```csharp
using System;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Exceptions
{
    /// <summary>
    /// <para>
    /// Represents the argument link does not exists exception.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ArgumentException"/>
    public class ArgumentLinkDoesNotExistsException<TLinkAddress> : ArgumentException
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>
        /// → instance.
        /// </para>
        /// <para>
        /// Инициализирует новый экземпляр класса <see
        /// → cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>A link.</para>
        /// <para>Связь.</para>
        /// </param>
        /// <param name="argumentName">
        /// <para>A argument name.</para>
        /// <para>Имя аргумента.</para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkDoesNotExistsException(TLinkAddress link, string argumentName) :
        → base(FormatMessage(link, argumentName), argumentName) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>
        /// → instance.
        /// </para>
        /// <para>
        /// Инициализирует новый экземпляр класса <see
        /// → cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>A link.</para>
        /// <para>Связь.</para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ArgumentLinkDoesNotExistsException(TLinkAddress link) : base(FormatMessage(link))
        → { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>
        /// → instance.
        /// </para>
        /// <para>
        /// Инициализирует новый экземпляр класса <see
        /// → cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>.
        /// </para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para>Сообщение.</para>
        /// </param>
        /// <param name="innerException">
        /// <para>A inner exception.</para>
        /// <para>Внутренняя ошибка.</para>
```

```csharp
 68          /// </param>
 69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
 70          public ArgumentLinkDoesNotExistsException(string message, Exception innerException) :
      ↪    base(message, innerException) { }

 72          /// <summary>
 73          /// <para>
 74          /// Initializes a new <see cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>
      ↪    instance.
 75          /// </para>
 76          /// <para>
 77          /// Инициализирует новый экземпляр класса <see
      ↪    cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>.
 78          /// </para>
 79          /// </summary>
 80          /// <param name="message">
 81          /// <para>A message.</para>
 82          /// <para>Сообщение.</para>
 83          /// </param>
 84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
 85          public ArgumentLinkDoesNotExistsException(string message) : base(message) { }

 87          /// <summary>
 88          /// <para>
 89          /// Initializes a new <see cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>
      ↪    instance.
 90          /// </para>
 91          /// <para>
 92          /// Инициализирует новый экземпляр класса <see
      ↪    cref="ArgumentLinkDoesNotExistsException{TLinkAddress}"/>.
 93          /// </para>
 94          /// </summary>
 95          [MethodImpl(MethodImplOptions.AggressiveInlining)]
 96          public ArgumentLinkDoesNotExistsException() { }
 97          [MethodImpl(MethodImplOptions.AggressiveInlining)]
 98          private static string FormatMessage(TLinkAddress link, string argumentName) => $"Связь
      ↪    [{link}] переданная в аргумент [{argumentName}] не существует.";
 99          [MethodImpl(MethodImplOptions.AggressiveInlining)]
100          private static string FormatMessage(TLinkAddress link) => $"Связь [{link}] переданная в
      ↪    качестве аргумента не существует.";
101      }
102  }
```

## 1.2  ./csharp/Platform.Data/Exceptions/ArgumentLinkHasDependenciesException.cs

```csharp
  1  using System;
  2  using System.Runtime.CompilerServices;
  3
  4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
  5
  6  namespace Platform.Data.Exceptions
  7  {
  8      /// <summary>
  9      /// <para>
 10      /// Represents the argument link has dependencies exception.
 11      /// </para>
 12      /// <para></para>
 13      /// </summary>
 14      /// <seealso cref="ArgumentException"/>
 15      public class ArgumentLinkHasDependenciesException<TLinkAddress> : ArgumentException
 16      {
 17          /// <summary>
 18          /// <para>
 19          /// Initializes a new <see cref="ArgumentLinkHasDependenciesException{TLinkAddress}"/>
          ↪    instance.
 20          /// </para>
 21          /// <para></para>
 22          /// </summary>
 23          /// <param name="link">
 24          /// <para>A link.</para>
 25          /// <para>Связь.</para>
 26          /// </param>
 27          /// <param name="paramName">
 28          /// <para>A param name.</para>
 29          /// <para>Имя параметра.</para>
 30          /// </param>
 31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
 32          public ArgumentLinkHasDependenciesException(TLinkAddress link, string paramName) :
          ↪    base(FormatMessage(link, paramName), paramName) { }
```

```csharp
33
34         /// <summary>
35         /// <para>
36         /// Initializes a new <see cref="ArgumentLinkHasDependenciesException{TLinkAddress}"/>
           ↪  instance.
37         /// </para>
38         /// <para></para>
39         /// </summary>
40         /// <param name="link">
41         /// <para>A link.</para>
42         /// <para></para>
43         /// </param>
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public ArgumentLinkHasDependenciesException(TLinkAddress link) :
           ↪  base(FormatMessage(link)) { }
46
47         /// <summary>
48         /// <para>
49         /// Initializes a new <see cref="ArgumentLinkHasDependenciesException{TLinkAddress}"/>
           ↪  instance.
50         /// </para>
51         /// <para></para>
52         /// </summary>
53         /// <param name="message">
54         /// <para>A message.</para>
55         /// <para></para>
56         /// </param>
57         /// <param name="innerException">
58         /// <para>A inner exception.</para>
59         /// <para></para>
60         /// </param>
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         public ArgumentLinkHasDependenciesException(string message, Exception innerException) :
           ↪  base(message, innerException) { }
63
64         /// <summary>
65         /// <para>
66         /// Initializes a new <see cref="ArgumentLinkHasDependenciesException{TLinkAddress}"/>
           ↪  instance.
67         /// </para>
68         /// <para></para>
69         /// </summary>
70         /// <param name="message">
71         /// <para>A message.</para>
72         /// <para></para>
73         /// </param>
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         public ArgumentLinkHasDependenciesException(string message) : base(message) { }
76
77         /// <summary>
78         /// <para>
79         /// Initializes a new <see cref="ArgumentLinkHasDependenciesException{TLinkAddress}"/>
           ↪  instance.
80         /// </para>
81         /// <para></para>
82         /// </summary>
83         [MethodImpl(MethodImplOptions.AggressiveInlining)]
84         public ArgumentLinkHasDependenciesException() { }
85         [MethodImpl(MethodImplOptions.AggressiveInlining)]
86         private static string FormatMessage(TLinkAddress link, string paramName) => $"У связи
           ↪  [{link}] переданной в аргумент [{paramName}] присутствуют зависимости, которые
           ↪  препятствуют изменению её внутренней структуры.";
87         [MethodImpl(MethodImplOptions.AggressiveInlining)]
88         private static string FormatMessage(TLinkAddress link) => $"У связи [{link}] переданной
           ↪  в качестве аргумента присутствуют зависимости, которые препятствуют изменению её
           ↪  внутренней структуры.";
89     }
90 }
```

## 1.3 ./csharp/Platform.Data/Exceptions/LinkWithSameValueAlreadyExistsException.cs

```csharp
1 using System;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Exceptions
7 {
8     /// <summary>
```

```csharp
     /// <para>
     /// Represents the link with same value already exists exception.
     /// </para>
     /// <para></para>
     /// </summary>
     /// <seealso cref="Exception"/>
     public class LinkWithSameValueAlreadyExistsException : Exception
     {
         /// <summary>
         /// <para>
         /// The default message.
         /// </para>
         /// <para></para>
         /// </summary>
         public static readonly string DefaultMessage = "Связь с таким же значением уже
         ↪   существует.";

         /// <summary>
         /// <para>
         /// Initializes a new <see cref="LinkWithSameValueAlreadyExistsException"/> instance.
         /// </para>
         /// <para></para>
         /// </summary>
         /// <param name="message">
         /// <para>A message.</para>
         /// <para></para>
         /// </param>
         /// <param name="innerException">
         /// <para>A inner exception.</para>
         /// <para></para>
         /// </param>
         [MethodImpl(MethodImplOptions.AggressiveInlining)]
         public LinkWithSameValueAlreadyExistsException(string message, Exception innerException)
         ↪   : base(message, innerException) { }

         /// <summary>
         /// <para>
         /// Initializes a new <see cref="LinkWithSameValueAlreadyExistsException"/> instance.
         /// </para>
         /// <para></para>
         /// </summary>
         /// <param name="message">
         /// <para>A message.</para>
         /// <para></para>
         /// </param>
         [MethodImpl(MethodImplOptions.AggressiveInlining)]
         public LinkWithSameValueAlreadyExistsException(string message) : base(message) { }

         /// <summary>
         /// <para>
         /// Initializes a new <see cref="LinkWithSameValueAlreadyExistsException"/> instance.
         /// </para>
         /// <para></para>
         /// </summary>
         [MethodImpl(MethodImplOptions.AggressiveInlining)]
         public LinkWithSameValueAlreadyExistsException() : base(DefaultMessage) { }
     }
}
```

## 1.4  ./csharp/Platform.Data/Exceptions/LinksLimitReachedException.cs

```csharp
using System;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Exceptions
{
    /// <summary>
    /// <para>
    /// Represents the links limit reached exception.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksLimitReachedExceptionBase"/>
    public class LinksLimitReachedException<TLinkAddress> : LinksLimitReachedExceptionBase
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksLimitReachedException{TLinkAddress}"/> instance.
```

```csharp
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="limit">
        /// <para>A limit.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksLimitReachedException(TLinkAddress limit) : this(FormatMessage(limit)) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksLimitReachedException{TLinkAddress}"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para></para>
        /// </param>
        /// <param name="innerException">
        /// <para>A inner exception.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksLimitReachedException(string message, Exception innerException) :
        ↪  base(message, innerException) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksLimitReachedException{TLinkAddress}"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksLimitReachedException(string message) : base(message) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksLimitReachedException{TLinkAddress}"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinksLimitReachedException() : base(DefaultMessage) { }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static string FormatMessage(TLinkAddress limit) => $"Достигнут лимит количества
        ↪  связей в хранилище ({limit}).";
    }
}
```

## 1.5 ./csharp/Platform.Data/Exceptions/LinksLimitReachedExceptionBase.cs

```csharp
using System;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data.Exceptions
{
    /// <summary>
    /// <para>
    /// Represents the links limit reached exception base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="Exception"/>
    public abstract class LinksLimitReachedExceptionBase : Exception
    {
        /// <summary>
        /// <para>
        /// The default message.
        /// </para>
        /// <para></para>
        /// </summary>
```

```csharp
        public static readonly string DefaultMessage = "Достигнут лимит количества связей в
        ↪   хранилище.";

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksLimitReachedExceptionBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para></para>
        /// </param>
        /// <param name="innerException">
        /// <para>A inner exception.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksLimitReachedExceptionBase(string message, Exception innerException) :
        ↪   base(message, innerException) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksLimitReachedExceptionBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="message">
        /// <para>A message.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksLimitReachedExceptionBase(string message) : base(message) { }
    }
}
```

## 1.6   ./csharp/Platform.Data/Hybrid.cs

```csharp
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data
{
    /// <summary>
    /// <para>
    /// The hybrid.
    /// </para>
    /// <para></para>
    /// </summary>
    public struct Hybrid<TLinkAddress> : IEquatable<Hybrid<TLinkAddress>>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪   EqualityComparer<TLinkAddress>.Default;
        private static readonly UncheckedSignExtendingConverter<TLinkAddress, long>
        ↪   _addressToInt64Converter = UncheckedSignExtendingConverter<TLinkAddress,
        ↪   long>.Default;
        private static readonly UncheckedConverter<long, TLinkAddress> _int64ToAddressConverter
        ↪   = UncheckedConverter<long, TLinkAddress>.Default;
        private static readonly UncheckedConverter<TLinkAddress, ulong>
        ↪   _addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
        private static readonly UncheckedConverter<ulong, TLinkAddress>
        ↪   _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
        private static readonly UncheckedConverter<object, long> _objectToInt64Converter =
        ↪   UncheckedConverter<object, long>.Default;

        /// <summary>
        /// <para>
        /// The max value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly ulong HalfOfNumberValuesRange =
        ↪   _addressToUInt64Converter.Convert(NumericType<TLinkAddress>.MaxValue) / 2;
```

```csharp
        /// <summary>
        /// <para>
        /// The half of number values range.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly TLinkAddress ExternalZero =
            _uInt64ToAddressConverter.Convert(HalfOfNumberValuesRange + 1UL);

        /// <summary>
        /// <para>
        /// The value.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly TLinkAddress Value;

        /// <summary>
        /// <para>
        /// Gets the is nothing value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool IsNothing
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) || SignedValue == 0;
        }

        /// <summary>
        /// <para>
        /// Gets the is internal value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool IsInternal
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => SignedValue > 0;
        }

        /// <summary>
        /// <para>
        /// Gets the is external value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool IsExternal
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) || SignedValue < 0;
        }

        /// <summary>
        /// <para>
        /// Gets the signed value value.
        /// </para>
        /// <para></para>
        /// </summary>
        public long SignedValue
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _addressToInt64Converter.Convert(Value);
        }

        /// <summary>
        /// <para>
        /// Gets the absolute value value.
        /// </para>
        /// <para></para>
        /// </summary>
        public long AbsoluteValue
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => _equalityComparer.Equals(Value, ExternalZero) ? 0 :
                Platform.Numbers.Math.Abs(SignedValue);
        }

        /// <summary>
```

```csharp
        /// <para>
        /// Initializes a new <see cref="Hybrid{TLinkAddress}"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>A value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Hybrid(TLinkAddress value)
        {
            Ensure.OnDebug.IsUnsignedInteger<TLinkAddress>();
            Value = value;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Hybrid{TLinkAddress}"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>A value.</para>
        /// <para></para>
        /// </param>
        /// <param name="isExternal">
        /// <para>A is external.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Hybrid(TLinkAddress value, bool isExternal)
        {
            if (_equalityComparer.Equals(value, default) && isExternal)
            {
                Value = ExternalZero;
            }
            else
            {
                if (isExternal)
                {
                    Value = Math<TLinkAddress>.Negate(value);
                }
                else
                {
                    Value = value;
                }
            }
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Hybrid{TLinkAddress}"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>A value.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Hybrid(object value) => Value =
          _int64ToAddressConverter.Convert(_objectToInt64Converter.Convert(value));

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Hybrid{TLinkAddress}"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>A value.</para>
        /// <para></para>
        /// </param>
        /// <param name="isExternal">
        /// <para>A is external.</para>
        /// <para></para>
        /// </param>
```

```csharp
189        [MethodImpl(MethodImplOptions.AggressiveInlining)]
190        public Hybrid(object value, bool isExternal)
191        {
192            var signedValue = value == null ? 0 : _objectToInt64Converter.Convert(value);
193            if (signedValue == 0 && isExternal)
194            {
195                Value = ExternalZero;
196            }
197            else
198            {
199                var absoluteValue = System.Math.Abs(signedValue);
200                Value = isExternal ? _int64ToAddressConverter.Convert(-absoluteValue) :
        ↪   _int64ToAddressConverter.Convert(absoluteValue);
201            }
202        }
203
204        [MethodImpl(MethodImplOptions.AggressiveInlining)]
205        public static implicit operator Hybrid<TLinkAddress>(TLinkAddress integer) => new
        ↪   Hybrid<TLinkAddress>(integer);
206
207        [MethodImpl(MethodImplOptions.AggressiveInlining)]
208        public static explicit operator Hybrid<TLinkAddress>(ulong integer) => new
        ↪   Hybrid<TLinkAddress>(integer);
209
210        [MethodImpl(MethodImplOptions.AggressiveInlining)]
211        public static explicit operator Hybrid<TLinkAddress>(long integer) => new
        ↪   Hybrid<TLinkAddress>(integer);
212
213        [MethodImpl(MethodImplOptions.AggressiveInlining)]
214        public static explicit operator Hybrid<TLinkAddress>(uint integer) => new
        ↪   Hybrid<TLinkAddress>(integer);
215
216        [MethodImpl(MethodImplOptions.AggressiveInlining)]
217        public static explicit operator Hybrid<TLinkAddress>(int integer) => new
        ↪   Hybrid<TLinkAddress>(integer);
218
219        [MethodImpl(MethodImplOptions.AggressiveInlining)]
220        public static explicit operator Hybrid<TLinkAddress>(ushort integer) => new
        ↪   Hybrid<TLinkAddress>(integer);
221
222        [MethodImpl(MethodImplOptions.AggressiveInlining)]
223        public static explicit operator Hybrid<TLinkAddress>(short integer) => new
        ↪   Hybrid<TLinkAddress>(integer);
224
225        [MethodImpl(MethodImplOptions.AggressiveInlining)]
226        public static explicit operator Hybrid<TLinkAddress>(byte integer) => new
        ↪   Hybrid<TLinkAddress>(integer);
227
228        [MethodImpl(MethodImplOptions.AggressiveInlining)]
229        public static explicit operator Hybrid<TLinkAddress>(sbyte integer) => new
        ↪   Hybrid<TLinkAddress>(integer);
230
231        [MethodImpl(MethodImplOptions.AggressiveInlining)]
232        public static implicit operator TLinkAddress(Hybrid<TLinkAddress> hybrid) =>
        ↪   hybrid.Value;
233
234        [MethodImpl(MethodImplOptions.AggressiveInlining)]
235        public static explicit operator ulong(Hybrid<TLinkAddress> hybrid) =>
        ↪   CheckedConverter<TLinkAddress, ulong>.Default.Convert(hybrid.Value);
236
237        [MethodImpl(MethodImplOptions.AggressiveInlining)]
238        public static explicit operator long(Hybrid<TLinkAddress> hybrid) =>
        ↪   hybrid.AbsoluteValue;
239
240        [MethodImpl(MethodImplOptions.AggressiveInlining)]
241        public static explicit operator uint(Hybrid<TLinkAddress> hybrid) =>
        ↪   CheckedConverter<TLinkAddress, uint>.Default.Convert(hybrid.Value);
242
243        [MethodImpl(MethodImplOptions.AggressiveInlining)]
244        public static explicit operator int(Hybrid<TLinkAddress> hybrid) =>
        ↪   (int)hybrid.AbsoluteValue;
245
246        [MethodImpl(MethodImplOptions.AggressiveInlining)]
247        public static explicit operator ushort(Hybrid<TLinkAddress> hybrid) =>
        ↪   CheckedConverter<TLinkAddress, ushort>.Default.Convert(hybrid.Value);
248
249        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
250         public static explicit operator short(Hybrid<TLinkAddress> hybrid) =>
        ↪   (short)hybrid.AbsoluteValue;

252         [MethodImpl(MethodImplOptions.AggressiveInlining)]
253         public static explicit operator byte(Hybrid<TLinkAddress> hybrid) =>
        ↪   CheckedConverter<TLinkAddress, byte>.Default.Convert(hybrid.Value);

255         [MethodImpl(MethodImplOptions.AggressiveInlining)]
256         public static explicit operator sbyte(Hybrid<TLinkAddress> hybrid) =>
        ↪   (sbyte)hybrid.AbsoluteValue;

258         /// <summary>
259         /// <para>
260         /// Returns the string.
261         /// </para>
262         /// <para></para>
263         /// </summary>
264         /// <returns>
265         /// <para>The string</para>
266         /// <para></para>
267         /// </returns>
268         [MethodImpl(MethodImplOptions.AggressiveInlining)]
269         public override string ToString() => IsExternal ? $"<{AbsoluteValue}>" :
        ↪   Value.ToString();

271         /// <summary>
272         /// <para>
273         /// Determines whether this instance equals.
274         /// </para>
275         /// <para></para>
276         /// </summary>
277         /// <param name="other">
278         /// <para>The other.</para>
279         /// <para></para>
280         /// </param>
281         /// <returns>
282         /// <para>The bool</para>
283         /// <para></para>
284         /// </returns>
285         [MethodImpl(MethodImplOptions.AggressiveInlining)]
286         public bool Equals(Hybrid<TLinkAddress> other) => _equalityComparer.Equals(Value,
        ↪   other.Value);

288         /// <summary>
289         /// <para>
290         /// Determines whether this instance equals.
291         /// </para>
292         /// <para></para>
293         /// </summary>
294         /// <param name="obj">
295         /// <para>The obj.</para>
296         /// <para></para>
297         /// </param>
298         /// <returns>
299         /// <para>The bool</para>
300         /// <para></para>
301         /// </returns>
302         [MethodImpl(MethodImplOptions.AggressiveInlining)]
303         public override bool Equals(object obj) => obj is Hybrid<TLinkAddress> hybrid ?
        ↪   Equals(hybrid) : false;

305         /// <summary>
306         /// <para>
307         /// Gets the hash code.
308         /// </para>
309         /// <para></para>
310         /// </summary>
311         /// <returns>
312         /// <para>The int</para>
313         /// <para></para>
314         /// </returns>
315         [MethodImpl(MethodImplOptions.AggressiveInlining)]
316         public override int GetHashCode() => Value.GetHashCode();

318         [MethodImpl(MethodImplOptions.AggressiveInlining)]
319         public static bool operator ==(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
        ↪   left.Equals(right);

320
```

```csharp
321         [MethodImpl(MethodImplOptions.AggressiveInlining)]
322         public static bool operator !=(Hybrid<TLinkAddress> left, Hybrid<TLinkAddress> right) =>
            ↪  !(left == right);
323     }
324 }
```

## 1.7 ./csharp/Platform.Data/ILinks.cs

```csharp
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Delegates;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data
9  {
10     /// <summary>
11     /// <para>Represents an interface for manipulating data in the Links (links storage)
        ↪  format.</para>
12     /// <para>Представляет интерфейс для манипуляции с данными в формате Links (хранилища
        ↪  связей).</para>
13     /// </summary>
14     /// <remarks>
15     /// <para>This interface is independent of the size of the content of the link, meaning it
        ↪  is suitable for both doublets, triplets, and link sequences of any size.</para>
16     /// <para>Этот интерфейс не зависит от размера содержимого связи, а значит подходит как для
        ↪  дуплетов, триплетов и последовательностей связей любого размера.</para>
17     /// </remarks>
18     public interface ILinks<TLinkAddress, TConstants>
19         where TConstants : LinksConstants<TLinkAddress>
20     {
21         #region Constants
22
23         /// <summary>
24         /// <para>Returns the set of constants that is necessary for effective communication
            ↪  with the methods of this interface.</para>
25         /// <para>Возвращает набор констант, который необходим для эффективной коммуникации с
            ↪  методами этого интерфейса.</para>
26         /// </summary>
27         /// <remarks>
28         /// <para>These constants are not changed since the creation of the links storage access
            ↪  point.</para>
29         /// <para>Эти константы не меняются с момента создания точки доступа к хранилищу
            ↪  связей.</para>
30         /// </remarks>
31         TConstants Constants
32         {
33             [MethodImpl(MethodImplOptions.AggressiveInlining)]
34             get;
35         }
36
37         #endregion
38
39         #region Read
40
41         /// <summary>
42         /// <para>Counts and returns the total number of links in the storage that meet the
            ↪  specified restriction.</para>
43         /// <para>Подсчитывает и возвращает общее число связей находящихся в хранилище,
            ↪  соответствующих указанному ограничению.</para>
44         /// </summary>
45         /// <param name="restriction"><para>Restriction on the contents of
            ↪  links.</para><para>Ограничение на содержимое связей.</para></param>
46         /// <returns><para>The total number of links in the storage that meet the specified
            ↪  restriction.</para><para>Общее число связей находящихся в хранилище, соответствующих
            ↪  указанному ограничению.</para></returns>
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         TLinkAddress Count(IList<TLinkAddress>? restriction);
49
50         /// <summary>
51         /// <para>Passes through all the links matching the pattern, invoking a handler for each
            ↪  matching link.</para>
52         /// <para>Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
            ↪  (handler) для каждой подходящей связи.</para>
53         /// </summary>
54         /// <param name="restriction">
```

```csharp
        /// <para>Restriction on the contents of links. Each constraint can have values:
        ↪  Constants.Null - the 0th link denoting a reference to the void, Any - the absence of
        ↪  a constraint, 1..∞ a specific link index.</para>
        /// <para>Ограничение на содержимое связей. Каждое ограничение может иметь значения:
        ↪  Constants.Null - 0-я связь, обозначающая ссылку на пустоту, Any - отсутствие
        ↪  ограничения, 1..∞ конкретный индекс связи.</para>
        /// </param>
        /// <param name="handler"><para>A handler for each matching link.</para><para>Обработчик
        ↪  для каждой подходящей связи.</para></param>
        /// <returns><para>Constants.Continue, if the pass through the links was not
        ↪  interrupted, and Constants.Break otherwise.</para><para>Constants.Continue, в случае
        ↪  если проход по связям не был прерван и Constants.Break в обратном
        ↪  случае.</para></returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TLinkAddress Each(IList<TLinkAddress>? restriction, ReadHandler<TLinkAddress>? handler);

        #endregion

        #region Write

        /// <summary>
        /// <para>Creates a link.</para>
        /// <para>Создаёт связь.</para>
        /// <param name="substitution">
        /// <para>The content of a new link. This argument is optional, if the null passed as
        ↪  value that means no content of a link is set.</para>
        /// <para>Содержимое новой связи. Этот аргумент опционален, если null передан в качестве
        ↪  значения это означает, что никакого содержимого для связи не установлено.</para>
        /// </param>
        /// <param name="handler">
        /// <para>A function to handle each executed change. This function can use
        ↪  Constants.Continue to continue proccess each change. Constants.Break can be used to
        ↪  stop receiving of executed changes.</para>
        /// <para>Функция для обработки каждого выполненного изменения. Эта функция может
        ↪  использовать Constants.Continue чтобы продолжить обрабатывать каждое изменение.
        ↪  Constants.Break может быть использована для остановки получения выполненных
        ↪  изменений.</para>
        /// </param>
        /// </summary>
        /// <returns>
        /// <para>
        /// Constants.Continue if all executed changes are handled.
        /// Constants.Break if proccessing of handled changes is stoped.
        /// </para>
        /// <para>
        /// Constants.Continue если все выполненные изменения обработаны.
        /// Constants.Break если обработака выполненных изменений остановлена.
        /// </para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        TLinkAddress Create(IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>?
        ↪  handler);

        /// <summary>
        /// Обновляет связь с указанными restriction[Constants.IndexPart] в адресом связи
        /// на связь с указанным новым содержимым.
        /// </summary>
        /// <param name="restriction">
        /// Ограничение на содержимое связей.
        /// Предполагается, что будет указан индекс связи (в restriction[Constants.IndexPart]) и
        ↪  далее за ним будет следовать содержимое связи.
        /// Каждое ограничение может иметь значения: Constants.Null - 0-я связь, обозначающая
        ↪  ссылку на пустоту,
        /// Constants.Itself - требование установить ссылку на себя, 1..∞ конкретный индекс
        ↪  другой связи.
        /// </param>
        /// <param name="substitution"></param>
        /// <param name="handler">
        /// <para>A function to handle each executed change. This function can use
        ↪  Constants.Continue to continue proccess each change. Constants.Break can be used to
        ↪  stop receiving of executed changes.</para>
        /// <para>Функция для обработки каждого выполненного изменения. Эта функция может
        ↪  использовать Constants.Continue чтобы продолжить обрабатывать каждое изменение.
        ↪  Constants.Break может быть использована для остановки получения выполненных
        ↪  изменений.</para>
        /// </param>
        /// <returns>
```

```csharp
108          /// <para>
109          /// Constants.Continue if all executed changes are handled.
110          /// Constants.Break if proccessing of handled changes is stoped.
111          /// </para>
112          /// <para>
113          /// Constants.Continue если все выполненные изменения обработаны.
114          /// Constants.Break если обработака выполненных изменений остановлена.
115          /// </para>
116          /// </returns>
117          [MethodImpl(MethodImplOptions.AggressiveInlining)]
118          TLinkAddress Update(IList<TLinkAddress>? restriction, IList<TLinkAddress>? substitution,
    ↪     WriteHandler<TLinkAddress>? handler);
119
120          /// <summary>
121          /// <para>Deletes links that match the specified restriction.</para>
122          /// <para>Удаляет связи соответствующие указанному ограничению.</para>
123          /// </summary>
124          /// <param name="restriction">
125          /// <para>Restriction on the content of a link. This argument is optional, if the null
    ↪     passed as value that means no restriction on the content of a link are set.</para>
126          /// <para>Ограничение на содержимое связи. Этот аргумент опционален, если null передан в
    ↪     качестве значения это означает, что никаких ограничений на содержимое связи не
    ↪     установлено.</para>
127          /// </param>
128          /// <param name="handler">
129          /// <para>A function to handle each executed change. This function can use
    ↪     Constants.Continue to continue proccess each change. Constants.Break can be used to
    ↪     stop receiving of executed changes.</para>
130          /// <para>Функция для обработки каждого выполненного изменения. Эта функция может
    ↪     использовать Constants.Continue чтобы продолжить обрабатывать каждое изменение.
    ↪     Constants.Break может быть использована для остановки получения выполненных
    ↪     изменений.</para>
131          /// </param>
132          /// <returns>
133          /// <para>
134          /// Constants.Continue if all executed changes are handled.
135          /// Constants.Break if proccessing of handled changes is stoped.
136          /// </para>
137          /// <para>
138          /// Constants.Continue если все выполненные изменения обработаны.
139          /// Constants.Break если обработака выполненных изменений остановлена.
140          /// </para>
141          /// </returns>
142          [MethodImpl(MethodImplOptions.AggressiveInlining)]
143          TLinkAddress Delete(IList<TLinkAddress>? restriction, WriteHandler<TLinkAddress>?
    ↪     handler);
144
145          #endregion
146      }
147  }
```

## 1.8 ./csharp/Platform.Data/ILinksExtensions.cs

```csharp
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Setters;
5  using Platform.Data.Exceptions;
6  using Platform.Delegates;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data
11 {
12     /// <summary>
13     /// <para>
14     /// Represents the links extensions.
15     /// </para>
16     /// <para></para>
17     /// </summary>
18     public static class ILinksExtensions
19     {
20         public static TLinkAddress Create<TLinkAddress>(this ILinks<TLinkAddress,
    ↪     LinksConstants<TLinkAddress>> links) => links.Create(null);
21
22         public static TLinkAddress Create<TLinkAddress>(this ILinks<TLinkAddress,
    ↪     LinksConstants<TLinkAddress>> links, IList<TLinkAddress>? substitution)
23         {
24             var constants = links.Constants;
```

```csharp
                Setter<TLinkAddress, TLinkAddress> setter = new Setter<TLinkAddress,
25                  TLinkAddress>(constants.Continue, constants.Break, constants.Null);
                links.Create(substitution, setter.SetFirstFromSecondListAndReturnTrue);
26
                return setter.Result;
27
            }
28

29
        public static TLinkAddress Update<TLinkAddress>(this ILinks<TLinkAddress,
30              LinksConstants<TLinkAddress>> links, IList<TLinkAddress>? restriction,
                IList<TLinkAddress>? substitution)
            {
31
                var constants = links.Constants;
32
                Setter<TLinkAddress, TLinkAddress> setter = new(constants.Continue, constants.Break,
33                  constants.Null);
                links.Update(restriction, substitution, setter.SetFirstFromSecondListAndReturnTrue);
34
                return setter.Result;
35
            }
36

37
        public static TLinkAddress Delete<TLinkAddress>(this ILinks<TLinkAddress,
38              LinksConstants<TLinkAddress>> links, TLinkAddress linkToDelete) => Delete(links,
                (IList<TLinkAddress>?)new LinkAddress<TLinkAddress>(linkToDelete));

39
        public static TLinkAddress Delete<TLinkAddress>(this ILinks<TLinkAddress,
40              LinksConstants<TLinkAddress>> links, IList<TLinkAddress>? restriction)
            {
41
                var constants = links.Constants;
42
                Setter<TLinkAddress, TLinkAddress> setter = new Setter<TLinkAddress,
43                  TLinkAddress>(constants.Continue, constants.Break, constants.Null);
                links.Delete(restriction, setter.SetFirstFromFirstListAndReturnTrue);
44
                return setter.Result;
45
            }
46

47
        /// <summary>
48
        /// <para>
49
        /// Counts the links.
50
        /// </para>
51
        /// <para></para>
52
        /// </summary>
53
        /// <typeparam name="TLinkAddress">
54
        /// <para>The link address.</para>
55
        /// <para></para>
56
        /// </typeparam>
57
        /// <typeparam name="TConstants">
58
        /// <para>The constants.</para>
59
        /// <para></para>
60
        /// </typeparam>
61
        /// <param name="links">
62
        /// <para>The links.</para>
63
        /// <para></para>
64
        /// </param>
65
        /// <param name="restrictions">
66
        /// <para>The restrictions.</para>
67
        /// <para></para>
68
        /// </param>
69
        /// <returns>
70
        /// <para>The link address</para>
71
        /// <para></para>
72
        /// </returns>
73
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
74
        public static TLinkAddress Count<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
75              TConstants> links, params TLinkAddress[] restrictions)
                where TConstants : LinksConstants<TLinkAddress>
76
                => links.Count(restrictions);
77

78
        /// <summary>
79
        /// Возвращает значение, определяющее существует ли связь с указанным индексом в
80              хранилище связей.
        /// </summary>
81
        /// <param name="links">Хранилище связей.</param>
82
        /// <param name="link">Индекс проверяемой на существование связи.</param>
83
        /// <returns>Значение, определяющее существует ли связь.</returns>
84
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
85
        public static bool Exists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
86              TConstants> links, TLinkAddress link)
                where TConstants : LinksConstants<TLinkAddress>
87
            {
88
                var constants = links.Constants;
89
```

```csharp
 90             return constants.IsExternalReference(link) || (constants.IsInternalReference(link)
                ↪  && Comparer<TLinkAddress>.Default.Compare(links.Count(new
                ↪  LinkAddress<TLinkAddress>(link)), default) > 0);
 91         }
 92
 93         /// <param name="links">Хранилище связей.</param>
 94         /// <param name="link">Индекс проверяемой на существование связи.</param>
 95         /// <remarks>
 96         /// TODO: May be move to EnsureExtensions or make it both there and here
 97         /// </remarks>
 98         [MethodImpl(MethodImplOptions.AggressiveInlining)]
 99         public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
                ↪  TConstants> links, TLinkAddress link)
100             where TConstants : LinksConstants<TLinkAddress>
101         {
102             if (!links.Exists(link))
103             {
104                 throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link);
105             }
106         }
107
108         /// <param name="links">Хранилище связей.</param>
109         /// <param name="link">Индекс проверяемой на существование связи.</param>
110         /// <param name="argumentName">Имя аргумента, в который передаётся индекс связи.</param>
111         [MethodImpl(MethodImplOptions.AggressiveInlining)]
112         public static void EnsureLinkExists<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
                ↪  TConstants> links, TLinkAddress link, string argumentName)
113             where TConstants : LinksConstants<TLinkAddress>
114         {
115             if (!links.Exists(link))
116             {
117                 throw new ArgumentLinkDoesNotExistsException<TLinkAddress>(link, argumentName);
118             }
119         }
120
121         /// <summary>
122         /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
                ↪  (handler) для каждой подходящей связи.
123         /// </summary>
124         /// <param name="links">Хранилище связей.</param>
125         /// <param name="handler">Обработчик каждой подходящей связи.</param>
126         /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
                ↪  может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
                ↪  Any - отсутствие ограничения, 1..∞ конкретный индекс связи.</param>
127         /// <returns>True, в случае если проход по связям не был прерван и False в обратном
                ↪  случае.</returns>
128         [MethodImpl(MethodImplOptions.AggressiveInlining)]
129         public static TLinkAddress Each<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
                ↪  TConstants> links, ReadHandler<TLinkAddress>? handler, params TLinkAddress[]
                ↪  restrictions)
130             where TConstants : LinksConstants<TLinkAddress>
131             => links.Each(restrictions, handler);
132
133         /// <summary>
134         /// Возвращает части-значения для связи с указанным индексом.
135         /// </summary>
136         /// <param name="links">Хранилище связей.</param>
137         /// <param name="link">Индекс связи.</param>
138         /// <returns>Уникальную связь.</returns>
139         [MethodImpl(MethodImplOptions.AggressiveInlining)]
140         public static IList<TLinkAddress>? GetLink<TLinkAddress, TConstants>(this
                ↪  ILinks<TLinkAddress, TConstants> links, TLinkAddress link)
141             where TConstants : LinksConstants<TLinkAddress>
142         {
143             var constants = links.Constants;
144             if (constants.IsExternalReference(link))
145             {
146                 return new Point<TLinkAddress>(link, constants.TargetPart + 1);
147             }
148             var linkPartsSetter = new Setter<IList<TLinkAddress>?,
                ↪  TLinkAddress>(constants.Continue, constants.Break);
149             links.Each(linkPartsSetter.SetAndReturnTrue, link);
150             return linkPartsSetter.Result;
151         }
152
153         #region Points
154
```

```csharp
155          /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
             ↪  точкой полностью (связью замкнутой на себе дважды).</summary>
156          /// <param name="links">Хранилище связей.</param>
157          /// <param name="link">Индекс проверяемой связи.</param>
158          /// <returns>Значение, определяющее является ли связь точкой полностью.</returns>
159          /// <remarks>
160          /// Связь точка - это связь, у которой начало (Source) и конец (Target) есть сама эта
             ↪  связь.
161          /// Но что, если точка уже есть, а нужно создать пару с таким же значением? Должны ли
             ↪  точка и пара существовать одновременно?
162          /// Или в качестве решения для точек нужно использовать 0 в качестве начала и конца, а
             ↪  сортировать по индексу в массиве связей?
163          /// Какое тогда будет значение Source и Target у точки? 0 или её индекс?
164          /// Или точка должна быть одновременно точкой и парой, а также последовательностями из
             ↪  самой себя любого размера?
165          /// Как только есть ссылка на себя, появляется этот парадокс, причём достаточно даже
             ↪  одной ссылки на себя (частичной точки).
166          /// А что если не выбирать что является точкой, пара нулей (цикл через пустоту) или
167          /// самостоятельный цикл через себя? Что если предоставить все варианты использования
             ↪  связей?
168          /// Что если разрешить и нули, а так же частичные варианты?
169          ///
170          /// Что если точка, это только в том случае когда link.Source == link &amp;&amp;
             ↪  link.Target == link , т.е. дважды ссылка на себя.
171          /// А пара это тогда, когда link.Source == link.Target &amp;&amp; link.Source != link ,
             ↪  т.е. ссылка не на себя а во вне.
172          ///
173          /// Тогда если у нас уже создана пара, но нам нужна точка, мы можем используя
             ↪  промежуточную связь,
174          /// например "DoubletOf" обозначить что является точно парой, а что точно точкой.
175          /// И наоборот этот же метод поможет, если уже существует точка, но нам нужна пара.
176          /// </remarks>
177          [MethodImpl(MethodImplOptions.AggressiveInlining)]
178          public static bool IsFullPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
             ↪  TConstants> links, TLinkAddress link)
179              where TConstants : LinksConstants<TLinkAddress>
180          {
181              if (links.Constants.IsExternalReference(link))
182              {
183                  return true;
184              }
185              links.EnsureLinkExists(link);
186              return Point<TLinkAddress>.IsFullPoint(links.GetLink(link));
187          }
188
189          /// <summary>Возвращает значение, определяющее является ли связь с указанным индексом
             ↪  точкой частично (связью замкнутой на себе как минимум один раз).</summary>
190          /// <param name="links">Хранилище связей.</param>
191          /// <param name="link">Индекс проверяемой связи.</param>
192          /// <returns>Значение, определяющее является ли связь точкой частично.</returns>
193          /// <remarks>
194          /// Достаточно любой одной ссылки на себя.
195          /// Также в будущем можно будет проверять и всех родителей, чтобы проверить есть ли
             ↪  ссылки на себя (на эту связь).
196          /// </remarks>
197          [MethodImpl(MethodImplOptions.AggressiveInlining)]
198          public static bool IsPartialPoint<TLinkAddress, TConstants>(this ILinks<TLinkAddress,
             ↪  TConstants> links, TLinkAddress link)
199              where TConstants : LinksConstants<TLinkAddress>
200          {
201              if (links.Constants.IsExternalReference(link))
202              {
203                  return true;
204              }
205              links.EnsureLinkExists(link);
206              return Point<TLinkAddress>.IsPartialPoint(links.GetLink(link));
207          }
208
209          #endregion
210      }
211  }
```

## 1.9   ./csharp/Platform.Data/ISynchronizedLinks.cs

```csharp
1  using Platform.Threading.Synchronization;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data
```

```csharp
 6   {
 7       /// <summary>
 8       /// <para>
 9       /// Defines the synchronized links.
10       /// </para>
11       /// <para></para>
12       /// </summary>
13       /// <seealso cref="ISynchronized{TLinks}"/>
14       /// <seealso cref="ILinks{TLinkAddress, TConstants}"/>
15       public interface ISynchronizedLinks<TLinkAddress, TLinks, TConstants> :
       ↪   ISynchronized<TLinks>, ILinks<TLinkAddress, TConstants>
16           where TLinks : ILinks<TLinkAddress, TConstants>
17           where TConstants : LinksConstants<TLinkAddress>
18       {
19       }
20   }
```

## 1.10 ./csharp/Platform.Data/LinkAddress.cs

```csharp
 1   using System;
 2   using System.Collections;
 3   using System.Collections.Generic;
 4   using System.Runtime.CompilerServices;
 5
 6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 7
 8   namespace Platform.Data
 9   {
10       /// <summary>
11       /// <para>
12       /// Represents the link address.
13       /// </para>
14       /// <para></para>
15       /// </summary>
16       /// <seealso cref="IEquatable{LinkAddress{TLinkAddress}}"/>
17       /// <seealso cref="IList{TLinkAddress}"/>
18       public class LinkAddress<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>,
       ↪   IList<TLinkAddress>
19       {
20           private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
       ↪   EqualityComparer<TLinkAddress>.Default;
21
22           /// <summary>
23           /// <para>
24           /// Gets the index value.
25           /// </para>
26           /// <para></para>
27           /// </summary>
28           public TLinkAddress Index
29           {
30               [MethodImpl(MethodImplOptions.AggressiveInlining)]
31               get;
32           }
33
34           /// <summary>
35           /// <para>
36           /// The not supported exception.
37           /// </para>
38           /// <para></para>
39           /// </summary>
40           public TLinkAddress this[int index]
41           {
42               [MethodImpl(MethodImplOptions.AggressiveInlining)]
43               get
44               {
45                   if (index == 0)
46                   {
47                       return Index;
48                   }
49                   else
50                   {
51                       throw new IndexOutOfRangeException();
52                   }
53               }
54               [MethodImpl(MethodImplOptions.AggressiveInlining)]
55               set => throw new NotSupportedException();
56           }
57
58           /// <summary>
59           /// <para>
```

```csharp
        /// Gets the count value.
        /// </para>
        /// <para></para>
        /// </summary>
        public int Count
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => 1;
        }

        /// <summary>
        /// <para>
        /// Gets the is read only value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool IsReadOnly
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => true;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinkAddress{TLinkAddress}"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>A index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkAddress(TLinkAddress index) => Index = index;

        /// <summary>
        /// <para>
        /// Adds the item.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="item">
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TLinkAddress item) => throw new NotSupportedException();

        /// <summary>
        /// <para>
        /// Clears this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Clear() => throw new NotSupportedException();

        /// <summary>
        /// <para>
        /// Determines whether this instance contains.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="item">
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index);

        /// <summary>
        /// <para>
        /// Copies the to using the specified array.
        /// </para>
```

```csharp
138         /// <para></para>
139         /// </summary>
140         /// <param name="array">
141         /// <para>The array.</para>
142         /// <para></para>
143         /// </param>
144         /// <param name="arrayIndex">
145         /// <para>The array index.</para>
146         /// <para></para>
147         /// </param>
148         [MethodImpl(MethodImplOptions.AggressiveInlining)]
149         public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;
150
151         /// <summary>
152         /// <para>
153         /// Gets the enumerator.
154         /// </para>
155         /// <para></para>
156         /// </summary>
157         /// <returns>
158         /// <para>An enumerator of t link address</para>
159         /// <para></para>
160         /// </returns>
161         [MethodImpl(MethodImplOptions.AggressiveInlining)]
162         public IEnumerator<TLinkAddress> GetEnumerator()
163         {
164             yield return Index;
165         }
166
167         /// <summary>
168         /// <para>
169         /// Indexes the of using the specified item.
170         /// </para>
171         /// <para></para>
172         /// </summary>
173         /// <param name="item">
174         /// <para>The item.</para>
175         /// <para></para>
176         /// </param>
177         /// <returns>
178         /// <para>The int</para>
179         /// <para></para>
180         /// </returns>
181         [MethodImpl(MethodImplOptions.AggressiveInlining)]
182         public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
    ↪  0 : -1;
183
184         /// <summary>
185         /// <para>
186         /// Inserts the index.
187         /// </para>
188         /// <para></para>
189         /// </summary>
190         /// <param name="index">
191         /// <para>The index.</para>
192         /// <para></para>
193         /// </param>
194         /// <param name="item">
195         /// <para>The item.</para>
196         /// <para></para>
197         /// </param>
198         [MethodImpl(MethodImplOptions.AggressiveInlining)]
199         public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();
200
201         /// <summary>
202         /// <para>
203         /// Determines whether this instance remove.
204         /// </para>
205         /// <para></para>
206         /// </summary>
207         /// <param name="item">
208         /// <para>The item.</para>
209         /// <para></para>
210         /// </param>
211         /// <returns>
212         /// <para>The bool</para>
213         /// <para></para>
214         /// </returns>
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Remove(TLinkAddress item) => throw new NotSupportedException();

        /// <summary>
        /// <para>
        /// Removes the at using the specified index.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>The index.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void RemoveAt(int index) => throw new NotSupportedException();

        /// <summary>
        /// <para>
        /// Gets the enumerator.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The enumerator</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        IEnumerator IEnumerable.GetEnumerator()
        {
            yield return Index;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="other">
        /// <para>The other.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Equals(LinkAddress<TLinkAddress> other) => other != null &&
        ↪  _equalityComparer.Equals(Index, other.Index);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator TLinkAddress(LinkAddress<TLinkAddress> linkAddress) =>
        ↪  linkAddress.Index;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator LinkAddress<TLinkAddress>(TLinkAddress linkAddress) =>
        ↪  new LinkAddress<TLinkAddress>(linkAddress);

        /// <summary>
        /// <para>
        /// Determines whether this instance equals.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="obj">
        /// <para>The obj.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override bool Equals(object obj) => obj is LinkAddress<TLinkAddress> linkAddress
        ↪  ? Equals(linkAddress) : false;

        /// <summary>
        /// <para>
```

```csharp
        /// Gets the hash code.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override int GetHashCode() => Index.GetHashCode();

        /// <summary>
        /// <para>
        /// Returns the string.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => Index.ToString();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator ==(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
        ↪  right)
        {
            if (left == null && right == null)
            {
                return true;
            }
            if (left == null)
            {
                return false;
            }
            return left.Equals(right);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool operator !=(LinkAddress<TLinkAddress> left, LinkAddress<TLinkAddress>
        ↪  right) => !(left == right);
    }
}
```

## 1.11 ./csharp/Platform.Data/LinksConstants.cs

```csharp
using System.Runtime.CompilerServices;
using Platform.Ranges;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Data
{
    /// <summary>
    /// <para>
    /// Represents the links constants.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksConstantsBase"/>
    public class LinksConstants<TLinkAddress> : LinksConstantsBase
    {
        private static readonly TLinkAddress _one = Arithmetic<TLinkAddress>.Increment(default);
        private static readonly UncheckedConverter<ulong, TLinkAddress>
        ↪  _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;

        #region Link parts

        /// <summary>Возвращает индекс части, которая отвечает за индекс (адрес, идентификатор)
        ↪  самой связи.</summary>
        public int IndexPart
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

```

```csharp
        /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-начало (первая
        ↪ часть-значение).</summary>
        public int SourcePart
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>Возвращает индекс части, которая отвечает за ссылку на связь-конец
        ↪ (последняя часть-значение).</summary>
        public int TargetPart
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        #endregion

        #region Flow control

        /// <summary>Возвращает значение, обозначающее продолжение прохода по связям.</summary>
        /// <remarks>Используется в функции обработчике, который передаётся в функцию
        ↪ Each.</remarks>
        public TLinkAddress Continue
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>Возвращает значение, обозначающее остановку прохода по связям.</summary>
        /// <remarks>Используется в функции обработчике, который передаётся в функцию
        ↪ Each.</remarks>
        public TLinkAddress Break
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>Возвращает значение, обозначающее пропуск в проходе по связям.</summary>
        public TLinkAddress Skip
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        #endregion

        #region Special symbols

        /// <summary>Возвращает значение, обозначающее отсутствие связи.</summary>
        public TLinkAddress Null
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>Возвращает значение, обозначающее любую связь.</summary>
        /// <remarks>Возможно нужно зарезервировать отдельное значение, тогда можно будет
        ↪ создавать все варианты последовательностей в функции Create.</remarks>
        public TLinkAddress Any
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>Возвращает значение, обозначающее связь-ссылку на саму связь.</summary>
        public TLinkAddress Itself
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public TLinkAddress Error { get; }

        #endregion

        #region References

        /// <summary>Возвращает диапазон возможных индексов для внутренних связей (внутренних
        ↪ ссылок).</summary>
```

```csharp
106        public Range<TLinkAddress> InternalReferencesRange
107        {
108            [MethodImpl(MethodImplOptions.AggressiveInlining)]
109            get;
110        }
111
112        /// <summary>Возвращает диапазон возможных индексов для внешних связей (внешних
    ↪   ссылок).</summary>
113        public Range<TLinkAddress>? ExternalReferencesRange
114        {
115            [MethodImpl(MethodImplOptions.AggressiveInlining)]
116            get;
117        }
118
119        #endregion
120
121        /// <summary>
122        /// <para>
123        /// Initializes a new <see cref="LinksConstants{TLinkAddress}"/> instance.
124        /// </para>
125        /// <para></para>
126        /// </summary>
127        /// <param name="targetPart">
128        /// <para>A target part.</para>
129        /// <para></para>
130        /// </param>
131        /// <param name="possibleInternalReferencesRange">
132        /// <para>A possible internal references range.</para>
133        /// <para></para>
134        /// </param>
135        /// <param name="possibleExternalReferencesRange">
136        /// <para>A possible external references range.</para>
137        /// <para></para>
138        /// </param>
139        [MethodImpl(MethodImplOptions.AggressiveInlining)]
140        public LinksConstants(int targetPart, Range<TLinkAddress>
    ↪   possibleInternalReferencesRange, Range<TLinkAddress>?
    ↪   possibleExternalReferencesRange)
141        {
142            IndexPart = 0;
143            SourcePart = 1;
144            TargetPart = targetPart;
145            var currentInternalReferenceIndex = possibleInternalReferencesRange.Maximum;
146            Null = default;
147            Continue = currentInternalReferenceIndex;
148            Break = Arithmetic.Decrement(ref currentInternalReferenceIndex);;
149            Skip = Arithmetic.Decrement(ref currentInternalReferenceIndex);
150            Any = Arithmetic.Decrement(ref currentInternalReferenceIndex);
151            Itself = Arithmetic.Decrement(ref currentInternalReferenceIndex);
152            Error = Arithmetic.Decrement(ref currentInternalReferenceIndex);
153            Arithmetic.Decrement(ref currentInternalReferenceIndex);
154            InternalReferencesRange = (possibleInternalReferencesRange.Minimum,
    ↪   currentInternalReferenceIndex);
155            ExternalReferencesRange = possibleExternalReferencesRange;
156        }
157
158        /// <summary>
159        /// <para>
160        /// Initializes a new <see cref="LinksConstants{TLinkAddress}"/> instance.
161        /// </para>
162        /// <para></para>
163        /// </summary>
164        /// <param name="targetPart">
165        /// <para>A target part.</para>
166        /// <para></para>
167        /// </param>
168        /// <param name="enableExternalReferencesSupport">
169        /// <para>A enable external references support.</para>
170        /// <para></para>
171        /// </param>
172        [MethodImpl(MethodImplOptions.AggressiveInlining)]
173        public LinksConstants(int targetPart, bool enableExternalReferencesSupport) :
    ↪   this(targetPart, GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
    ↪   GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }
174
175        /// <summary>
176        /// <para>
177        /// Initializes a new <see cref="LinksConstants{TLinkAddress}"/> instance.
178        /// </para>
```

```
179        /// <para></para>
180        /// </summary>
181        /// <param name="possibleInternalReferencesRange">
182        /// <para>A possible internal references range.</para>
183        /// <para></para>
184        /// </param>
185        /// <param name="possibleExternalReferencesRange">
186        /// <para>A possible external references range.</para>
187        /// <para></para>
188        /// </param>
189        [MethodImpl(MethodImplOptions.AggressiveInlining)]
190        public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange,
           ↪ Range<TLinkAddress>? possibleExternalReferencesRange) : this(DefaultTargetPart,
           ↪ possibleInternalReferencesRange, possibleExternalReferencesRange) { }
191
192        /// <summary>
193        /// <para>
194        /// Initializes a new <see cref="LinksConstants{TLinkAddress}"/> instance.
195        /// </para>
196        /// <para></para>
197        /// </summary>
198        /// <param name="enableExternalReferencesSupport">
199        /// <para>A enable external references support.</para>
200        /// <para></para>
201        /// </param>
202        [MethodImpl(MethodImplOptions.AggressiveInlining)]
203        public LinksConstants(bool enableExternalReferencesSupport) :
           ↪ this(GetDefaultInternalReferencesRange(enableExternalReferencesSupport),
           ↪ GetDefaultExternalReferencesRange(enableExternalReferencesSupport)) { }
204
205        /// <summary>
206        /// <para>
207        /// Initializes a new <see cref="LinksConstants{TLinkAddress}"/> instance.
208        /// </para>
209        /// <para></para>
210        /// </summary>
211        /// <param name="targetPart">
212        /// <para>A target part.</para>
213        /// <para></para>
214        /// </param>
215        /// <param name="possibleInternalReferencesRange">
216        /// <para>A possible internal references range.</para>
217        /// <para></para>
218        /// </param>
219        [MethodImpl(MethodImplOptions.AggressiveInlining)]
220        public LinksConstants(int targetPart, Range<TLinkAddress>
           ↪ possibleInternalReferencesRange) : this(targetPart, possibleInternalReferencesRange,
           ↪ null) { }
221
222        /// <summary>
223        /// <para>
224        /// Initializes a new <see cref="LinksConstants{TLinkAddress}"/> instance.
225        /// </para>
226        /// <para></para>
227        /// </summary>
228        /// <param name="possibleInternalReferencesRange">
229        /// <para>A possible internal references range.</para>
230        /// <para></para>
231        /// </param>
232        [MethodImpl(MethodImplOptions.AggressiveInlining)]
233        public LinksConstants(Range<TLinkAddress> possibleInternalReferencesRange) :
           ↪ this(DefaultTargetPart, possibleInternalReferencesRange, null) { }
234
235        /// <summary>
236        /// <para>
237        /// Initializes a new <see cref="LinksConstants{TLinkAddress}"/> instance.
238        /// </para>
239        /// <para></para>
240        /// </summary>
241        [MethodImpl(MethodImplOptions.AggressiveInlining)]
242        public LinksConstants() : this(DefaultTargetPart, enableExternalReferencesSupport:
           ↪ false) { }
243
244        /// <summary>
245        /// <para>
246        /// Gets the default internal references range using the specified enable external
           ↪ references support.
```

```csharp
247         /// </para>
248         /// <para></para>
249         /// </summary>
250         /// <param name="enableExternalReferencesSupport">
251         /// <para>The enable external references support.</para>
252         /// <para></para>
253         /// </param>
254         /// <returns>
255         /// <para>A range of t link address</para>
256         /// <para></para>
257         /// </returns>
258         [MethodImpl(MethodImplOptions.AggressiveInlining)]
259         public static Range<TLinkAddress> GetDefaultInternalReferencesRange(bool
            ↪  enableExternalReferencesSupport)
260         {
261             if (enableExternalReferencesSupport)
262             {
263                 return (_one, _uInt64ToAddressConverter.Convert(Hybrid<TLinkAddress>.HalfOfNumbe
                    ↪  rValuesRange));
264             }
265             else
266             {
267                 return (_one, NumericType<TLinkAddress>.MaxValue);
268             }
269         }
270
271         /// <summary>
272         /// <para>
273         /// Gets the default external references range using the specified enable external
            ↪  references support.
274         /// </para>
275         /// <para></para>
276         /// </summary>
277         /// <param name="enableExternalReferencesSupport">
278         /// <para>The enable external references support.</para>
279         /// <para></para>
280         /// </param>
281         /// <returns>
282         /// <para>A range of t link address</para>
283         /// <para></para>
284         /// </returns>
285         [MethodImpl(MethodImplOptions.AggressiveInlining)]
286         public static Range<TLinkAddress>? GetDefaultExternalReferencesRange(bool
            ↪  enableExternalReferencesSupport)
287         {
288             if (enableExternalReferencesSupport)
289             {
290                 return (Hybrid<TLinkAddress>.ExternalZero, NumericType<TLinkAddress>.MaxValue);
291             }
292             else
293             {
294                 return null;
295             }
296         }
297     }
298 }
```

## 1.12 ./csharp/Platform.Data/LinksConstantsBase.cs

```csharp
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data
4  {
5      /// <summary>
6      /// <para>
7      /// Represents the links constants base.
8      /// </para>
9      /// <para></para>
10     /// </summary>
11     public abstract class LinksConstantsBase
12     {
13         /// <summary>
14         /// <para>
15         /// The default target part.
16         /// </para>
17         /// <para></para>
18         /// </summary>
19         public static readonly int DefaultTargetPart = 2;
20     }
```

```
21    }

1.13  ./csharp/Platform.Data/LinksConstantsExtensions.cs
 1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 2
 3  using System.Runtime.CompilerServices;
 4
 5  namespace Platform.Data
 6  {
 7      /// <summary>
 8      /// <para>
 9      /// Represents the links constants extensions.
10      /// </para>
11      /// <para></para>
12      /// </summary>
13      public static class LinksConstantsExtensions
14      {
15          /// <summary>
16          /// <para>
17          /// Determines whether is reference.
18          /// </para>
19          /// <para></para>
20          /// </summary>
21          /// <typeparam name="TLinkAddress">
22          /// <para>The link address.</para>
23          /// <para></para>
24          /// </typeparam>
25          /// <param name="linksConstants">
26          /// <para>The links constants.</para>
27          /// <para></para>
28          /// </param>
29          /// <param name="address">
30          /// <para>The address.</para>
31          /// <para></para>
32          /// </param>
33          /// <returns>
34          /// <para>The bool</para>
35          /// <para></para>
36          /// </returns>
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          public static bool IsReference<TLinkAddress>(this LinksConstants<TLinkAddress>
              ↪  linksConstants, TLinkAddress address) => linksConstants.IsInternalReference(address)
              ↪  || linksConstants.IsExternalReference(address);
39
40          /// <summary>
41          /// <para>
42          /// Determines whether is internal reference.
43          /// </para>
44          /// <para></para>
45          /// </summary>
46          /// <typeparam name="TLinkAddress">
47          /// <para>The link address.</para>
48          /// <para></para>
49          /// </typeparam>
50          /// <param name="linksConstants">
51          /// <para>The links constants.</para>
52          /// <para></para>
53          /// </param>
54          /// <param name="address">
55          /// <para>The address.</para>
56          /// <para></para>
57          /// </param>
58          /// <returns>
59          /// <para>The bool</para>
60          /// <para></para>
61          /// </returns>
62          [MethodImpl(MethodImplOptions.AggressiveInlining)]
63          public static bool IsInternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
              ↪  linksConstants, TLinkAddress address) =>
              ↪  linksConstants.InternalReferencesRange.Contains(address);
64
65          /// <summary>
66          /// <para>
67          /// Determines whether is external reference.
68          /// </para>
69          /// <para></para>
70          /// </summary>
71          /// <typeparam name="TLinkAddress">
```

```csharp
72    /// <para>The link address.</para>
73    /// <para></para>
74    /// </typeparam>
75    /// <param name="linksConstants">
76    /// <para>The links constants.</para>
77    /// <para></para>
78    /// </param>
79    /// <param name="address">
80    /// <para>The address.</para>
81    /// <para></para>
82    /// </param>
83    /// <returns>
84    /// <para>The bool</para>
85    /// <para></para>
86    /// </returns>
87    [MethodImpl(MethodImplOptions.AggressiveInlining)]
88    public static bool IsExternalReference<TLinkAddress>(this LinksConstants<TLinkAddress>
      ↪  linksConstants, TLinkAddress address) =>
      ↪  linksConstants.ExternalReferencesRange?.Contains(address) ?? false;
89    }
90 }
```

## 1.14  ./csharp/Platform.Data/Numbers/Raw/AddressToRawNumberConverter.cs

```csharp
1   using System.Runtime.CompilerServices;
2   using Platform.Converters;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Numbers.Raw
7   {
8       /// <summary>
9       /// <para>
10      /// Represents the address to raw number converter.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="IConverter{TLinkAddress}"/>
15      public class AddressToRawNumberConverter<TLinkAddress> : IConverter<TLinkAddress>
16      {
17          /// <summary>
18          /// <para>
19          /// Converts the source.
20          /// </para>
21          /// <para></para>
22          /// </summary>
23          /// <param name="source">
24          /// <para>The source.</para>
25          /// <para></para>
26          /// </param>
27          /// <returns>
28          /// <para>The link</para>
29          /// <para></para>
30          /// </returns>
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public TLinkAddress Convert(TLinkAddress source) => new Hybrid<TLinkAddress>(source,
            ↪  isExternal: true);
33      }
34  }
```

## 1.15  ./csharp/Platform.Data/Numbers/Raw/RawNumberToAddressConverter.cs

```csharp
1   using System.Runtime.CompilerServices;
2   using Platform.Converters;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Numbers.Raw
7   {
8       /// <summary>
9       /// <para>
10      /// Represents the raw number to address converter.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="IConverter{TLinkAddress}"/>
15      public class RawNumberToAddressConverter<TLinkAddress> : IConverter<TLinkAddress>
16      {
17          /// <summary>
18          /// <para>
```

```
19    /// The default.
20    /// </para>
21    /// <para></para>
22    /// </summary>
23    static private readonly UncheckedConverter<long, TLinkAddress> _converter =
      ↪  UncheckedConverter<long, TLinkAddress>.Default;

24
25    /// <summary>
26    /// <para>
27    /// Converts the source.
28    /// </para>
29    /// <para></para>
30    /// </summary>
31    /// <param name="source">
32    /// <para>The source.</para>
33    /// <para></para>
34    /// </param>
35    /// <returns>
36    /// <para>The link</para>
37    /// <para></para>
38    /// </returns>
39    [MethodImpl(MethodImplOptions.AggressiveInlining)]
40    public TLinkAddress Convert(TLinkAddress source) => _converter.Convert(new
      ↪  Hybrid<TLinkAddress>(source).AbsoluteValue);
41    }
42  }
```

## 1.16   ./csharp/Platform.Data/Point.cs

```
1   using System;
2   using System.Collections;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5   using Platform.Exceptions;
6   using Platform.Ranges;
7   using Platform.Collections;
8
9   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11  namespace Platform.Data
12  {
13      /// <summary>
14      /// <para>
15      /// Represents the point.
16      /// </para>
17      /// <para></para>
18      /// </summary>
19      /// <seealso cref="IEquatable{LinkAddress{TLinkAddress}}"/>
20      /// <seealso cref="IList{TLinkAddress}"/>
21      public class Point<TLinkAddress> : IEquatable<LinkAddress<TLinkAddress>>, IList<TLinkAddress>
22      {
23          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            ↪  EqualityComparer<TLinkAddress>.Default;

24
25          /// <summary>
26          /// <para>
27          /// Gets the index value.
28          /// </para>
29          /// <para></para>
30          /// </summary>
31          public TLinkAddress Index
32          {
33              [MethodImpl(MethodImplOptions.AggressiveInlining)]
34              get;
35          }

36
37          /// <summary>
38          /// <para>
39          /// Gets the size value.
40          /// </para>
41          /// <para></para>
42          /// </summary>
43          public int Size
44          {
45              [MethodImpl(MethodImplOptions.AggressiveInlining)]
46              get;
47          }

48
49          /// <summary>
50          /// <para>
```

```csharp
        /// The not supported exception.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress this[int index]
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                if (index < Size)
                {
                    return Index;
                }
                else
                {
                    throw new IndexOutOfRangeException();
                }
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set => throw new NotSupportedException();
        }

        /// <summary>
        /// <para>
        /// Gets the count value.
        /// </para>
        /// <para></para>
        /// </summary>
        public int Count
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Size;
        }

        /// <summary>
        /// <para>
        /// Gets the is read only value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool IsReadOnly
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => true;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="Point{TLinkAddress}"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="index">
        /// <para>A index.</para>
        /// <para></para>
        /// </param>
        /// <param name="size">
        /// <para>A size.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Point(TLinkAddress index, int size)
        {
            Index = index;
            Size = size;
        }

        /// <summary>
        /// <para>
        /// Adds the item.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="item">
        /// <para>The item.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TLinkAddress item) => throw new NotSupportedException();
```

```csharp
130
131        /// <summary>
132        /// <para>
133        /// Clears this instance.
134        /// </para>
135        /// <para></para>
136        /// </summary>
137        [MethodImpl(MethodImplOptions.AggressiveInlining)]
138        public void Clear() => throw new NotSupportedException();
139
140        /// <summary>
141        /// <para>
142        /// Determines whether this instance contains.
143        /// </para>
144        /// <para></para>
145        /// </summary>
146        /// <param name="item">
147        /// <para>The item.</para>
148        /// <para></para>
149        /// </param>
150        /// <returns>
151        /// <para>The bool</para>
152        /// <para></para>
153        /// </returns>
154        [MethodImpl(MethodImplOptions.AggressiveInlining)]
155        public virtual bool Contains(TLinkAddress item) => _equalityComparer.Equals(item, Index);
156
157        /// <summary>
158        /// <para>
159        /// Copies the to using the specified array.
160        /// </para>
161        /// <para></para>
162        /// </summary>
163        /// <param name="array">
164        /// <para>The array.</para>
165        /// <para></para>
166        /// </param>
167        /// <param name="arrayIndex">
168        /// <para>The array index.</para>
169        /// <para></para>
170        /// </param>
171        [MethodImpl(MethodImplOptions.AggressiveInlining)]
172        public void CopyTo(TLinkAddress[] array, int arrayIndex) => array[arrayIndex] = Index;
173
174        /// <summary>
175        /// <para>
176        /// Gets the enumerator.
177        /// </para>
178        /// <para></para>
179        /// </summary>
180        /// <returns>
181        /// <para>An enumerator of t link address</para>
182        /// <para></para>
183        /// </returns>
184        [MethodImpl(MethodImplOptions.AggressiveInlining)]
185        public IEnumerator<TLinkAddress> GetEnumerator()
186        {
187            for (int i = 0; i < Size; i++)
188            {
189                yield return Index;
190            }
191        }
192
193        /// <summary>
194        /// <para>
195        /// Indexes the of using the specified item.
196        /// </para>
197        /// <para></para>
198        /// </summary>
199        /// <param name="item">
200        /// <para>The item.</para>
201        /// <para></para>
202        /// </param>
203        /// <returns>
204        /// <para>The int</para>
205        /// <para></para>
206        /// </returns>
207        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
208        public virtual int IndexOf(TLinkAddress item) => _equalityComparer.Equals(item, Index) ?
       ↪  0 : -1;

209

210        /// <summary>
211        /// <para>
212        /// Inserts the index.
213        /// </para>
214        /// <para></para>
215        /// </summary>
216        /// <param name="index">
217        /// <para>The index.</para>
218        /// <para></para>
219        /// </param>
220        /// <param name="item">
221        /// <para>The item.</para>
222        /// <para></para>
223        /// </param>
224        [MethodImpl(MethodImplOptions.AggressiveInlining)]
225        public void Insert(int index, TLinkAddress item) => throw new NotSupportedException();

226

227        /// <summary>
228        /// <para>
229        /// Determines whether this instance remove.
230        /// </para>
231        /// <para></para>
232        /// </summary>
233        /// <param name="item">
234        /// <para>The item.</para>
235        /// <para></para>
236        /// </param>
237        /// <returns>
238        /// <para>The bool</para>
239        /// <para></para>
240        /// </returns>
241        [MethodImpl(MethodImplOptions.AggressiveInlining)]
242        public bool Remove(TLinkAddress item) => throw new NotSupportedException();

243

244        /// <summary>
245        /// <para>
246        /// Removes the at using the specified index.
247        /// </para>
248        /// <para></para>
249        /// </summary>
250        /// <param name="index">
251        /// <para>The index.</para>
252        /// <para></para>
253        /// </param>
254        [MethodImpl(MethodImplOptions.AggressiveInlining)]
255        public void RemoveAt(int index) => throw new NotSupportedException();

256

257        /// <summary>
258        /// <para>
259        /// Gets the enumerator.
260        /// </para>
261        /// <para></para>
262        /// </summary>
263        /// <returns>
264        /// <para>The enumerator</para>
265        /// <para></para>
266        /// </returns>
267        [MethodImpl(MethodImplOptions.AggressiveInlining)]
268        IEnumerator IEnumerable.GetEnumerator()
269        {
270            for (int i = 0; i < Size; i++)
271            {
272                yield return Index;
273            }
274        }

275

276        /// <summary>
277        /// <para>
278        /// Determines whether this instance equals.
279        /// </para>
280        /// <para></para>
281        /// </summary>
282        /// <param name="other">
283        /// <para>The other.</para>
284        /// <para></para>
```

```csharp
285         /// </param>
286         /// <returns>
287         /// <para>The bool</para>
288         /// <para></para>
289         /// </returns>
290         [MethodImpl(MethodImplOptions.AggressiveInlining)]
291         public virtual bool Equals(LinkAddress<TLinkAddress> other) => other == null ? false :
       ↪  _equalityComparer.Equals(Index, other.Index);
292
293         [MethodImpl(MethodImplOptions.AggressiveInlining)]
294         public static implicit operator TLinkAddress(Point<TLinkAddress> linkAddress) =>
       ↪  linkAddress.Index;
295
296         /// <summary>
297         /// <para>
298         /// Determines whether this instance equals.
299         /// </para>
300         /// <para></para>
301         /// </summary>
302         /// <param name="obj">
303         /// <para>The obj.</para>
304         /// <para></para>
305         /// </param>
306         /// <returns>
307         /// <para>The bool</para>
308         /// <para></para>
309         /// </returns>
310         [MethodImpl(MethodImplOptions.AggressiveInlining)]
311         public override bool Equals(object obj) => obj is Point<TLinkAddress> linkAddress ?
       ↪  Equals(linkAddress) : false;
312
313         /// <summary>
314         /// <para>
315         /// Gets the hash code.
316         /// </para>
317         /// <para></para>
318         /// </summary>
319         /// <returns>
320         /// <para>The int</para>
321         /// <para></para>
322         /// </returns>
323         [MethodImpl(MethodImplOptions.AggressiveInlining)]
324         public override int GetHashCode() => Index.GetHashCode();
325
326         /// <summary>
327         /// <para>
328         /// Returns the string.
329         /// </para>
330         /// <para></para>
331         /// </summary>
332         /// <returns>
333         /// <para>The string</para>
334         /// <para></para>
335         /// </returns>
336         [MethodImpl(MethodImplOptions.AggressiveInlining)]
337         public override string ToString() => Index.ToString();
338
339         [MethodImpl(MethodImplOptions.AggressiveInlining)]
340         public static bool operator ==(Point<TLinkAddress> left, Point<TLinkAddress> right)
341         {
342             if (left == null && right == null)
343             {
344                 return true;
345             }
346             if (left == null)
347             {
348                 return false;
349             }
350             return left.Equals(right);
351         }
352
353         [MethodImpl(MethodImplOptions.AggressiveInlining)]
354         public static bool operator !=(Point<TLinkAddress> left, Point<TLinkAddress> right) =>
       ↪  !(left == right);
355
356         /// <summary>
357         /// <para>
358         /// Determines whether is full point.
```

```csharp
359        /// </para>
360        /// <para></para>
361        /// </summary>
362        /// <param name="link">
363        /// <para>The link.</para>
364        /// <para></para>
365        /// </param>
366        /// <returns>
367        /// <para>The bool</para>
368        /// <para></para>
369        /// </returns>
370        [MethodImpl(MethodImplOptions.AggressiveInlining)]
371        public static bool IsFullPoint(params TLinkAddress[] link) =>
    ↪    IsFullPoint((IList<TLinkAddress>?)link);
372
373        /// <summary>
374        /// <para>
375        /// Determines whether is full point.
376        /// </para>
377        /// <para></para>
378        /// </summary>
379        /// <param name="link">
380        /// <para>The link.</para>
381        /// <para></para>
382        /// </param>
383        /// <returns>
384        /// <para>The bool</para>
385        /// <para></para>
386        /// </returns>
387        [MethodImpl(MethodImplOptions.AggressiveInlining)]
388        public static bool IsFullPoint(IList<TLinkAddress>? link)
389        {
390            Ensure.Always.ArgumentNotEmpty(link, nameof(link));
391            Ensure.Always.ArgumentInRange(link.Count, (2, int.MaxValue), nameof(link), "Cannot
    ↪    determine link's pointness using only its identifier.");
392            return IsFullPointUnchecked(link);
393        }
394
395        /// <summary>
396        /// <para>
397        /// Determines whether is full point unchecked.
398        /// </para>
399        /// <para></para>
400        /// </summary>
401        /// <param name="link">
402        /// <para>The link.</para>
403        /// <para></para>
404        /// </param>
405        /// <returns>
406        /// <para>The result.</para>
407        /// <para></para>
408        /// </returns>
409        [MethodImpl(MethodImplOptions.AggressiveInlining)]
410        public static bool IsFullPointUnchecked(IList<TLinkAddress>? link)
411        {
412            var result = true;
413            for (var i = 1; result && i < link.Count; i++)
414            {
415                result = _equalityComparer.Equals(link[0], link[i]);
416            }
417            return result;
418        }
419
420        /// <summary>
421        /// <para>
422        /// Determines whether is partial point.
423        /// </para>
424        /// <para></para>
425        /// </summary>
426        /// <param name="link">
427        /// <para>The link.</para>
428        /// <para></para>
429        /// </param>
430        /// <returns>
431        /// <para>The bool</para>
432        /// <para></para>
433        /// </returns>
434        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
435         public static bool IsPartialPoint(params TLinkAddress[] link) =>
            ↪    IsPartialPoint((IList<TLinkAddress>?)link);
436
437         /// <summary>
438         /// <para>
439         /// Determines whether is partial point.
440         /// </para>
441         /// <para></para>
442         /// </summary>
443         /// <param name="link">
444         /// <para>The link.</para>
445         /// <para></para>
446         /// </param>
447         /// <returns>
448         /// <para>The bool</para>
449         /// <para></para>
450         /// </returns>
451         [MethodImpl(MethodImplOptions.AggressiveInlining)]
452         public static bool IsPartialPoint(IList<TLinkAddress>? link)
453         {
454             Ensure.Always.ArgumentNotEmpty(link, nameof(link));
455             Ensure.Always.ArgumentInRange(link.Count, (2, int.MaxValue), nameof(link), "Cannot
                ↪    determine link's pointness using only its identifier.");
456             return IsPartialPointUnchecked(link);
457         }
458
459         /// <summary>
460         /// <para>
461         /// Determines whether is partial point unchecked.
462         /// </para>
463         /// <para></para>
464         /// </summary>
465         /// <param name="link">
466         /// <para>The link.</para>
467         /// <para></para>
468         /// </param>
469         /// <returns>
470         /// <para>The result.</para>
471         /// <para></para>
472         /// </returns>
473         [MethodImpl(MethodImplOptions.AggressiveInlining)]
474         public static bool IsPartialPointUnchecked(IList<TLinkAddress>? link)
475         {
476             var result = false;
477             for (var i = 1; !result && i < link.Count; i++)
478             {
479                 result = _equalityComparer.Equals(link[0], link[i]);
480             }
481             return result;
482         }
483     }
484 }
```

## 1.17 ./csharp/Platform.Data/Universal/IUniLinks.cs

```
1  using System;
2  using System.Collections.Generic;
3  using Platform.Delegates;
4
5  // ReSharper disable TypeParameterCanBeVariant
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Universal
9  {
10     /// <remarks>Minimal sufficient universal Links API (for bulk operations).</remarks>
11     public partial interface IUniLinks<TLinkAddress>
12     {
13         /// <summary>
14         /// <para>
15         /// Triggers the condition.
16         /// </para>
17         /// <para></para>
18         /// </summary>
19         /// <param name="condition">
20         /// <para>The condition.</para>
21         /// <para></para>
22         /// </param>
23         /// <param name="substitution">
24         /// <para>The substitution.</para>
```

```csharp
25         /// <para></para>
26         /// </param>
27         /// <returns>
28         /// <para>A list of i list i list t link address</para>
29         /// <para></para>
30         /// </returns>
31         IList<IList<IList<TLinkAddress>?>> Trigger(IList<TLinkAddress>? condition,
              ↪  IList<TLinkAddress>? substitution);
32     }
33
34     /// <remarks>Minimal sufficient universal Links API (for step by step operations).</remarks>
35     public partial interface IUniLinks<TLinkAddress>
36     {
37         /// <returns>
38         /// TLinkAddress that represents True (was finished fully) or TLinkAddress that
              ↪  represents False (was stopped).
39         /// This is done to assure ability to push up stop signal through recursion stack.
40         /// </returns>
41         /// <remarks>
42         /// { 0, 0, 0 } => { itself, itself, itself } // create
43         /// { 1, any, any } => { itself, any, 3 } // update
44         /// { 3, any, any } => { 0, 0, 0 } // delete
45         /// </remarks>
46         TLinkAddress Trigger(IList<TLinkAddress>? patternOrCondition, ReadHandler<TLinkAddress>?
              ↪  matchHandler,
47                      IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>?
                          ↪  substitutionHandler);
48
49         /// <summary>
50         /// <para>
51         /// Triggers the restriction.
52         /// </para>
53         /// <para></para>
54         /// </summary>
55         /// <param name="restriction">
56         /// <para>The restriction.</para>
57         /// <para></para>
58         /// </param>
59         /// <param name="matchedHandler">
60         /// <para>The matched handler.</para>
61         /// <para></para>
62         /// </param>
63         /// <param name="substitution">
64         /// <para>The substitution.</para>
65         /// <para></para>
66         /// </param>
67         /// <param name="substitutedHandler">
68         /// <para>The substituted handler.</para>
69         /// <para></para>
70         /// </param>
71         /// <returns>
72         /// <para>The link address</para>
73         /// <para></para>
74         /// </returns>
75         TLinkAddress Trigger(IList<TLinkAddress>? restriction, WriteHandler<TLinkAddress>?
              ↪  matchedHandler,
76             IList<TLinkAddress>? substitution, WriteHandler<TLinkAddress>? substitutedHandler);
77     }
78
79     /// <remarks>Extended with small optimization.</remarks>
80     public partial interface IUniLinks<TLinkAddress>
81     {
82         /// <remarks>
83         /// Something simple should be simple and optimized.
84         /// </remarks>
85         TLinkAddress Count(IList<TLinkAddress>? restrictions);
86     }
87 }
```

## 1.18  ./csharp/Platform.Data/Universal/IUniLinksCRUD.cs

```csharp
1  using System;
2  using System.Collections.Generic;
3
4  // ReSharper disable TypeParameterCanBeVariant
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Universal
8  {
```

```csharp
    /// <remarks>
    /// CRUD aliases for IUniLinks.
    /// </remarks>
    public interface IUniLinksCRUD<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Reads the part type.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="partType">
        /// <para>The part type.</para>
        /// <para></para>
        /// </param>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
        TLinkAddress Read(int partType, TLinkAddress link);
        /// <summary>
        /// <para>
        /// Reads the handler.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="handler">
        /// <para>The handler.</para>
        /// <para></para>
        /// </param>
        /// <param name="pattern">
        /// <para>The pattern.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
        TLinkAddress Read(Func<TLinkAddress, bool> handler, IList<TLinkAddress>? pattern);
        /// <summary>
        /// <para>
        /// Creates the parts.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="parts">
        /// <para>The parts.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
        TLinkAddress Create(IList<TLinkAddress>? parts);
        /// <summary>
        /// <para>
        /// Updates the before.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="before">
        /// <para>The before.</para>
        /// <para></para>
        /// </param>
        /// <param name="after">
        /// <para>The after.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link address</para>
        /// <para></para>
        /// </returns>
        TLinkAddress Update(IList<TLinkAddress>? before, IList<TLinkAddress>? after);
        /// <summary>
```

```
87          /// <para>
88          /// Deletes the parts.
89          /// </para>
90          /// <para></para>
91          /// </summary>
92          /// <param name="parts">
93          /// <para>The parts.</para>
94          /// <para></para>
95          /// </param>
96          TLinkAddress Delete(IList<TLinkAddress>? parts);
97      }
98  }
```

## 1.19 ./csharp/Platform.Data/Universal/IUniLinksGS.cs

```
1   using System;
2   using System.Collections.Generic;
3
4   // ReSharper disable TypeParameterCanBeVariant
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Universal
8   {
9       /// <remarks>
10      /// Get/Set aliases for IUniLinks.
11      /// </remarks>
12      public interface IUniLinksGS<TLinkAddress>
13      {
14          /// <summary>
15          /// <para>
16          /// Gets the part type.
17          /// </para>
18          /// <para></para>
19          /// </summary>
20          /// <param name="partType">
21          /// <para>The part type.</para>
22          /// <para></para>
23          /// </param>
24          /// <param name="link">
25          /// <para>The link.</para>
26          /// <para></para>
27          /// </param>
28          /// <returns>
29          /// <para>The link address</para>
30          /// <para></para>
31          /// </returns>
32          TLinkAddress Get(int partType, TLinkAddress link);
33          /// <summary>
34          /// <para>
35          /// Gets the handler.
36          /// </para>
37          /// <para></para>
38          /// </summary>
39          /// <param name="handler">
40          /// <para>The handler.</para>
41          /// <para></para>
42          /// </param>
43          /// <param name="pattern">
44          /// <para>The pattern.</para>
45          /// <para></para>
46          /// </param>
47          /// <returns>
48          /// <para>The link address</para>
49          /// <para></para>
50          /// </returns>
51          TLinkAddress Get(Func<TLinkAddress, bool> handler, IList<TLinkAddress>? pattern);
52          /// <summary>
53          /// <para>
54          /// Sets the before.
55          /// </para>
56          /// <para></para>
57          /// </summary>
58          /// <param name="before">
59          /// <para>The before.</para>
60          /// <para></para>
61          /// </param>
62          /// <param name="after">
63          /// <para>The after.</para>
64          /// <para></para>
```

```
65    /// </param>
66    /// <returns>
67    /// <para>The link address</para>
68    /// <para></para>
69    /// </returns>
70    TLinkAddress Set(IList<TLinkAddress>? before, IList<TLinkAddress>? after);
71  }
72 }
```

## 1.20 ./csharp/Platform.Data/Universal/IUniLinksIO.cs

```
1  using System;
2  using System.Collections.Generic;
3
4  // ReSharper disable TypeParameterCanBeVariant
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Universal
8  {
9      /// <remarks>
10     /// In/Out aliases for IUniLinks.
11     /// TLinkAddress can be any number type of any size.
12     /// </remarks>
13     public interface IUniLinksIO<TLinkAddress>
14     {
15         /// <remarks>
16         /// default(TLinkAddress) means any link.
17         /// Single element pattern means just element (link).
18         /// Handler gets array of link contents.
19         /// * link[0] is index or identifier.
20         /// * link[1] is source or first.
21         /// * link[2] is target or second.
22         /// * link[3] is linker or third.
23         /// * link[n] is nth part/parent/element/value
24         /// of link (if variable length links used).
25         ///
26         /// Stops and returns false if handler return false.
27         ///
28         /// Acts as Each, Foreach, Select, Search, Match &amp; ...
29         ///
30         /// Handles all links in store if pattern/restrictions is not defined.
31         /// </remarks>
32         bool Out(Func<IList<TLinkAddress>?, bool> handler, IList<TLinkAddress>? pattern);
33
34         /// <remarks>
35         /// default(TLinkAddress) means itself.
36         /// Equivalent to:
37         /// * creation if before == null
38         /// * deletion if after == null
39         /// * update if before != null &amp;&amp; after != null
40         /// * default(TLinkAddress) if before == null &amp;&amp; after == null
41         ///
42         /// Possible interpretation
43         /// * In(null, new[] { }) creates point (link that points to itself using minimum number
44         ///    of parts).
45         /// * In(new[] { 4 }, null) deletes 4th link.
46         /// * In(new[] { 4 }, new [] { 5 }) delete 5th link if it exists and moves 4th link to
47         ///    5th index.
48         /// * In(new[] { 4 }, new [] { 0, 2, 3 }) replaces 4th link with new doublet link (with
49         ///    2 as source and 3 as target), 0 means it can be placed in any address.
50         /// ...
51         /// </remarks>
52         TLinkAddress In(IList<TLinkAddress>? before, IList<TLinkAddress>? after);
53     }
54 }
```

## 1.21 ./csharp/Platform.Data/Universal/IUniLinksIOWithExtensions.cs

```
1  // ReSharper disable TypeParameterCanBeVariant
2  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
3
4  using System.Collections.Generic;
5
6  namespace Platform.Data.Universal
7  {
8      /// <remarks>Contains some optimizations of Out.</remarks>
9      public interface IUniLinksIOWithExtensions<TLinkAddress> : IUniLinksIO<TLinkAddress>
10     {
11         /// <remarks>
12         /// default(TLinkAddress) means nothing or null.
```

```
13          /// Single element pattern means just element (link).
14          /// OutPart(n, null) returns default(TLinkAddress).
15          /// OutPart(0, pattern) ~ Exists(link) or Search(pattern)
16          /// OutPart(1, pattern) ~ GetSource(link) or GetSource(Search(pattern))
17          /// OutPart(2, pattern) ~ GetTarget(link) or GetTarget(Search(pattern))
18          /// OutPart(3, pattern) ~ GeTLinkAddresser(link) or GeTLinkAddresser(Search(pattern))
19          /// OutPart(n, pattern) => For any variable length links, returns link or
   ↪        default(TLinkAddress).
20          ///
21          /// Outs(returns) inner contents of link, its part/parent/element/value.
22          /// </remarks>
23          TLinkAddress OutOne(int partType, IList<TLinkAddress>? pattern);
24
25          /// <remarks>OutCount() returns total links in store as array.</remarks>
26          IList<IList<TLinkAddress>?> OutAll(IList<TLinkAddress>? pattern);
27
28          /// <remarks>OutCount() returns total amount of links in store.</remarks>
29          ulong OutCount(IList<TLinkAddress>? pattern);
30      }
31  }
```

## 1.22 ./csharp/Platform.Data/Universal/IUniLinksRW.cs

```
1   using System;
2   using System.Collections.Generic;
3
4   // ReSharper disable TypeParameterCanBeVariant
5   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7   namespace Platform.Data.Universal
8   {
9       /// <remarks>
10      /// Read/Write aliases for IUniLinks.
11      /// </remarks>
12      public interface IUniLinksRW<TLinkAddress>
13      {
14          /// <summary>
15          /// <para>
16          /// Reads the part type.
17          /// </para>
18          /// <para></para>
19          /// </summary>
20          /// <param name="partType">
21          /// <para>The part type.</para>
22          /// <para></para>
23          /// </param>
24          /// <param name="link">
25          /// <para>The link.</para>
26          /// <para></para>
27          /// </param>
28          /// <returns>
29          /// <para>The link address</para>
30          /// <para></para>
31          /// </returns>
32          TLinkAddress Read(int partType, TLinkAddress link);
33          /// <summary>
34          /// <para>
35          /// Determines whether this instance read.
36          /// </para>
37          /// <para></para>
38          /// </summary>
39          /// <param name="handler">
40          /// <para>The handler.</para>
41          /// <para></para>
42          /// </param>
43          /// <param name="pattern">
44          /// <para>The pattern.</para>
45          /// <para></para>
46          /// </param>
47          /// <returns>
48          /// <para>The bool</para>
49          /// <para></para>
50          /// </returns>
51          bool Read(Func<TLinkAddress, bool> handler, IList<TLinkAddress>? pattern);
52          /// <summary>
53          /// <para>
54          /// Writes the before.
55          /// </para>
56          /// <para></para>
```

```
57        /// </summary>
58        /// <param name="before">
59        /// <para>The before.</para>
60        /// <para></para>
61        /// </param>
62        /// <param name="after">
63        /// <para>The after.</para>
64        /// <para></para>
65        /// </param>
66        /// <returns>
67        /// <para>The link address</para>
68        /// <para></para>
69        /// </returns>
70        TLinkAddress Write(IList<TLinkAddress>? before, IList<TLinkAddress>? after);
71    }
72  }
```

## 1.23 ./csharp/Platform.Data/WriteHandlerState.cs

```
1   using System.Collections.Generic;
2   using Platform.Delegates;
3
4   namespace Platform.Data
5   {
6       public struct WriteHandlerState<TLinkAddress>
7       {
8           private readonly EqualityComparer<TLinkAddress> _equalityComparer;
9           public TLinkAddress Result;
10          public WriteHandler<TLinkAddress>? Handler;
11          private TLinkAddress Break;
12
13          public WriteHandlerState(TLinkAddress @continue, TLinkAddress @break,
            ↪  WriteHandler<TLinkAddress>? handler)
14          {
15              _equalityComparer = EqualityComparer<TLinkAddress>.Default;
16              Break = @break;
17              Result = @continue;
18              Handler = handler;
19          }
20
21          public void Apply(TLinkAddress result)
22          {
23              var isAlreadyBreak = _equalityComparer.Equals(Break, Result);
24              var isCurrentlyBreak = _equalityComparer.Equals(Break, result);
25              if (isAlreadyBreak || !isCurrentlyBreak)
26              {
27                  return;
28              }
29              Handler = null;
30              Result = Break;
31
32          }
33
34          public TLinkAddress Handle(IList<TLinkAddress> before, IList<TLinkAddress> after)
35          {
36              if (Handler != null)
37              {
38                  Apply(Handler(before, after));
39              }
40              return Result;
41          }
42      }
43  }
```

## 1.24 ./csharp/Platform.Data.Tests/HybridTests.cs

```
1   using Xunit;
2
3   namespace Platform.Data.Tests
4   {
5       /// <summary>
6       /// <para>
7       /// Represents the hybrid tests.
8       /// </para>
9       /// <para></para>
10      /// </summary>
11      public static class HybridTests
12      {
13          /// <summary>
14          /// <para>
15          /// Tests that object constructor test.
```

```csharp
16        /// </para>
17        /// <para></para>
18        /// </summary>
19        [Fact]
20        public static void ObjectConstructorTest()
21        {
22            Assert.Equal(0, new Hybrid<byte>(unchecked((byte)128)).AbsoluteValue);
23            Assert.Equal(0, new Hybrid<byte>((object)128).AbsoluteValue);
24            Assert.Equal(1, new Hybrid<byte>(unchecked((byte)-1)).AbsoluteValue);
25            Assert.Equal(1, new Hybrid<byte>((object)-1).AbsoluteValue);
26            Assert.Equal(0, new Hybrid<byte>(unchecked((byte)0)).AbsoluteValue);
27            Assert.Equal(0, new Hybrid<byte>((object)0).AbsoluteValue);
28            Assert.Equal(1, new Hybrid<byte>(unchecked((byte)1)).AbsoluteValue);
29            Assert.Equal(1, new Hybrid<byte>((object)1).AbsoluteValue);
30        }
31    }
32 }
```

## 1.25  ./csharp/Platform.Data.Tests/LinksConstantsTests.cs

```csharp
1  using Xunit;
2  using Platform.Reflection;
3  using Platform.Converters;
4  using Platform.Numbers;
5
6  namespace Platform.Data.Tests
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the links constants tests.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     public static class LinksConstantsTests
15     {
16         /// <summary>
17         /// <para>
18         /// Tests that constructor test.
19         /// </para>
20         /// <para></para>
21         /// </summary>
22         [Fact]
23         public static void ConstructorTest()
24         {
25             var constants = new LinksConstants<ulong>(enableExternalReferencesSupport: true);
26             Assert.Equal(Hybrid<ulong>.ExternalZero,
                    ↪  constants.ExternalReferencesRange.Value.Minimum);
27             Assert.Equal(ulong.MaxValue, constants.ExternalReferencesRange.Value.Maximum);
28         }
29
30         /// <summary>
31         /// <para>
32         /// Tests that external references test.
33         /// </para>
34         /// <para></para>
35         /// </summary>
36         [Fact]
37         public static void ExternalReferencesTest()
38         {
39             TestExternalReferences<ulong, long>();
40             TestExternalReferences<uint, int>();
41             TestExternalReferences<ushort, short>();
42             TestExternalReferences<byte, sbyte>();
43         }
44         private static void TestExternalReferences<TUnsigned, TSigned>()
45         {
46             var unsingedOne = Arithmetic.Increment(default(TUnsigned));
47             var converter = UncheckedConverter<TSigned, TUnsigned>.Default;
48             var half = converter.Convert(NumericType<TSigned>.MaxValue);
49             LinksConstants<TUnsigned> constants = new LinksConstants<TUnsigned>((unsingedOne,
                    ↪  half), (Arithmetic.Add(half, unsingedOne), NumericType<TUnsigned>.MaxValue));
50
51             var minimum = new Hybrid<TUnsigned>(default, isExternal: true);
52             var maximum = new Hybrid<TUnsigned>(half, isExternal: true);
53
54             Assert.True(constants.IsExternalReference(minimum));
55             Assert.True(minimum.IsExternal);
56             Assert.False(minimum.IsInternal);
57             Assert.True(constants.IsExternalReference(maximum));
```

```
            Assert.True(maximum.IsExternal);
            Assert.False(maximum.IsInternal);
        }
    }
}
```

# Index