Arcus

Sandia National Laboratories

Nov 11, 2019

IP ADDRESSES RANGES

Arcus is a C# manipulation library for calculating, parsing, formatting, converting, and comparing both IPv4 and IPv6 addresses and subnets. It accounts for 128-bit numbers on 32-bit platforms.

Arcus provides extension and helper methods for the pre-existing `System.Net.IPAddress` and other objects within that realm. It was created to fill in some of the gaps left by the absence of a representation of a *Subnet*. As more gaps were found, they were filled. Like all coding projects, Arcus is a work in progress. We rely on both our free time and our *community* in order to provide the best solution we can given the constraints we must conform to.

Hint: Chances are you're primarily here looking for the *Subnet* object.

Arcus heavily relies upon one of our other libraries Gulliver, if you're interested in byte manipulation it is worth checking out.

CHAPTER

ONE

IIPADDRESSRANGE

Arcus defines the `IIPAddressRange` interface for representation of consecutive `IPAddress` objects. It implements both `IFormattable` and `IEnumerable<IPAddress>`.

Caution: `IIPAddressRange` implements `IEnumerable<IPAddress>`, this means that you should pay particular attention when you may be iterating over large ranges. Such as the full set of IPv6 addresses, which will take a while. A long while. It isn't recommended.

Hint: When dealing with more than one `IPAddress` or multiple implementations of `IIPAddressRange` unless otherwise explicitly stated their `AddressFamily`, or equivalent properties, **must** match.

Hint: `AddressFamily` unless otherwise explicitly stated are expected to be either `InterNetwork` or `InterNetworkV6`.

`IIPAddressRange` is implemented by *AbstractIPAddressRange*, *IPAddress Range*, and *Subnet*.

1.1 Functionality Promises

1.1.1 Properties

`IIPAddressRange` has a handful of useful properties for your use

`AddressFamily` AddressFamily The family of the Address Range. You'll most likely encounter `InterNetwork` or `InterNetworkV6`

`IPAddress` Head The first `IPAddress` within the range

`bool` IsIPv4 Returns `true` if, and only if, the range is IPv4

`bool` IsIPv6 Returns `true` if, and only if, the range is IPv6

`bool` IsSingleIP Returns `true` if, and only if, the range is comprised of only a single `IPAddress`

`BigInteger` Length The number of `IPAddress` within the range

`IPAddress` Tail The last `IPAddress` within the range

1.1.2 Set Based Operations

At its core an implementation of the `IIPAddressRange` interface is a range of consecutive `IPAddress` objects, as such there are some set based operations available.

HeadOverlappedBy

`HeadOverlappedBy` will return `true` if the `head` of `this` is within the range defined by `IIPAddressRange that`.

```
bool HeadOverlappedBy(IIPAddressRange that);
```

TailOverlappedBy

`TailOverlappedBy` will return `true` if the `tail` of `this` is within the range defined by `IIPAddressRange that`.

```
bool TailOverlappedBy(IIPAddressRange that);
```

Overlaps

`Overlaps` will return `true` if the `head` or `tail` of `IIPAddressRange that` is within the `this IIPAddressRange`.

```
bool Overlaps(IIPAddressRange that);
```

Touches

`Touches` will return `true` if the `tail` of `this IIPAddressRange` is followed consecutively by the `head` of `IIPAddressRange that`, or if the `tail` of `IIPAddressRange that` is followed consecutively by the `head` of `this IIPAddressRange` without any additional `IPAddress` objects in between.

```
bool Touches(IIPAddressRange that);
```

1.1.3 Length and TryGetLength

The `IIPAddressRange` implements `IEnumerable<IPAddress>`, but because of the possible size of this range it may not always be safe to attempt to do a count or get the length in a traditional manner. A `BigInteger Length` property is provided but not always ideal but often necessary. Keep in mind the full range of IPv6 Addresses is 2^{128} in length. That's 3.4×10^{38} or over 340 undecillion. Certainly not something that should be iterated in order to be counted.

Given that the `BigInteger` object isn't the best thing to drag around Arcus uses the *magic* of math and with the various implementations of `TryGetLength` to get the length of the range in a more portable manner if possible, returning `true` on success and outing the more reasonable `int` or `long` length.

```
bool TryGetLength(out int length);
```

```
bool TryGetLength(out long length);
```

CHAPTER

TWO

ABSTRACTIPADDRESSRANGE

The `AbstractIPAddressRange` is an abstract implementation of *IIPAddressRange*. It is extended by both *IPAddress Range*, and *Subnet*.

2.1 Functionality Implementation

2.1.1 IFormatable

Extensions of `AbstractIPAddressRange`, depending on overrides and implementation, provide a general format (`G`, `g`, or empty string) that will express a range of IP addresses in a `head - tail` format for example `192.168.1.1 - 192.168.1.10`.

Listing 1: AbstractIPAddressRange IFormattable Example

```
[Fact]
public void IFormattable_Example()
{
    // Arrange
    var head = IPAddress.Parse("192.168.0.0");
    var tail = IPAddress.Parse("192.168.128.0");
    var ipAddressRange = new IPAddressRange(head, tail);

    const string expected = "192.168.0.0 - 192.168.128.0";

    // Act
    var formattableString = string.Format("{0:g}", ipAddressRange);

    // Assert
    Assert.Equal(expected, formattableString);
}
```

CHAPTER

THREE

IPADDRESS RANGE

`IPAddressRange` is a very basic implementation of an *AbstractIPAddressRange* used to represent an inclusive range of arbitrary IP Addresses of the same address family. It isn't restricted to a CIDR representation like a *Subnet* is, allowing for non-power of two range sizes.

The `IPAddressRange` class extends *AbstractIPAddressRange* and implements *IIPAddressRange*, `IEquatable<IPAddressRange>`, `IComparable<IPAddressRange>`, `IFormattable`, `IEnumerable<IPAddress>`, and `ISerializable`.

3.1 Creation

3.1.1 constructor `IPAddress head, IPAddress tail`

To standard way of creating an `IPAddressRange` is to construct it via a `IPAddress head` and `IPAddress tail`. This will construct an `IPAddressRange` that would inclusively start with the provided `head` and end with `tail`.

Addresses *MUST* be the same address family (either `InterNetwork` or `InterNetworkV6`).

```
public IPAddressRange(IPAddress head, IPAddress tail)
```

3.1.2 constructor `IPAddress address`

On the rare occasion it may be desirable to make a `IPAddressRange` comprised of a single `IPAddress`. This too is possible with the following constructor.

```
public IPAddressRange(IPAddress address)
```

3.2 Static Functionality

3.2.1 TryCollapseAll

`TryCollapseAll` attempts to or collapse the given input of `IEnumerable<IPAddressRange> ranges` into as few ranges as possible thus minifying the number or ranges supporting the same data.

Ranges may be collapsed if, and only if, they either overlap, or touch each other and they share the same `AddressFamily`.

The function call will return `true` if it could collapse two or more ranges. Regardless of if a collapse was possible the *out* value for `result` will be comprised of an `IEnumerable<IPAddressRange>` of the calculated ranges.

```
public static bool TryCollapseAll(IEnumerable<IPAddressRange> ranges, out IEnumerable
↪<IPAddressRange> result)
```

The following example shows that the three touching ranges of `192.168.1.0 - 192.168.1.5`, `192.168.1.6 - 192.168.1.7`, and `192.168.1.8 - 192.168.1.20` were collapsed into the new `IPAddressRange` of `192.168.1.0 - 192.168.1.20`.

Listing 1: IPAddressRange TryCollapseAll Example

```
[Fact]
public void TryCollapseAll_Consecutive_Example()
{
    // Arrange
    var ranges = new[]
                    {
                        new IPAddressRange(IPAddress.Parse("192.168.1.0"), IPAddress.
↪Parse("192.168.1.5")),
                        new IPAddressRange(IPAddress.Parse("192.168.1.6"), IPAddress.
↪Parse("192.168.1.7")),
                        new IPAddressRange(IPAddress.Parse("192.168.1.8"), IPAddress.
↪Parse("192.168.1.20"))
                    };

    // Act
    var success = IPAddressRange.TryCollapseAll(ranges, out var results);
    var resultList = results?.ToList();

    // Assert
    Assert.True(success);
    Assert.NotNull(results);
    Assert.Single(resultList);

    var result = resultList.Single();

    Assert.Equal(IPAddress.Parse("192.168.1.0"), result.Head);
    Assert.Equal(IPAddress.Parse("192.168.1.20"), result.Tail);
}
```

3.2.2 TryExcludeAll

`TryExcludeAll` is a tricky beast, but if you're willing to take the time to tame it'll not only respect you, but it may also take care of you in very specific cases. The method takes a `IPAddressRange initialRange` and with that it attempts to systematically remove each of the sub ranges defined within `IEnumerable<IPAddressRange> excludedRanges`. On success, the operation returns `true` and will *out* an `IEnumerable<IPAddressRange> result` which is comprised of a distinct remaining ranges after `excludedRanges` have been carved out.

```
public static bool TryExcludeAll(IPAddressRange initialRange, IEnumerable
↪<IPAddressRange> excludedRanges, out IEnumerable<IPAddressRange> result)
```

3.2.3 TryMerge

`TryMerge` will take the input of `IPAddressRange left` and `IPAddressRange right`, and if the two ranges touch or overlap, regardless of order, it will return `true` and *out* `IPAddressRange mergedRange` comprised of the now combined ranges sourcing its `head` from the lowest valued address of the two inputs and its `tail` from the highest valued address of the two.

```
public static bool TryMerge(IPAddressRange left, IPAddressRange right, out␣
↪IPAddressRange mergedRange)
```

CHAPTER

FOUR

SUBNET

The `Subnet` type, flavored in both IPv4 or IPv6, is a representation of a subnetwork within Arcus. It is the workhorse and original reason for the Arcus library. Outside the concept of the `Subnet` object, most everything else in Arcus is auxiliary and exists only in support of making this one facet work. That's not to say that the remaining pieces of the Arcus library aren't useful, on the contrary their utility can benefit a developer greatly. But that said, once the dark and mysterious magic of the `Subnet` is understood the rest of Arcus should be easy to understand.

Keep in mind that a `Subnet` is not an arbitrary range of addresses, for that you want an *IPAddress Range*, but rather conforms to a range of length 2^n starting a particular position, following the typical rules of Classless Inter-Domain Routing.

The `Subnet` class extends *AbstractIPAddressRange* and implements *IIPAddressRange*, `IEquatable<Subnet>`, `IComparable<Subnet>`, `IFormattable`, `IEnumerable<IPAddress>`, and `ISerializable`.

Note: Be aware that `Subnet` does *not* extend *IPAddress Range* but does implement *IIPAddressRange*.

4.1 Creation

There are a number of ways to instantiate a `Subnet`. Your most likely candidates are direct construction with a `new`, the use of a static factory method on the `Subnet` class, or the use of sub-set of static factory methods that handle parsing of strings. Most of the factory methods have a "try" style safe alternative that will return a `bool` and *out* the constructed value.

Note: Unless otherwise specified each creation technique is valid for both IPv4 and IPv6 subnetworks.

4.1.1 constructor `IPAddress lowAddress, IPAddress highAddress`

The most common way to create a `Subnet` is to construct it via a `IPAddress lowAddress` and `IPAddress highAddress`. This will construct the smallest possible `Subnet` that would contain both IP addresses. Typically, the address specified are the Network and Broadcast addresses (lower and higher bounds) but this is not necessary.

Addresses *MUST* be the same address family (either `InterNetwork` or `InterNetworkV6`).

```
public Subnet(IPAddress lowAddress, IPAddress highAddress)
```

4.1.2 constructor `IPAddress address, int routingPrefix`

It is also possible to create a `Subnet` from an `IPAddress address` and an `int routingPrefix`. This is equivalent of programmatically using a CIDR to define your `Subnet`.

```
public Subnet(IPAddress address, int routingPrefix)
```

The following example shows that the `IPAddress` and `routingPrefix` constructor taking an input of `192.168.1.1` and `24` creates a `Subnet` `192.168.1.0/32`. Note that the `Head` is `192.168.1.0` and not `192.168.1.1`, this is done as Arcus will autocorrect the input to a valid `Subnet`. If this is not desired it is advised that you compare the `Head` to the input in order to validate expectations.

Listing 1: Subnet Address and Route Prefix Constructor Example

```
[Fact]
public void Address_RoutePrefix_Subnet_Example()
{
    // Arrange
    var ipAddress = IPAddress.Parse("192.168.1.1");
    const int routePrefix = 24;

    // Act
    var subnet = new Subnet(ipAddress, routePrefix);

    // Assert
    Assert.False(subnet.IsSingleIP);
    Assert.Equal(256, subnet.Length);
    Assert.Equal("192.168.1.0", subnet.Head.ToString());
    Assert.Equal("192.168.1.255", subnet.Tail.ToString());
    Assert.Equal(24, subnet.RoutingPrefix);
    Assert.Equal("192.168.1.0/24", subnet.ToString());
}
```

4.1.3 constructor `IPAddress address`

On the rare occasion it may be desired to make a `Subnet` comprised of a single `IPAddress`. This is possible with the following constructor.

```
public Subnet(IPAddress address)
```

The following example shows that the single `IPAddress` constructor taking an input of `192.168.1.1` creates a `Subnet` `192.168.1.1/32` that is comprised of only the single input address.

Listing 2: Subnet Single Address Constructor Example

```
[Fact]
public void Single_Address_Subnet_Example()
{
   // Arrange
   var ipAddress = IPAddress.Parse("192.168.1.1");

   // Act
   var subnet = new Subnet(ipAddress);

   // Assert
```

(continues on next page)

(continued from previous page)

```
    Assert.Equal(1, subnet.Length);
    Assert.Equal(ipAddress, subnet.Single());
    Assert.True(subnet.IsSingleIP);
    Assert.Equal("192.168.1.1/32", subnet.ToString());
}
```

4.1.4 factory IPAddress and NetMask

A once popular way to define a IPv4 subnetwork was to use a *netmask*, a specialized form of consecutive *bitmasking*, along side an `IPAddress`.

The following factory methods may be used to create an IPv4 `Subnet` where as the `IPAddress address` is the address, and the `IPAddress netmask` is the valid *netmask*.

```
public static Subnet FromNetMask(IPAddress address, IPAddress netmask)
```

```
public static bool TryFromNetMask(IPAddress address, IPAddress netmask, out Subnet␣
↪subnet)
```

4.1.5 factory From Big-Endian Byte Arrays

`IPAddress` objects may not always be handy, in some cases only a couple of big-endian byte arrays may be available. This will construct the smallest possible `Subnet` that would contain both byte arrays as IP addresses. Typically, the address specified are the Network and Broadcast addresses (lower and upper bounds) but this is not necessary.

The given `byte` arrays are interpreted as being in big-endian ordering are are functionally the equivalent construction an `IPAddress` using its `byte[]` constructor.

```
public static Subnet FromBytes(byte[] lowAddressBytes, byte[] highAddressBytes)
```

```
public static bool TryFromBytes(byte[] lowAddressBytes, byte[] highAddressBytes, out␣
↪Subnet subnet)
```

4.1.6 parse string

It is pretty common to tote around a `string` as a representation of a subnet, but you needn't do such any longer. Assuming said `string subnetString` represents something roughly similar to a CIDR Arcus will hand you a `Subnet`.

If a representation of an IP Address `string` is provided the resulting `Subnet` will consist of only that address.

```
public static Subnet Parse(string subnetString)
```

```
public static bool TryParse(string subnetString, out Subnet subnet)
```

4.1.7 parse IPAddress string and RoutingPrefix int

It is also possible to build a `Subnet` from an `String address` and an `int routingPrefix`.

```
public static Subnet Parse(string addressString, int routingPrefix)
```

```
public static bool TryParse(string addressString, int routingPrefix, out Subnet␣
↪subnet)
```

4.1.8 parse IPAddress strings

A rather common way to to build a `Subnet` is to provide a pair of `string` objects, in this case a `string lowAddress` and `string highAddress`. This will construct the smallest possible `Subnet` that would contain both IP addresses. Typically, the address specified are the Network and Broadcast addresses (lower and higher bounds) but this is not necessary.

```
public static Subnet Parse(string lowAddressString, string highAddressString)
```

```
public static bool TryParse(string lowAddressString, string highAddressString, out␣
↪Subnet subnet)
```

4.2 Functionality

The `Subnet` implements *IIPAddressRange*, `IEquatable<Subnet>`, `IComparable<Subnet>`, `IFormattable`, and `IEnumerable<IPAddress>`, and there by contains all the expected functionality it inherits.

4.2.1 Properties

In addition to the properties defined in *IIPAddressRange* `Subnet` provides a few more additional options

`IPAddress` BroadcastAddress An alias to the `Tail` property

`IPAddress` Netmask The calculated netmask of the subnet, only valid for IPv4 based subnets. All others will be return a `null` value

`IPAddress` NetworkPrefixAddress An alias to the `Head` property

`int` RoutingPrefix The routing prefix used to specify the subnet

`BigInteger` UsableHostAddressCount The number of usable addresses in the subnet ignoring both the Broadcast and Network addresses

4.2.2 Set Based Operations

Inherently a `Subnet` is a range of `IPAddress` objects, as such there is some set based operations available.

In addition to the set based operations promised by *IIPAddressRange*, the `Subnet` type also has a few new options.

Contains

It is possible to easily check if a subnet is entirely encapsulates another subnet by using the `Contains` method on the larger `Subnet`.

```
public bool Subnet.Contains(Subnet subnet)
```

In the following example it is shown that `192.168.1.0/8` contains `192.168.0.0`, but as expected `192.168.1.0/8` does not contain `255.0.0.0/8`

Listing 3: Subnet Contains Example

```
[Fact]
public void Contains_Example()
{
    // Arrange
    var subnetA = Subnet.Parse("192.168.1.0", 8);    // 192.0.0.0 - 192.255.255.255
    var subnetB = Subnet.Parse("192.168.0.0", 16);   // 192.168.0.0 - 192.168.255.255
    var subnetC = Subnet.Parse("255.0.0.0", 8);      // 255.0.0.0 - 255.255.255.255

    // Assert
    Assert.True(subnetA.Contains(subnetB));
    Assert.False(subnetA.Contains(subnetC));
}
```

Overlaps

It is possible determine if a subnet in any way overlaps another subnet, even if just by a single address, by using the `Contains` between two subnets.

This is a transitive operation, so if `Subnet A` overlaps `Subnet B` then B overlaps A as well.

```
public bool Overlaps(Subnet subnet)
```

In the following example it is shown that `255.255.0.0/16` and `0.0.0.0/0` each overlap each other. However, due to their disparate address families, `::/0` and `0.0.0.0/0` do not overlap despite being equivalent ranges in the differing in integer spaces.

Listing 4: Subnet Overlaps Example

```
[Fact]
public void Overlaps_Example()
{
   // Arrange
   var ipv4SubnetA = Subnet.Parse("255.255.0.0", 16);
   var ipv4SubnetB = Subnet.Parse("0.0.0.0", 0);

   var ipv6SubnetA = Subnet.Parse("::", 0);
   var ipv6SubnetB = Subnet.Parse("abcd:ef01::", 64);

   // Act
   Assert.True(ipv4SubnetA.Overlaps(ipv4SubnetB));
   Assert.True(ipv4SubnetB.Overlaps(ipv4SubnetA));
   Assert.True(ipv6SubnetA.Overlaps(ipv6SubnetB));
   Assert.False(ipv6SubnetA.Overlaps(ipv4SubnetA));
}
```

4.2.3 IFormatable

`Subnet` offers a number or preexisting formats that are accessible via the standard `ToString` method provided by `IFormattable`

Table 1: Subnet format values

Format	Name	Description	Example
null, empty string, `g`, `G`	Default / General format	CIDR representation	`255.255.0.0/16`
`f`, `F`	"friendly" format	CIDR representaion for Subnets of size > 1 Single address representation for Subnetes of size 1	`255.255.0.0/16` or `192.168.1.1`
`r`, `R`	range format	A range represented by NetworkPrefix - Brodcast	`ab::3d00 - ab::3dff`

CHAPTER

FIVE

SUBNET UTILITIES

`Arcus.Utilities.SubnetUtilities` is a static utility class containing miscellaneous operations for *Subnet* and collections there of. It is a catchall for methods and functionally that didn't make sense on the `Subnet` class itself.

5.1 find Fewest Consecutive Subnets

Given an inclusive range of IP Addresses defined by `IPAddress left` and `IPAddress right` get the fewest consecutive subnets that would contain all addresses within the range between and no other addresses.

```
public static IEnumerable<Subnet> FewestConsecutiveSubnetsFor(IPAddress left,␣
↪IPAddress right)
```

The following examples shows that the range defined by `192.168.1.3 - 192.168.1.5` fits in two consecutive subnets defined by `192.168.1.4/31` and `192.168.1.3/32`.

Listing 1: FewestConsecutiveSubnetsFor Example

```
[Fact]
public void FewestConsecutiveSubnetsFor_Example()
{
    // Arrange
    var left = IPAddress.Parse("192.168.1.3");
    var right = IPAddress.Parse("192.168.1.5");

    // Act
    var result = SubnetUtilities.FewestConsecutiveSubnetsFor(left, right);

    // Assert
    Assert.Equal(2, result.Length);
    Assert.Contains(Subnet.Parse("192.168.1.4/31"), result);
    Assert.Contains(Subnet.Parse("192.168.1.3/32"), result);
}
```

5.2 find the Largest Subnet in an enumerable

The `LargestSubnet` method, given an ``IEnumerable<Subnet>` will select first largest subnet from within the collection.

Note: If there is no single largest in the input, the first largest subnet encountered will be returned. In cases such as this it may be preferable to consider usage of the `DefaultSubnetComparer`.

```
public static Subnet LargestSubnet(IEnumerable<Subnet> subnets)
```

The following example provides that given the several oddly named sizes of subnets that *trenta*, composed of 1048576 addresses, is both largest and probably more caffeine than your originally anticipated.

Listing 2: LargestSubnet Example

```
[Fact]
public void LargestSubnet_Example()
{
    // Arrange
    var tall = Subnet.Parse("255.255.255.254/31");  // 2^1 = 2
    var grande = Subnet.Parse("192.168.1.0/24");    // 2^8 = 256
    var vente = Subnet.Parse("10.10.0.0/16");       // 2^16 = 65536
    var trenta = Subnet.Parse("16.240.0.0/12");     // 2^20 = 1048576

    var subnets = new[] { tall, grande, vente, trenta };

    // Act
    var result = SubnetUtilities.LargestSubnet(subnets);

    // Assert
    Assert.Equal(trenta, result);
}
```

5.3 find the Smallest Subnet in an enumerable

The `SmallestSubnet` method, given an `IEnumerable<Subnet>` will select the first smallest subnet from within the collection.

Note: If there is no single smallest in the input, the first smallest subnet encountered will be returned. In cases such as this it may be preferable to consider usage of the `DefaultSubnetComparer`.

```
public static Subnet SmallestSubnet(IEnumerable<Subnet> subnets)
```

The included example shows that given the several seemingly familiar named subnets that *tall*, composed of 2 addresses, is not only the smallest, but likely will cost you a few bucks and taste a bit burnt.

Listing 3: SmallestSubnet Example

```
[Fact]
public void SmallestSubnet_Example()
{
    // Arrange
    var tall = Subnet.Parse("255.255.255.254/31");  // 2^1 = 2
    var grande = Subnet.Parse("192.168.1.0/24");    // 2^8 = 256
    var vente = Subnet.Parse("10.10.0.0/16");       // 2^16 = 65536
    var trenta = Subnet.Parse("16.240.0.0/12");     // 2^20 = 1048576
```

(continues on next page)

(continued from previous page)

```
    var subnets = new[] { tall, grande, vente, trenta };

    // Act
    var result = SubnetUtilities.SmallestSubnet(subnets);

    // Assert
    Assert.Equal(tall, result);
}
```

CHAPTER

SIX

IP ADDRESS RANGE COMPARERS

Unsurprisingly, sometimes it is necessary to compare an *IIPAddressRange* to another. For that an implementation of a `Comparer<IIPAddressRange>` is just what the code monkey ordered.

6.1 DefaultIPAddressRangeComparer

Note: the `DefaultIPAddressRangeComparer` will happily compare `IIPAddressRange` of differing address families.

The `DefaultIPAddressRangeComparer` is a `Comparer<IIPAddressRange>` that compares implementations of `IIPAddressRange` first by their `IIPAddressRange.Head` and then by their total length.

By default the two `IIPAddressRange.Head` values are compared via the *DefaultIPAddressComparer*, but that may be overridden by providing your own `IComparer<IPAddress>` to the appropriate constructor.

```
public DefaultIPAddressRangeComparer()
```

```
public DefaultIPAddressRangeComparer(IComparer<IPAddress> ipAddressComparer)
```

CHAPTER

SEVEN

IP ADDRESS CONVERTERS

`Arcus.Converters.IPAddressConverters` is a static utility class containing conversion methods for converting `IPAddress` objects into something else.

7.1 Integer Converters

Integer Converters are used to turn an `IPAddress` into an integer value.

7.1.1 Netmask To Cidr Route Prefix

> **Warning:** This operation only valid for IPv4 netmasks.

`NetmaskToCidrRoutePrefix` will convert the valid IPv4 `IPAddress netmask` into a CIDR route prefix.

```
public static int NetmaskToCidrRoutePrefix(this IPAddress netmask)
```

The following example generates a table of all route prefixes for the equivalent netmask `IPAddress` input. Note that this example uses Gulliver[1] in order to deal with byte manipulation.

Listing 1: NetmaskToCidrRoutePrefix Example

```
public void NetmaskToCidrRoutePrefix_Example()
{
    // equivalent byte value of 255.255.255.255 or 2^32
    var maxIPv4Bytes = Enumerable.Repeat((byte) 0xFF, 4)
                                 .ToArray();

    // build all valid net masks
    var allNetMasks = Enumerable.Range(7, 10)
                                .Select(i => maxIPv4Bytes.ShiftBitsLeft(32 - i)) //␣
↪use Gulliver to shift bits of byte array
                                .Select(b => new IPAddress(b))
                                .ToArray();

    var sb = new StringBuilder();
```

(continues on next page)

[1] Interested in byte manipulation? Is endianess your calling? You should check out Gulliver, an awesome opensource C# library developed by a number of smart and attractive people that like playing with thier bits.

(continued from previous page)

```
    foreach (var netmask in allNetMasks)
    {
        var routePrefix = netmask.NetmaskToCidrRoutePrefix();
        _ = sb.Append(routePrefix)
              .Append('\t')
              .AppendFormat("{0,-15}", netmask)
              .Append('\t')
              .Append(netmask.GetAddressBytes()
                             .ToString("b")) // using Gulliver to print bytes as bits
              .AppendLine();
    }
    this.output.WriteLine(sb.ToString());
}
```

Listing 2: NetmaskToCidrRoutePrefix Example Output

```
7   254.0.0.0       11111110 00000000 00000000 00000000
8   255.0.0.0       11111111 00000000 00000000 00000000
9   255.128.0.0     11111111 10000000 00000000 00000000
10  255.192.0.0     11111111 11000000 00000000 00000000
11  255.224.0.0     11111111 11100000 00000000 00000000
12  255.240.0.0     11111111 11110000 00000000 00000000
13  255.248.0.0     11111111 11111000 00000000 00000000
14  255.252.0.0     11111111 11111100 00000000 00000000
15  255.254.0.0     11111111 11111110 00000000 00000000
16  255.255.0.0     11111111 11111111 00000000 00000000
```

7.2 String Converters

Unfortunately `IPAddress` does not implement `IFormattable`, and we chose for compatibility sake not to to extend `IPAddress` with our own proxy class. This however does not mean we don't want that precious data hidden within.

It should not be a profound world changing experience to realize that string converters will convert `IPAddress` to a `string`. Game changing perhaps, but not world changing.

7.2.1 ToDottedQuadString

`ToDottedQuadString` will take the IPv6 input of `IPAddress ipAddress` and convert it into a dotted quad representation.

> **Warning:** A non-IPv6 input will cause the method to simply return the value of the input `IPAddress`.

```
public static string ToDottedQuadString(this IPAddress ipAddress)
```

The example below shows the output generated by calling the `ToDottedQuadString` extension method on an `IPAddress`.

Listing 3: ToDottedQuadString Example

```
public void ToDottedQuadString_Example()
{
    var addresses = new[]
    {
        "::",
        "::ffff",
        "a:b:c::ff00:ff",
        "ffff::",
        "ffff::0102:0304",
        "ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff"
    }.Select(IPAddress.Parse)
    .ToArray();

    var sb = new StringBuilder();

    foreach (var address in addresses)
    {
        var dottedQuadString = address.ToDottedQuadString();

        sb.AppendFormat("{0,-40}", address)
            .Append('\t').Append("=>").Append('\t')
            .Append(dottedQuadString)
            .AppendLine();
    }

    output.WriteLine(sb.ToString());
}
```

Listing 4: ToDottedQuadString Example Output

```
::                                          =>      ::0.0.0.0
::ffff                                      =>      ::0.0.255.255
a:b:c::ff00:ff                              =>      a:b:c::255.0.0.255
ffff::                                      =>      ffff::0.0.0.0
ffff::102:304                               =>      ffff::1.2.3.4
ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff     =>
↪ffff:ffff:ffff:ffff:ffff:ffff:255.255.255.255
```

7.2.2 ToHexString

`ToHexString` may be used to encode the `IPAddress ipAddress` as a Big-Endian[1] ordered string. It will keep all zero-valued most significant bytes.

```
public static string ToHexString(this IPAddress ipAddress)
```

The example below shows the output created by calling the `ToHexString` extension method on an `IPAddress`.

Listing 5: ToHexString Example

```
public void ToHexString_Example()
{
    var addresses = new[]
    {
```

(continues on next page)

(continued from previous page)

```
            "::",
            "::ffff",
            "10.1.1.1",
            "192.168.1.1",
            "255.255.255.255",
            "ffff::",
            "ffff::0102:0304",
            "ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff"
        }.Select(IPAddress.Parse)
        .ToArray();

        var sb = new StringBuilder();

        foreach (var address in addresses)
        {
            var hexString = address.ToHexString();

            sb.AppendFormat("{0,-40}", address)
                .Append('\t').Append("=>").Append('\t')
                .Append(hexString)
                .AppendLine();
        }

        output.WriteLine(sb.ToString());
    }
```

Listing 6: ToHexString Example Output

```
::                                      =>      00000000000000000000000000000000
::ffff                                  =>      00000000000000000000000000000FFFF
10.1.1.1                                =>      0A010101
192.168.1.1                             =>      C0A80101
255.255.255.255                         =>      FFFFFFFF
ffff::                                  =>      FFFF0000000000000000000000000000
ffff::102:304                           =>      FFFF0000000000000000001020304
ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff =>      FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

7.2.3 ToNumericString

`ToNumericString` takes the provided `IPAddress ipAddress` and will return a string representing an unsigned integer value of said address.

Note: The return value will be somewhere between 0 and 340282366920938463463374607431768211455.

```
public static string ToNumericString(this IPAddress ipAddress)
```

The example below shows the output created by calling the `ToNumericString` extension method on an `IPAddress`.

Listing 7: ToNumericString Example

```
public void ToNumericString_Example()
{
```

(continues on next page)

(continued from previous page)

```
    var addresses = new[]
    {
        "::",
        "::ffff",
        "10.1.1.1",
        "192.168.1.1",
        "255.255.255.255",
        "ffff::",
        "ffff::0102:0304",
        "ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff"
    }.Select(IPAddress.Parse)
    .ToArray();

    var sb = new StringBuilder();

    foreach (var address in addresses)
    {
        var numericString = address.ToNumericString();

        sb.AppendFormat("{0,-40}", address)
            .Append('\t').Append("=>").Append('\t')
            .Append(numericString)
            .AppendLine();
    }

    output.WriteLine(sb.ToString());
}
```

Listing 8: ToNumericString Example Output

```
::                                      =>      0
::ffff                                  =>      65535
10.1.1.1                                =>      167837953
192.168.1.1                             =>      3232235777
255.255.255.255                         =>      4294967295
ffff::                                  =>      ␣
↪340277174624079928635746076935438991360
ffff::102:304                           =>      ␣
↪340277174624079928635746076935455900420
ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff     =>      ␣
↪340282366920938463463374607431768211455
```

7.2.4 ToUncompressedString

`ToUncompressedString` converts the given `IPAddress ipAddress` input to an "uncompressed" IPv4 or IPv6 address string.

The function will add appropriate most significant zeros between octects and hextets, as well as expanding '::' to the appropriate zeroed-hextets in IPv6 addresses.

```
public static string ToUncompressedString(this IPAddress ipAddress)
```

The example below shows the output created by calling the `ToUncompressedString` extension method on an `IPAddress`.

Listing 9: ToUncompressedString Example

```
public void ToUncompressedString_Example()
{
    var addresses = new[]
    {
        "::",
        "::ffff",
        "10.1.1.1",
        "192.168.1.1",
        "255.255.255.255",
        "ffff::",
        "ffff::0102:0304"
    }.Select(IPAddress.Parse)
    .ToArray();

    var sb = new StringBuilder();

    foreach (var address in addresses)
    {
        var uncompressedString = address.ToUncompressedString();

        sb.AppendFormat("{0,-40}", address)
            .Append('\t').Append("=>").Append('\t')
            .Append(uncompressedString)
            .AppendLine();
    }

    output.WriteLine(sb.ToString());
}
```

Listing 10: ToUncompressedString Example Output

```
::                                      =>      ␣
↪0000:0000:0000:0000:0000:0000:0000:0000
::ffff                                  =>      ␣
↪0000:0000:0000:0000:0000:0000:0000:ffff
10.1.1.1                                =>       010.001.001.001
192.168.1.1                             =>       192.168.001.001
255.255.255.255                         =>       255.255.255.255
ffff::                                  =>      ␣
↪ffff:0000:0000:0000:0000:0000:0000:0000
ffff::102:304                           =>      ␣
↪ffff:0000:0000:0000:0000:0000:0102:0304
```

7.2.5 ToBase85String

`ToBase85String` will take an IPv6 `IPAddress ipAddress` and convert it to Base85, AKA Ascii85, in accordance to RFC1924[2] which defines a "A Compact Representation of IPv6 Addresses".

Note: The input of a non-IPv6 address will return an empty string.

[2] RFC 1924 is an April Fools Day Joke, but we implemented it anyhow. The question is, did we realize it was a joke before we implemented it or not. Ah, programmer jokes. There are 10 types of developers out there, those that get the joke, and those that don't.

```
public static string ToBase85String(this IPAddress ipAddress)
```

The example below shows the output created by calling the `ToBase85String` extension method on an `IPAddress`.

Listing 11: ToBase85String Example

```
public void ToBase85String_Example()
{
    var addresses = new[]
    {
        "::",
        "::ffff",
        "1080:0:0:0:8:800:200C:417A", // specific example from RFC 1924
        "ffff::",
        "ffff::0102:0304",
        "ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff"
    }.Select(IPAddress.Parse)
    .ToArray();

    var sb = new StringBuilder();

    foreach (var address in addresses)
    {
        var base85String = address.ToBase85String();

        sb.AppendFormat("{0,-40}", address)
            .Append('\t').Append("=>").Append('\t')
            .Append(base85String)
            .AppendLine();
    }

    output.WriteLine(sb.ToString());
}
```

Listing 12: ToBase85String Example Output

```
::                                      =>      0000000000000000000
::ffff                                  =>      0000000000000000960
1080::8:800:200c:417a                   =>      4)+k&C#VzJ4br>0wv%Yp
ffff::                                  =>      =q{+M|w0(OeO5^EGP660
ffff::102:304                           =>      =q{+M|w0(OeO5^EGqpaA
ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff =>      =r54lj&NUUO~Hi%c2ym0
```

CHAPTER

EIGHT

IP ADDRESS MATH

Too frequently the existing implementation of the C# `IPAddress` object is too limited for anything beyond some of the most trivial interactions. Mathematical operations in fact are wholly absent, forcing developers to directly manipulate bytes[1], often requiring a great deal of manual implementation of non-existent byte math. Don't worry though, Arcus is here to fill in some of those gaps.

Note: Unless otherwise specified regarding the math of the `IPAddress` object treats it as an unsigned integer based on its bytes interpenetrated as 32-bit for IPv4 and 128-bit for IPv6 all in big-endian byte order.

8.1 Increment

Incrementing an `IPAddress` allows for the the addition or subtraction of a provided optional `long delta` value.

There exist two implementations of Increment methods. `Increment` and the safe `TryIncrement`.

```
public static IPAddress Increment(this IPAddress input, long delta = 1)
```

```
public static bool TryIncrement(IPAddress input, out IPAddress address, long delta =␣
↪1)
```

8.2 Comparisons

8.2.1 Compare to Another IPAddress

The `IPAddress` does not implement the standard comparison operators, and thus far we can't write extension methods for operators on a class[2]. Arcus did the next best thing, deciding not to extend the `IPAddress`, opting to provide a handful of simple extension methods to bend the will of the `IPAddress` to suit our needs.

Note: Barring the use of the methods below, the *DefaultIPAddressComparer* may also be of interest to you.

It should be pretty obvious based on name alone as to what each of the following five methods will accomplish:

[1] If you actually want to manipulate bytes take a gander at Gulliver, an C# library developed by the same folks that wrote Arcus. They're kinda great.

[2] A GitHub issue for Extension function members requesting a champion for some proposed changes regarding the future of extension methods.

```
public static bool IsEqualTo(this IPAddress left, IPAddress right)
```

```
public static bool IsGreaterThan(this IPAddress left, IPAddress right)
```

```
public static bool IsGreaterThanOrEqualTo(this IPAddress left, IPAddress right)
```

```
public static bool IsLessThan(this IPAddress left, IPAddress right)
```

```
public static bool IsLessThanOrEqualTo(this IPAddress left, IPAddress right)
```

8.2.2 Get IsBetween

Slightly different than the other comparison extension method above is the *IsBetween* method. As is hopefully is obvious it will test if an `IPAddress` occurs numerically between the given `high` and `low` addresses. Likewise the *inclusive* bit may be set to include equality to either `low` or `high` to be considered an inclusive between.

```
public static bool IsBetween(this IPAddress input, IPAddress low, IPAddress high,␣
↪bool inclusive = true)
```

8.2.3 Get Min / Max

The `Min` and `Max` methods will return the `IPAddress left` or `IPAddress right` that is the smallest or largest of the two respectively.

```
public static IPAddress Min(IPAddress left, IPAddress right)
```

```
public static IPAddress Max(IPAddress left, IPAddress right)
```

8.2.4 Determine Scale

`IsAtMin` and `IsAtMax` tests the `IPAddress address` to determine if it is at its minimum or maximum value respectively.

Note: For IPv4 the minimum value is `0.0.0.0` (0), and maximum is `255.255.255.255` ($2^{32} - 1$)

Note: For IPv6 the minimum value is `::` (0), and maximum is `ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff` ($2^{128} - 1$)

```
public static bool IsAtMin(this IPAddress address)
```

```
public static bool IsAtMax(this IPAddress address)
```

CHAPTER

NINE

IP ADDRESS UTILITIES

`Arcus.Utilities.IPAddressUtilities` is a static utility class containing miscellaneous methods and definitions for the `IPAddress` object.

9.1 Useful Values

Included within are some handy-dandy constant values and static readonly properties:

`int` IPv4BitCount The number of bits in an IPv4 address (`32`)

`int` IPv4ByteCount The number of bytes in an IPv4 address (`4`)

`int` IPv4OctetCount The number of octets in an IPv4 address (`4`)

`int` IPv6BitCount The number of bits in an IPv6 address (`128`)

`int` IPv6ByteCount The number of bytes in an IPv6 address (`16`)

`int` IPv6HextetCount The number of hextets in an IPv6 address (`8`)

`IPAddress` IPv4MaxAddress The maximum IPv4 Address value (`0.0.0.0`)

`IPAddress` IPv4MinAddress The minimum IPv4 Address value (`255.255.255.255`)

`IPAddress` IPv6MaxAddress The maximum IPv6 value (`ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff`)

`IPAddress` IPv6MinAddress The minimum IPv6 value (`::`)

`IReadOnlyCollection<AddressFamily>` ValidAddressFamilies The standard valid `AddressFamily` values (`InterNetwork` and `InterNetworkV6`)

9.2 Methods

9.2.1 Minimum and Maximum Address

Given an instance of `AddressFamily` the `MinIPAddress` and `MaxIPAddress` methods will return the minimum value of an address with the `AddressFamily` or the maximum value respectively.

> **Warning:** these methods are only valid for `InterNetwork` and `InterNetworkV6`

```
public static IPAddress MinIPAddress(this AddressFamily addressFamily)
```

```
public static IPAddress MaxIPAddress(this AddressFamily addressFamily)
```

9.2.2 Address Family Detection

Given an instance of `IPAddress ipAddress` the `IsIPv4` and `IsIPv6` methods will return `true` if the given address has the address family `InterNetwork` or `InterNetworkV6` respectively.

```
public static bool IsIPv4(this IPAddress ipAddress)
```

```
public static bool IsIPv6(this IPAddress ipAddress)
```

9.2.3 Address Format Detection

Arcus provides a few ways to detect the format of an `IPAddress` that isn't already built into the pre-existing C# packages.

IsIPv4MappedIPv6

`IsIPv4MappedIPv6` will return `true` if, and only if,‘‘IPAddress ipAddress‘‘ is an IPv4 addressed mapped to IPv6.

This check is made in accordance of in accordance to RFC4291 - IP Version 6 Addressing Architecture - 2.5.5.2. “IPv4-Mapped IPv6 Address.”

```
public static bool IsIPv4MappedIPv6(this IPAddress ipAddress)
```

IsValidNetMask

`IsValidNetMask` checks if the given `IPAddress netmask` is a valid IPv4 netmask, if, and only if, it is then the method returns `true`.

```
public static bool IsValidNetMask(this IPAddress netmask)
```

9.2.4 Parsing

Arcus provides a few more out of the box parsing mechanisms to convert different types of input into an `IPAddress`.

Most of these new parsing routines have a “safe” method that will be prefixed by “Try” that will return `true` on a successful parsing and will *out* the `IPAddress`.

Hexadecimal

`ParseFromHexString` and `TryParseFromHexString` will attempt to parse a hexadecimal `string input` as an IP Address of the given `AddressFamily addressFamily`.

Note: Valid input must be comprised of only hexadecimal characters with an optional “0x” prefix. Input is case insensitive, and assumed to be in big-endian byte order. Zero valued most significant bytes will be ignored.

```
public static IPAddress ParseFromHexString(string input, AddressFamily addressFamily)
```

```
public static bool TryParseFromHexString(string input, AddressFamily addressFamily,␣
↪out IPAddress address)
```

Octal

By Microsoft's implementation of the `IPAddress.Parse(string)` any string representation of an IP Address having a zero-valued most significant number in an octet position is interpreted as octal (base 8) rather than decimal (base 10). This isn't always a desired way to go about parsing values.

These methods convert an `string input` IPv4 address representation to `IPAddress` instance ignoring leading zeros (octal notation) of dotted quad format.

```
public static IPAddress ParseIgnoreOctalInIPv4(string input)
```

```
public static bool TryParseIgnoreOctalInIPv4(string input, out IPAddress address)
```

byte[]

The following `byte[]` parsing methods will attempt to convert a big-endian ordered byte array to an `IPAddress` automatically providing the appropriate number of zero-valued most significant bytes as needed to meet the desired address family.

Note: This implementation differs from the constructor implementation on `IPAddress` that takes `byte[]` as input. Said constructor takes an explicit sized byte array and will outright fail if the input isn't explicitly 4 or 16 bytes long.

```
public static IPAddress Parse(byte[] input, AddressFamily addressFamily)
```

```
public static bool TryParse(byte[] input, AddressFamily addressFamily, out IPAddress␣
↪address)
```

CHAPTER

TEN

IPADDRESS COMPARERS

IP Addresses are just numbers. Numbers are comparable. Some are bigger, some are smaller, some are even equal.

10.1 DefaultIPAddressComparer

Note: the `DefaultIPAddressComparer` will gladly compare `IPAddress` of differing address families.

The `DefaultIPAddressComparer` extends `Comparer<IPAddress>`. Its behavior is to first compare two `IPAddress` objects via the `IComparer<AddressFamily>` and then ordinally based on the `IPAddress` big-endian unsigned integer value.

By default the *DefaultAddressFamilyComparer* is used to compare the address families of the addresses, but that may be overridden by providing your own `IComparer<AddressFamily>` to the appropriate constructor

```
public DefaultIPAddressComparer()
```

```
public DefaultIPAddressComparer(IComparer<AddressFamily> addressFamilyComparer)
```

CHAPTER

ELEVEN

ADDRESSFAMILY COMPARERS

AddressFamily comparers are simply classes that extend `Comparer<AddressFamily>`.

11.1 DefaultAddressFamilyComparer

Behind the scenes `AddressFamily` is simply an `enum`. Typically we're only concerned with `InterNetwork`, with a value of 2, and `InternNetworkV6` which is valued at 23.

The `DefaultAddressFamilyComparer` is used to compare the address families of the addresses. No real magic here, we're simply comparing two `AddressFamily` values based on their inherit inherit value.

Listing 1: Compare Implementation

```
public override int Compare(AddressFamily x,
                            AddressFamily y)
{
    return x.CompareTo(y);
}
```

CHAPTER

TWELVE

MACADDRESS

The `MacAddress` type represents a 48-bit MAC Address[1] as per the IEEE EUI standard[3]. It serves the purpose of a Networking Adjacent worker class, and as a handy way to represent, store, format, and compare MAC addresses.

The `MacAddress` class implements `IEquatable<MacAddress>`, `IComparable<MacAddress>`, `IComparable`, `IFormattable`, and `ISerializable`.

Note: Unless otherwise stated recognized readable MAC Address formats include only the following formats:

- IEEE 802 format for printing **EUI-48** and **MAC-48** addresses in six groups of two hexadecimal digits, separated by a dash (`-`). *E.g.* `AA-BB-CC-DD-EE-FF`
- Common Six groups of two hexadecimal digits separated by colons (`:`). *E.g.* `AA:BB:CC:DD:EE:FF`
- Six groups of two hexadecimal digits separated by a space character. *E.g.* `AA BB CC DD EE FF`
- 12 hexadecimal digits with no delimitation. *E.g.* `AABBCCDDEEFF`
- Cisco three groups of four hexadecimal digits separated by dots (`.`). *E.g.* `AABB.CCDD.EEFF`

For the sake of parsing and reading these formats are case insensitive.

Fig. 1: Structure of a MAC-48 Address

12.1 Creation

12.1.1 Constructor

`IEnumerable<byte>`

A new `MacAddress` may be constructed by providing an `IEnumerable<byte>` of six bytes to the constructor.

```
public MacAddress(IEnumerable<byte> bytes)
```

[1] **48-Bit MAC** is a Media Access Control Address (MAC) following both the now deprecated *MAC-48* and the active *EUI-48* specifications.
[3] Guidelines for Use of Extended Unique Identifier (EUI), Organizationally Unique Identifier (OUI), and Company ID (CID)

12.1.2 Factory

Parse `string`

A `MacAddress` may also be created via either the `Parse` or safe `TryParse` method. Not that these methods are strict in that they will only succeed with a MAC address in a known format. If you wish to more liberally parse a string into a `MacAddress` see the `ParseAny` and `TryParseAny` defined below.

```
public static MacAddress Parse(string input)
```

```
public static bool TryParse(string input, out MacAddress macAddress)
```

ParseAny `string`

`ParseAny` and the safe `TryParseAny` allow the parsing of an arbitrary string that may be a Mac address into a `MacAddress`. It looks for six hexadecimal digits within the string, joins them and interprets the result as consecutive big-endian hextets. If six, and only six, hexadecimal digits are not found the parse will fail.

```
public static MacAddress ParseAny(string input)
```

```
public static bool TryParseAny(string input, out MacAddress macAddress)
```

12.2 Functionality

12.2.1 Properties

`bool` IsDefault returns `true` if, and only if, the MAC Address is the EUI-48 default[2], meaning all bits of the MAC Address are set making it equivalent to `FF:FF:FF:FF:FF:FF`.

`bool` IsGloballyUnique returns `true` if, and only if, is globally unique (OUI[4] enforced).

`bool` IsLocallyAdministered returns `true` if, and only if, is locally administered.

`bool` IsMulticast returns `true` if, and only if, the MAC Address is multicast.

`bool` IsUnicast returns `true` if, and only if, the MAC Address is unicast.

`bool` IsUnusable returns `true` if, and only if, the MAC Address is "unusable", meaning all OUI bits of the MAC Address are unset.

`MacAddress` DefaultMacAddress Provides a `MacAddress` that represents the default or `null` case MAC address.

`Regex` AllFormatMacAddressRegularExpression Returns a regular expression for matching accepted MAC Address formats.

`Regex` CommonFormatMacAddressRegularExpression Returns a regular expression for matching the "common" six groups of two uppercase hexadecimal digits format.

`string` AllFormatMacAddressPattern Returns a regular expression pattern for matching accepted MAC Address formats.

[2] The recommended null or default value for **EUI-48** is `FF-FF-FF-FF-FF-FF`

[4] *Organizationally Unique Identifier (OUI)* is the first 3-bytes (24-bits) of a MAC-48 MAC Address.

`string` CommonFormatMacAddressPattern Returns a regular expression pattern for matching the "common" six groups of two uppercase hexadecimal digits format.

12.2.2 Methods

GetAddressBytes

`GetAddressBytes` returns a copy of the underlying big-endian bytes of the `MacAddress`. This will always be six bytes in length.

```
public byte[] GetAddressBytes()
```

GetOuiBytes

`GetOuiBytes` returns the *Organizationally Unique Identifier (OUI)*[4] of the `MAcAddress`.

```
public byte[] GetOuiBytes()
```

GetCidBytes

`GetCidBytes` returns the *Company ID (CID)*[5] of the `MAcAddress`.

```
public byte[] GetCidBytes()
```

12.2.3 IFormatable

`MacAddress` offers a number or preexisting formats that are accessible via the standard `ToString` method provided by `IFormattable` interface.

Table 1: Subnet format values

Format	Name	Description	Example
`g`	General Format	Uppercase hexadecimal encoded bytes separated by colons	`AA:BB:CC:DD:EE:FF`
`H`	Uppercase Hexadecimal	Contiguous uppercase hexadecimal digits	`AABBCCDDEEFF`
`h`	Lowercase Hexadecimal	Contiguous lowerrcase hexadecimal digits	`aabbccddeeff`
`c`	Cisco	Three groups of four uppercase hexadecimal digits separated by periods	`AAAA.BBBB.CCCC`
`s`	Space delimited hextets	Hexadecimal bytes separated by a space character	`AA BB CC DD EE FF`
`d`	IEEE 802	Hexadecimal encoded bytes separated by a dash characte	`AA-BB-CC-DD-EE-FF`
`i`	Integer	Big-endian integer value	`187723572702975`

[5] *Company Id (Cid)* is the last 3-bytes (24-bits) of a MAC-48 MAC Address.

12.2.4 Operators

`MacAddress` implements all the standard C# equality and comparison operators. The comparison operators treat the `MacAddress` bytes as an unsigned big-endian integer value.

CHAPTER

THIRTEEN

FREQUENTLY ASKED QUESTIONS

13.1 IPv6 is big, huh?

Yes

13.1.1 Can you elaborate?

Absolutely.

There are 2^{128} possible IPv6 Addresses, compared to the 2^{32} possible IPv4 addresses.

That's roughly 3.4×10^{38} addresses.

340 undecillion 282 decillion 366 nonillion 920 octillion 938 septillion 463 sextillion 463 quintillion 374 quadrillion 607 trillion 431 billion 768 million 211 thousand 456 to be exact.

Let's face it, arbitrary numbers much bigger than 7 are hard to conceptualize for some of us[1]. I personally get lost after three-ish. The awe inspiring scale of IPv6 is much bigger than 3, at least double, probably even over triple that. It is so big we had to jump through some hoops to make C# do the math necessary. This is why both the Arcus and Gulliver libraries now exist.

As a thought exercise let's try to visualize the mighty scale of IPv6.

According to un data estimates there are approximately 7.55 billion people alive as I write this sentence.

If we take all 2^{128} IPv6 addresses and distribute them equally amongst everyone we'd each get about 4.51×10^{28} addresses. That's a rather lot of IoT devices to keep track of.

Thanks to new and inventive imaginary non-existent technology we're going to assign each of our grain of sand sized network devices an address from our own personal IPv6 address pool. This will be a wireless network obviously, it is rather difficult to jam a RJ45 cable into something $0.05mm^3$.

As it turns out that's approximately $2.25 \times 10^{19} m^3$ of much bigger than nano-bot devices you've got there. Hope you have some deep pockets, as that's nearly the volume of 1.8 times all of earths oceans. That's per person.

This means that with the power of all our sand-bots combined we'd have roughly the volume of twelve of earth's suns.

Conversely, all 2^{32} IPv4 addresses would slightly overflow a 50-gallon drum amassing a measly 56.7 gallons. It is not a surprise that we've practically exhausted the IPv4 address space. That said, if we mismanage IPv6 we may run out there too, and Arcus will have to do 256-bit or 1024-bit math, I'm ready.

IPv6 is 7.9×10^{28} times larger than IPv4

[1] The number of objects an average person can hold in working memory is about seven. see Wikipedia

CHAPTER

FOURTEEN

GLOSSARY

Arcus Arcus is the lesser known Roman equivalent of the Greek goddess Iris. She is the Olympian messenger god. You know, because IP Addresses and Subnets are all about sending messages. Rainbows are cool too.

AddressFamily The C# AddressFamily is an enum that defines the type of an `IPAddress`. Both IPAddress and Arcus are only concerned with `InterNetwork` an IPv4 address, and `InterNetworkV6` an IPv6 address.

CIDR Short for **Classless Inter-Domain Routing**, is a way of expressing a range of IP addresses.

see CIDR on Wikipedia

Endianness Endianness referees to the ordering of bytes in the binary representation of data.

see Endianness on Wikipedia

Big-Endian Big-Endian ordering, at times also referred to as *Network Byte Order*, is a left-to-right ordering of bytes where the left most bytes are most significant than right most.

For example, the decimal value of the unsigned integer `6060842` may be represented as `0x5C7B2A` in hexadecimal. This hexadecimal value is composed of the three bytes `0x5C`, `0x7B`, and `0x28`. As such the value `6060842` may be represented in Big-Endian as a byte array of `[0x5C, 0x7B, 0x2A]`.

see Gulliver's What is Endianness

Gulliver 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. It was developed by the same folks who created Arcus.

see Gulliver on GitHub

IP Address Short for **Internet Protocol Address** it is a numeric representation that typically comes in two flavors IPv4 and IPv6.

see IP Address on Wikipedia

IPv4 IPv4 is an IP Address that follows version 4 of the Internet Protocol. It is a 32-bit number, four bytes, with 2^{32} distinct addresses. IPv4 Addresses are typically represented in a format referred to as *Dotted Quad* or *Quad-dotted* in which the four bytes making the address are delimited by a period (.) character in decimal big-endian order, such as `192.168.1.0`.

see IPv4 on Wikipedia

IPv6 IPv6 is an IP Address following version 6 of the Internet Protocol. It is a 128-bit number, 16 bytes, with 2^{128} distinct addresses. It is typically expressed in a "human readable"[1] format in Big-Endian byte order typically with hextets delimited with colons and collapses, such as the equivalent `fd04:f0bf:44a0:df4e::` and `fd04:f0bf:44a0:df4e:0000:0000:0000:0000`.

[1] And by "human readable" the author means a draconian format consisting of groupings of two byte hextets delimited by colons that aren't always two bytes long and sometimes the colons do funny things as do zeros, and oh yeah, occasionally the IPv4 dotted-quad format pops up and makes things even more interesting. see RFC5952.

see IPv6 on Wikipedia

Subnet Subnet, also known as **Subnetwork**, is a logical subdivision of an Internet Protocol network. Much like IP Addresses they come in both IPv4 and IPv6 flavors.

see Subnetwork on Wikipedia

CHAPTER

FIFTEEN

HANDY REFERENCES

15.1 IPv4 CIDR Table

Table 1: Subnet format values

CIDR	Network Prefix Address	Route Prefix	Netmask	Netmask (bits)
255.255.255.255/32	255.255.255.255	32	255.255.255.255	11111111 11111111 11111111 11111111
255.255.255.254/31	255.255.255.254	31	255.255.255.254	11111111 11111111 11111111 11111110
255.255.255.252/30	255.255.255.252	30	255.255.255.252	11111111 11111111 11111111 11111100
255.255.255.248/29	255.255.255.248	29	255.255.255.248	11111111 11111111 11111111 11111000
255.255.255.240/28	255.255.255.240	28	255.255.255.240	11111111 11111111 11111111 11110000
255.255.255.224/27	255.255.255.224	27	255.255.255.224	11111111 11111111 11111111 11100000
255.255.255.192/26	255.255.255.192	26	255.255.255.192	11111111 11111111 11111111 11000000
255.255.255.128/25	255.255.255.128	25	255.255.255.128	11111111 11111111 11111111 10000000
255.255.255.0/24	255.255.255.0	24	255.255.255.0	11111111 11111111 11111111 00000000
255.255.254.0/23	255.255.254.0	23	255.255.254.0	11111111 11111111 11111110 00000000
255.255.252.0/22	255.255.252.0	22	255.255.252.0	11111111 11111111 11111100 00000000
255.255.248.0/21	255.255.248.0	21	255.255.248.0	11111111 11111111 11111000 00000000
255.255.240.0/20	255.255.240.0	20	255.255.240.0	11111111 11111111 11110000 00000000
255.255.224.0/19	255.255.224.0	19	255.255.224.0	11111111 11111111 11100000 00000000
255.255.192.0/18	255.255.192.0	18	255.255.192.0	11111111 11111111 11000000 00000000
255.255.128.0/17	255.255.128.0	17	255.255.128.0	11111111 11111111 10000000 00000000
255.255.0.0/16	255.255.0.0	16	255.255.0.0	11111111 11111111 00000000 00000000
255.254.0.0/15	255.254.0.0	15	255.254.0.0	11111111 11111110 00000000 00000000
255.252.0.0/14	255.252.0.0	14	255.252.0.0	11111111 11111100 00000000 00000000
255.248.0.0/13	255.248.0.0	13	255.248.0.0	11111111 11111000 00000000 00000000
255.240.0.0/12	255.240.0.0	12	255.240.0.0	11111111 11110000 00000000 00000000
255.224.0.0/11	255.224.0.0	11	255.224.0.0	11111111 11100000 00000000 00000000
255.192.0.0/10	255.192.0.0	10	255.192.0.0	11111111 11000000 00000000 00000000
255.128.0.0/9	255.128.0.0	9	255.128.0.0	11111111 10000000 00000000 00000000
255.0.0.0/8	255.0.0.0	8	255.0.0.0	11111111 00000000 00000000 00000000
254.0.0.0/7	254.0.0.0	7	254.0.0.0	11111110 00000000 00000000 00000000
252.0.0.0/6	252.0.0.0	6	252.0.0.0	11111100 00000000 00000000 00000000
248.0.0.0/5	248.0.0.0	5	248.0.0.0	11111000 00000000 00000000 00000000
240.0.0.0/4	240.0.0.0	4	240.0.0.0	11110000 00000000 00000000 00000000
224.0.0.0/3	224.0.0.0	3	224.0.0.0	11100000 00000000 00000000 00000000
192.0.0.0/2	192.0.0.0	2	192.0.0.0	11000000 00000000 00000000 00000000
128.0.0.0/1	128.0.0.0	1	128.0.0.0	10000000 00000000 00000000 00000000
0.0.0.0/0	0.0.0.0	0	0.0.0.0	00000000 00000000 00000000 00000000

15.2 IPv6 CIDR Table

Table 2: Subnet format values

CIDR	Network Prefix Address	Route Prefix	Address Count
ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff/128	ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff	128	1
ffff:ffff:ffff:ffff:ffff:ffff:ffff:fffe/127	ffff:ffff:ffff:ffff:ffff:ffff:ffff:fffe	127	2
ffff:ffff:ffff:ffff:ffff:ffff:ffff:fffc/126	ffff:ffff:ffff:ffff:ffff:ffff:ffff:fffc	126	4
ffff:ffff:ffff:ffff:ffff:ffff:ffff:fff8/125	ffff:ffff:ffff:ffff:ffff:ffff:ffff:fff8	125	8
ffff:ffff:ffff:ffff:ffff:ffff:ffff:fff0/124	ffff:ffff:ffff:ffff:ffff:ffff:ffff:fff0	124	16
ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffe0/123	ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffe0	123	32
ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffc0/122	ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffc0	122	64
ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff80/121	ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff80	121	128
ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff00/120	ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff00	120	256
ffff:ffff:ffff:ffff:ffff:ffff:ffff:fe00/119	ffff:ffff:ffff:ffff:ffff:ffff:ffff:fe00	119	512
ffff:ffff:ffff:ffff:ffff:ffff:ffff:fc00/118	ffff:ffff:ffff:ffff:ffff:ffff:ffff:fc00	118	1024
ffff:ffff:ffff:ffff:ffff:ffff:ffff:f800/117	ffff:ffff:ffff:ffff:ffff:ffff:ffff:f800	117	2048
ffff:ffff:ffff:ffff:ffff:ffff:ffff:f000/116	ffff:ffff:ffff:ffff:ffff:ffff:ffff:f000	116	4096
ffff:ffff:ffff:ffff:ffff:ffff:ffff:e000/115	ffff:ffff:ffff:ffff:ffff:ffff:ffff:e000	115	8192
ffff:ffff:ffff:ffff:ffff:ffff:ffff:c000/114	ffff:ffff:ffff:ffff:ffff:ffff:ffff:c000	114	16384
ffff:ffff:ffff:ffff:ffff:ffff:ffff:8000/113	ffff:ffff:ffff:ffff:ffff:ffff:ffff:8000	113	32768
ffff:ffff:ffff:ffff:ffff:ffff:ffff:0/112	ffff:ffff:ffff:ffff:ffff:ffff:ffff:0	112	65536
ffff:ffff:ffff:ffff:ffff:ffff:fffe:0/111	ffff:ffff:ffff:ffff:ffff:ffff:fffe:0	111	131072
ffff:ffff:ffff:ffff:ffff:ffff:fffc:0/110	ffff:ffff:ffff:ffff:ffff:ffff:fffc:0	110	262144
ffff:ffff:ffff:ffff:ffff:ffff:fff8:0/109	ffff:ffff:ffff:ffff:ffff:ffff:fff8:0	109	524288
ffff:ffff:ffff:ffff:ffff:ffff:fff0:0/108	ffff:ffff:ffff:ffff:ffff:ffff:fff0:0	108	1048576
ffff:ffff:ffff:ffff:ffff:ffff:ffe0:0/107	ffff:ffff:ffff:ffff:ffff:ffff:ffe0:0	107	2097152
ffff:ffff:ffff:ffff:ffff:ffff:ffc0:0/106	ffff:ffff:ffff:ffff:ffff:ffff:ffc0:0	106	4194304
ffff:ffff:ffff:ffff:ffff:ffff:ff80:0/105	ffff:ffff:ffff:ffff:ffff:ffff:ff80:0	105	8388608
ffff:ffff:ffff:ffff:ffff:ffff:ff00:0/104	ffff:ffff:ffff:ffff:ffff:ffff:ff00:0	104	16777216
ffff:ffff:ffff:ffff:ffff:ffff:fe00:0/103	ffff:ffff:ffff:ffff:ffff:ffff:fe00:0	103	33554432
ffff:ffff:ffff:ffff:ffff:ffff:fc00:0/102	ffff:ffff:ffff:ffff:ffff:ffff:fc00:0	102	67108864
ffff:ffff:ffff:ffff:ffff:ffff:f800:0/101	ffff:ffff:ffff:ffff:ffff:ffff:f800:0	101	134217728
ffff:ffff:ffff:ffff:ffff:ffff:f000:0/100	ffff:ffff:ffff:ffff:ffff:ffff:f000:0	100	268435456
ffff:ffff:ffff:ffff:ffff:ffff:e000:0/99	ffff:ffff:ffff:ffff:ffff:ffff:e000:0	99	536870912
ffff:ffff:ffff:ffff:ffff:ffff:c000:0/98	ffff:ffff:ffff:ffff:ffff:ffff:c000:0	98	1073741824
ffff:ffff:ffff:ffff:ffff:ffff:8000:0/97	ffff:ffff:ffff:ffff:ffff:ffff:8000:0	97	2147483648
ffff:ffff:ffff:ffff:ffff:ffff::/96	ffff:ffff:ffff:ffff:ffff:ffff:	96	4294967296
ffff:ffff:ffff:ffff:ffff:fffe::/95	ffff:ffff:ffff:ffff:ffff:fffe:	95	8589934592
ffff:ffff:ffff:ffff:ffff:fffc::/94	ffff:ffff:ffff:ffff:ffff:fffc:	94	17179869184
ffff:ffff:ffff:ffff:ffff:fff8::/93	ffff:ffff:ffff:ffff:ffff:fff8:	93	34359738368
ffff:ffff:ffff:ffff:ffff:fff0::/92	ffff:ffff:ffff:ffff:ffff:fff0:	92	68719476736
ffff:ffff:ffff:ffff:ffff:ffe0::/91	ffff:ffff:ffff:ffff:ffff:ffe0:	91	137438953472
ffff:ffff:ffff:ffff:ffff:ffc0::/90	ffff:ffff:ffff:ffff:ffff:ffc0:	90	274877906944
ffff:ffff:ffff:ffff:ffff:ff80::/89	ffff:ffff:ffff:ffff:ffff:ff80:	89	549755813888
ffff:ffff:ffff:ffff:ffff:ff00::/88	ffff:ffff:ffff:ffff:ffff:ff00:	88	1099511627776
ffff:ffff:ffff:ffff:ffff:fe00::/87	ffff:ffff:ffff:ffff:ffff:fe00:	87	2199023255552
ffff:ffff:ffff:ffff:ffff:fc00::/86	ffff:ffff:ffff:ffff:ffff:fc00:	86	4398046511104
ffff:ffff:ffff:ffff:ffff:f800::/85	ffff:ffff:ffff:ffff:ffff:f800:	85	8796093022208
ffff:ffff:ffff:ffff:ffff:f000::/84	ffff:ffff:ffff:ffff:ffff:f000:	84	17592186044416
ffff:ffff:ffff:ffff:ffff:e000::/83	ffff:ffff:ffff:ffff:ffff:e000:	83	35184372088832

Table 2 – continued from previous page

CIDR	Network Prefix Address	Route Prefix	Address Count
ffff:ffff:ffff:ffff:ffff:c000::/82	ffff:ffff:ffff:ffff:ffff:c000:	82	70368744177664
ffff:ffff:ffff:ffff:ffff:8000::/81	ffff:ffff:ffff:ffff:ffff:8000:	81	140737488355328
ffff:ffff:ffff:ffff:ffff::/80	ffff:ffff:ffff:ffff:ffff:	80	281474976710656
ffff:ffff:ffff:ffff:fffe::/79	ffff:ffff:ffff:ffff:fffe:	79	562949953421312
ffff:ffff:ffff:ffff:fffc::/78	ffff:ffff:ffff:ffff:fffc:	78	1125899906842624
ffff:ffff:ffff:ffff:fff8::/77	ffff:ffff:ffff:ffff:fff8:	77	2251799813685248
ffff:ffff:ffff:ffff:fff0::/76	ffff:ffff:ffff:ffff:fff0:	76	4503599627370496
ffff:ffff:ffff:ffff:ffe0::/75	ffff:ffff:ffff:ffff:ffe0:	75	9007199254740992
ffff:ffff:ffff:ffff:ffc0::/74	ffff:ffff:ffff:ffff:ffc0:	74	18014398509481984
ffff:ffff:ffff:ffff:ff80::/73	ffff:ffff:ffff:ffff:ff80:	73	36028797018963968
ffff:ffff:ffff:ffff:ff00::/72	ffff:ffff:ffff:ffff:ff00:	72	72057594037927936
ffff:ffff:ffff:ffff:fe00::/71	ffff:ffff:ffff:ffff:fe00:	71	144115188075855872
ffff:ffff:ffff:ffff:fc00::/70	ffff:ffff:ffff:ffff:fc00:	70	288230376151711744
ffff:ffff:ffff:ffff:f800::/69	ffff:ffff:ffff:ffff:f800:	69	576460752303423488
ffff:ffff:ffff:ffff:f000::/68	ffff:ffff:ffff:ffff:f000:	68	1152921504606846976
ffff:ffff:ffff:ffff:e000::/67	ffff:ffff:ffff:ffff:e000:	67	2305843009213693952
ffff:ffff:ffff:ffff:c000::/66	ffff:ffff:ffff:ffff:c000:	66	4611686018427387904
ffff:ffff:ffff:ffff:8000::/65	ffff:ffff:ffff:ffff:8000:	65	9223372036854775808
ffff:ffff:ffff:ffff::/64	ffff:ffff:ffff:ffff:	64	18446744073709551616
ffff:ffff:ffff:fffe::/63	ffff:ffff:ffff:fffe:	63	36893488147419103232
ffff:ffff:ffff:fffc::/62	ffff:ffff:ffff:fffc:	62	73786976294838206464
ffff:ffff:ffff:fff8::/61	ffff:ffff:ffff:fff8:	61	147573952589676412928
ffff:ffff:ffff:fff0::/60	ffff:ffff:ffff:fff0:	60	295147905179352825856
ffff:ffff:ffff:ffe0::/59	ffff:ffff:ffff:ffe0:	59	590295810358705651712
ffff:ffff:ffff:ffc0::/58	ffff:ffff:ffff:ffc0:	58	1180591620717411303424
ffff:ffff:ffff:ff80::/57	ffff:ffff:ffff:ff80:	57	2361183241434822606848
ffff:ffff:ffff:ff00::/56	ffff:ffff:ffff:ff00:	56	4722366482869645213696
ffff:ffff:ffff:fe00::/55	ffff:ffff:ffff:fe00:	55	9444732965739290427392
ffff:ffff:ffff:fc00::/54	ffff:ffff:ffff:fc00:	54	18889465931478580854784
ffff:ffff:ffff:f800::/53	ffff:ffff:ffff:f800:	53	37778931862957161709568
ffff:ffff:ffff:f000::/52	ffff:ffff:ffff:f000:	52	75557863725914323419136
ffff:ffff:ffff:e000::/51	ffff:ffff:ffff:e000:	51	151115727451828646838272
ffff:ffff:ffff:c000::/50	ffff:ffff:ffff:c000:	50	302231454903657293676544
ffff:ffff:ffff:8000::/49	ffff:ffff:ffff:8000:	49	604462909807314587353088
ffff:ffff:ffff::/48	ffff:ffff:ffff:	48	1208925819614629174706176
ffff:ffff:fffe::/47	ffff:ffff:fffe:	47	2417851639229258349412352
ffff:ffff:fffc::/46	ffff:ffff:fffc:	46	4835703278458516698824704
ffff:ffff:fff8::/45	ffff:ffff:fff8:	45	9671406556917033397649408
ffff:ffff:fff0::/44	ffff:ffff:fff0:	44	19342813113834066795298816
ffff:ffff:ffe0::/43	ffff:ffff:ffe0:	43	38685626227668133590597632
ffff:ffff:ffc0::/42	ffff:ffff:ffc0:	42	77371252455336267181195264
ffff:ffff:ff80::/41	ffff:ffff:ff80:	41	154742504910672534362390528
ffff:ffff:ff00::/40	ffff:ffff:ff00:	40	309485009821345068724781056
ffff:ffff:fe00::/39	ffff:ffff:fe00:	39	618970019642690137449562112
ffff:ffff:fc00::/38	ffff:ffff:fc00:	38	1237940039285380274899124224
ffff:ffff:f800::/37	ffff:ffff:f800:	37	2475880078570760549798248448
ffff:ffff:f000::/36	ffff:ffff:f000:	36	4951760157141521099596496896
ffff:ffff:e000::/35	ffff:ffff:e000:	35	9903520314283042199192993792
ffff:ffff:c000::/34	ffff:ffff:c000:	34	19807040628566084398385987584

Table 2 – continued from previous page

CIDR	Network Prefix Address	Route Prefix	Address Count
ffff:ffff:8000::/33	ffff:ffff:8000:	33	3961408125713216879677197516
ffff:ffff::/32	ffff:ffff:	32	7922816251426433759354395033
ffff:fffe::/31	ffff:fffe:	31	1584563250285286751870879006
ffff:fffc::/30	ffff:fffc:	30	3169126500570573503741758013
ffff:fff8::/29	ffff:fff8:	29	6338253001141147007483516026
ffff:fff0::/28	ffff:fff0:	28	1267650600228229401496703205376
ffff:ffe0::/27	ffff:ffe0:	27	2535301200456458802993406410752
ffff:ffc0::/26	ffff:ffc0:	26	5070602400912917605986812821504
ffff:ff80::/25	ffff:ff80:	25	1014120480182583521197362564300
ffff:ff00::/24	ffff:ff00:	24	2028240960365167042394725128601
ffff:fe00::/23	ffff:fe00:	23	4056481920730334084789450257203
ffff:fc00::/22	ffff:fc00:	22	8112963841460668169578900514406
ffff:f800::/21	ffff:f800:	21	1622592768292133633915780102881
ffff:f000::/20	ffff:f000:	20	3245185536584267267831560205762
ffff:e000::/19	ffff:e000:	19	6490371073168534535663120411525
ffff:c000::/18	ffff:c000:	18	1298074214633706907132624082305024
ffff:8000::/17	ffff:8000:	17	2596148429267413814265248164610048
ffff::/16	ffff:	16	5192296858534827628530496329220096
fffe::/15	fffe:	15	1038459371706965525706099265844019
fffc::/14	fffc:	14	2076918743413931051412198531688038
fff8::/13	fff8:	13	4153837486827862102824397063376076
fff0::/12	fff0:	12	8307674973655724205648794126752153
ffe0::/11	ffe0:	11	1661534994731144841129758825350430
ffc0::/10	ffc0:	10	3323069989462289682259517650700861
ff80::/9	ff80:	9	6646139978924579364519035301401722
ff00::/8	ff00:	8	1329227995784915872903807060280344
fe00::/7	fe00:	7	2658455991569831745807614120560689
fc00::/6	fc00:	6	5316911983139663491615228241121378
f800::/5	f800:	5	1063382396627932698323045648224275
f000::/4	f000:	4	2126764793255865396646091296448551
e000::/3	e000:	3	4253529586511730793292182592897102
c000::/2	c000:	2	8507059173023461586584365185794205
8000::/1	8000:	1	1701411834604692317316873037158841
::/0		0	3402823669209384634633746074317682

15.3 Valid IPv4 Netmasks

Listing 1: Valid Netmasks

```
0.0.0.0             00000000 00000000 00000000 00000000
128.0.0.0           10000000 00000000 00000000 00000000
192.0.0.0           11000000 00000000 00000000 00000000
224.0.0.0           11100000 00000000 00000000 00000000
240.0.0.0           11110000 00000000 00000000 00000000
248.0.0.0           11111000 00000000 00000000 00000000
252.0.0.0           11111100 00000000 00000000 00000000
254.0.0.0           11111110 00000000 00000000 00000000
255.0.0.0           11111111 00000000 00000000 00000000
255.128.0.0         11111111 10000000 00000000 00000000
255.192.0.0         11111111 11000000 00000000 00000000
```

(continues on next page)

(continued from previous page)

```
255.224.0.0           11111111 11100000 00000000 00000000
255.240.0.0           11111111 11110000 00000000 00000000
255.248.0.0           11111111 11111000 00000000 00000000
255.252.0.0           11111111 11111100 00000000 00000000
255.254.0.0           11111111 11111110 00000000 00000000
255.255.0.0           11111111 11111111 00000000 00000000
255.255.128.0         11111111 11111111 10000000 00000000
255.255.192.0         11111111 11111111 11000000 00000000
255.255.224.0         11111111 11111111 11100000 00000000
255.255.240.0         11111111 11111111 11110000 00000000
255.255.248.0         11111111 11111111 11111000 00000000
255.255.252.0         11111111 11111111 11111100 00000000
255.255.254.0         11111111 11111111 11111110 00000000
255.255.255.0         11111111 11111111 11111111 00000000
255.255.255.128       11111111 11111111 11111111 10000000
255.255.255.192       11111111 11111111 11111111 11000000
255.255.255.224       11111111 11111111 11111111 11100000
255.255.255.240       11111111 11111111 11111111 11110000
255.255.255.248       11111111 11111111 11111111 11111000
255.255.255.252       11111111 11111111 11111111 11111100
255.255.255.254       11111111 11111111 11111111 11111110
255.255.255.255       11111111 11111111 11111111 11111111
```

CHAPTER

SIXTEEN

COMMUNITY

16.1 GitHub

Source Code available on Arcus GitHub

16.2 GITTER

The developers monitor the Arcus Gitter chat drop us a line.

16.3 File an Issue

Issues should be filed on the Arcus GitHub Issue Tracker.

CHAPTER

SEVENTEEN

ACKNOWLEDGEMENTS

17.1 Citations

17.1.1 Arcus logo

Logo cropped from image "Iris Carrying the Water of the River Styx to Olympus for the Gods to Swear By, Guy Head, c. 1793 - Nelson-Atkins Museum of Art" sourced from Wikimedia Commons.

> Exhibit in the Nelson-Atkins Museum of Art, Kansas City, Missouri, USA. Photography was permitted in the museum without restriction
>
> This file is made available under the Creative Commons CC0 1.0 Universal Public Domain Dedication.
>
> The person who associated a work with this deed has dedicated the work to the public domain by waiving all of their rights to the work worldwide under copyright law, including all related and neighboring rights, to the extent allowed by law. You can copy, modify, distribute and perform the work, even for commercial purposes, all without asking permission.

17.1.2 Structure of a 48-bit MAC address

Diagram showing the structure of a MAC-48 network address, explicitly showing the positions of the multicast/unicast bit and the OUI/local address type bit.

> This file is licensed under the Creative Commons Attribution-Share Alike 2.5 Generic, 2.0 Generic and 1.0 Generic license.

INDEX