:\t{delta}");
    Console.WriteLine($"success:\t{success}");
    Console.WriteLine($"result:\t{result.ToString("H")}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"bytes:\t{bytes.ToString("ILE")}");
    Console.WriteLine($"result:\t{result.ToString("ILE")}");
    Console.WriteLine(string.Empty);
}
```

```
TrySumLittleEndian Example
bytes:  AD DE D0
delta:  -42
success:        True
result: 83 DE D0

bytes:  13688493
result: 13688451
```

Safe Subtraction

Note: Seemingly conspicuously absent are the `TrySubtractBigEndian` and `TrySubtractLittleEndian` equivalents of the `TrySumBigEndian` and `TrySumLittleEndian` methods. In actuality the various TrySum methods allow for a negative `delta` and therefore are functionally equivalent for safe subtraction.

6.4 Comparison

It is often not enough to simply compare the lengths of two arbitrary byte arrays to determine the equality or the largest / smallest unsigned integer value encoded as bytes.

Both `ByteArrayUtils.CompareUnsignedBigEndian` and `ByteArrayUtils.CompareUnsignedLittleEndian` provide the ability to easily compare byte arrays as their unsigned integer values.

The result is similar to that of `IComparer.Compare(left, right)`. The signed integer indicates the relative values of `left` and `right`:

- If 0, `left` equals `right`
- If less than 0, `left` is less than 'right'
- If greater than 0, `right` is greater than `left`

6.4.1 Big Endian

```
public static int ByteArrayUtils.CompareUnsignedBigEndian(byte[] left, byte[] right)
```

Listing 7: Compare Unsigned Big-Endian Example

```
public static void CompareUnsignedBigEndianExample()
{
    // Setup
    var lhs = new byte[] { 0xB1, 0x66, 0xE5, 0x70 };
    var rhs = new byte[] { 0x5A, 0x11 };

    // Act
    var result = ByteArrayUtils.CompareUnsignedBigEndian(lhs, rhs);

    // Conclusion
    Console.WriteLine("CompareUnsignedBigEndian Example");
    Console.WriteLine($"lhs:\t{lhs.ToString("H")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("H")}");
    Console.WriteLine($"result:\t{result}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"lhs:\t{lhs.ToString("IBE")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("IBE")}");
    Console.WriteLine($"result:\t{result}");
    Console.WriteLine(string.Empty);
}
```

```
lhs:    B1 66 E5 70
rhs:    5A 11
result: 1

lhs:    2976310640
rhs:    23057
result: 1
```

6.4.2 Little Endian

```
public static int ByteArrayUtils.CompareUnsignedLittleEndian(byte[] left, byte[] right)
```

Listing 8: Compare Unsigned Little-Endian Example

```
public static void CompareUnsignedLittleEndianExample()
{
    // Setup
    var lhs = new byte[] { 0xB1, 0x66, 0xE5, 0x70 };
    var rhs = new byte[] { 0x5A, 0x11 };

    // Act
    var result = ByteArrayUtils.CompareUnsignedLittleEndian(lhs, rhs);

    // Conclusion
```

(continues on next page)

(continued from previous page)

```
    Console.WriteLine("CompareUnsignedLittleEndian Example");
    Console.WriteLine($"lhs:\t{lhs.ToString("H")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("H")}");
    Console.WriteLine($"result:\t{result}");
    Console.WriteLine(string.Empty);
    Console.WriteLine($"lhs:\t{lhs.ToString("ILE")}");
    Console.WriteLine($"rhs:\t{rhs.ToString("ILE")}");
    Console.WriteLine($"result:\t{result}");
    Console.WriteLine(string.Empty);
}
```

```
CompareUnsignedLittleEndian Example
lhs:    B1 66 E5 70
rhs:    5A 11
result: 1

lhs:    1894082225
rhs:    4442
result: 1
```

CHAPTER

SEVEN

ENDIAN BYTE ENUMERABLES

The `LittleEndianByteEnumerable` and `BigEndianByteEnumerable` gives access to more cleanly treat little-endian and big-endian byte arrays as enumerables in an expected indexable manner regardless of the underlying endianness ignoring `0x00` valued most significant bytes and managing indexing of the most significant byte at the 0th index.

Warning: This topic will be further expounded upon at a later date. In the meantime please feel free to browse the source code available:

- GitHub/Gulliver
  - IByteEnumerable
    - AbstractByteEnumerable
      - BigEndianByteEnumerable
      - LittleEndianByteEnumerable

CHAPTER

EIGHT

CONCURRENT ENDIAN BYTE ENUMERABLES

`ConcurrentBigEndianByteEnumerable` and `ConcurrentLittleEndianByteEnumerable` allows for ease in parallel indexing a pair of byte arrays, that may not be of the same length, in the desired endianness. This comes in particularly useful when running bitwise or mathematical operations.

Warning: This topic will be further expounded upon at a later date. In the meantime please feel free to browse the source code available:

- GitHub/Gulliver
  - IConcurrentByteEnumerable
    - AbstractConcurrentByteEnumerable
      - ConcurrentBigEndianByteEnumerable
      - ConcurrentLittleEndianByteEnumerable

CHAPTER

NINE

FIXEDBYTES

The `FixedBytes` class brings many of these operations together allowing developers to treat a `byte[]` as a more complex object without the need to explicitly call helper or extension methods. It acts as a wrapper around an array of bytes in BigEndian byte order.

9.1 Implements

- `IFormattable`
- `IReadOnlyCollection<byte>`
- `IEquatable<FixedBytes>`
- `IEquatable<IEnumerable<byte>>`
- `IComparable<FixedBytes>`
- `IComparable<IEnumerable<byte>>`
- `IComparable`

9.2 Operators

Note: Operators, where pertinent, treat `FixedBytes` as unsigned big-endian integers

9.2.1 Bitwise

- `|` - OR
- `&` - AND
- `^` - XOR
- `!` - NOT
- `<<` - Shift Left
- `>>` - Shift Right

9.2.2 Mathematical

- + - Addition
- - - Subtraction

9.2.3 Comparison

- > - Greater Than
- < - Less Than
- >= - Greater Than or Equal
- <= - Less Than or Equal
- == - Equals
- != - Not Equals

9.2.4 Explicit Conversion (cast from)

- `byte[]`
- `List<byte>`

9.2.5 Implicit Conversion (cast to)

- `byte[]`

Warning: This topic will be further expounded upon at a later date. In the mean time please feel free to browse the source code available

- GitHub/Gulliver
  - FixedBytes

CHAPTER

TEN

COMMUNITY

10.1 GitHub

10.2 File an Issue

Issues should be filed on the Gulliver GitHub Issue Tracker.

CHAPTER

ELEVEN

HOW TO BUILD GULLIVER

Note: Make sure you clone the latest version of Gulliver's master branch from Github https://github.com/sandialabs/Gulliver.git.

11.1 Building with psake

Gulliver uses psake, *a build automation tool written in PowerShell*, to aid in the build, but it isn't explicitly necessary. To use psake follow the "How to get started" guide on their site, adding it to your path.

If you'd rather not use psake jump ahead to see the manual way of doing things.

11.1.1 psake tasks

The psake tasks are defined in the `pasake.ps1` PowerShell script in the project root directory.

Note: If you're using Visual Studio Code, and if you're not you should be, the psake tasks are available in the Task Explorer. The tasks are referenced in the `.vscode\tasks.json` in the root directly of Gulliver.

Warning: Before attempting to run any of the psake tasks make sure you have the appropriate prerequisites in place.

The tasks of most concern are as follows:

`clean`
: Cleans the C# portion of the project by removing the various `obj` and `bin` folders.

`build_src`
: Cleans and builds the C# source.

`test`
: Runs the Gulliver unit tests.

`pack_debug`
: Create a `Gulliver.nupkg` and `Gulliver.snupkg` debug packages.

`build_docs`
: Builds the Sphinx Documentation as HTML.

11.2 C# Code

11.2.1 Setup Your C# Environment

Gulliver is built with C#, if you want to build Gulliver you'll want to set yourself up to build C# code.

1. Install the .NET Core SDK appropriate for your environment.
2. Consider installing Visual Studio and/or Visual Studio Code.
3. Everything you want to do here on out can be done with dotnet command line tool.
    - To build

      ```
      dotnet build path\to\gulliver\src\Gulliver.sln
      ```

    - To test

      ```
      dotnet test path\to\gulliver\src\Gulliver.Tests\Gulliver.Tests.csproj
      ```

    - To package

      ```
      dotnet pack path\to\gulliver\src\Gulliver\Gulliver.csproj
      ```

11.3 Build The Documentation

11.3.1 First Things First

Note: Before you can run, first you must walk. Likewise, before you can build docs fist you must do some Python stuff.

The documentation relies on Sphinx and Python and a number of Python packages.

1. Install Python 3.7+
2. Install Sphinx 2.2.0+ via the Python package manager *pip*

   ```
   pip install sphinx
   ```

3. Install the Sphinx RTD Theme via *pip*

   ```
   pip install sphinx_rtd_theme
   ```

11.3.2 Build

Once you have all the perquisites in place building the documentation, as HTML[1], is as simple as locating the `make.bat` in the Gulliver `docs` folder. Then simply execute

```
path\to\gulliver\docs\make.bat html
```

Once complete the documentation will be present in the `_build` sub folder of `docs`.

footnotes

[1] You don't have to stick with HTML builds, Sphinx provides other artifacts types as well, take a look at their Invocation of sphinx-build for other options.

CHAPTER

TWELVE

ACKNOWLEDGEMENTS

12.1 Citations

12.1.1 Wikipedia on Endianness

Wikipedia contributors. (2019, October 10). Endianness. In Wikipedia, The Free Encyclopedia. Retrieved 18:31, October 13, 2019, from https://en.wikipedia.org/w/index.php?title=Endianness&oldid=920566520.

12.1.2 "Simply Explained" comic

"Simply Explained" comic

Widder, O. (2011, September 7). Simply Explained. Retrieved October 13, 2019, from <http://geek-and-poke.com/geekandpoke/2011/9/7/simply-explained.html.

12.1.3 Gulliver logo

Title page of first edition of Gulliver's Travels by Jonathan Swift.

This work is in the public domain in its country of origin and other countries and areas where the copyright term is the author's life plus 70 years or fewer.

U.S. work public domain in the U.S. for unspecified reason but presumably because it was published in the U.S. before 1924. This work has been identified as being free of known restrictions under copyright law, including all related and neighboring rights.