1.1 ./csharp/Platform.IO/ConsoleCancellation.cs

```
using System;
using System.Runtime.CompilerServices;
using System.Threading;
using Platform.Disposables;
using Platform.Threading;

namespace Platform.IO
{
    /// <summary>
    /// <para>Represents the class that simplifies the console applications implementation that can be terminated manually during execution.</para>
    /// <para>Представляет класс, упрощающий реализацию консольных приложений, выполнение которых может быть прекращено в процессе выполнения вручную.</para>
    /// </summary>
    public class ConsoleCancellation : DisposableBase
    {
        /// <summary>
        /// <para>Gets the <see cref="CancellationTokenSource"/> class instance.</para>
        /// <para>Возвращает экземпляр класса <see cref="CancellationTokenSource"/>.</para>
        /// </summary>
        public CancellationTokenSource Source
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>
        /// <para>Gets the <see cref="CancellationToken"/> class instance.</para>
        /// <para>Возвращает экземпляр класса <see cref="CancellationToken"/>.</para>
        /// </summary>
        public CancellationToken Token
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        /// <summary>
        /// <para>Gets a <see cref="Boolean"/> value that determines whether cancellation was requested for the <see cref="CancellationTokenSource"/>.</para>
        /// <para>Возвращает значение типа <see cref="Boolean"/>, определяющее, запрошена ли отмена для <see cref="CancellationTokenSource"/>.</para>
        /// </summary>
        public bool IsRequested
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Source.IsCancellationRequested;
        }

        /// <summary>
        /// <para>Gets a <see cref="Boolean"/> value that determines whether cancellation was not requested for the <see cref="CancellationTokenSource"/>.</para>
        /// <para>Возвращает значение типа <see cref="Boolean"/>, определяющее, не запрошена ли отмена для <see cref="CancellationTokenSource"/>.</para>
        /// </summary>
        public bool NotRequested
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => !Source.IsCancellationRequested;
        }

        /// <summary>
        /// <para>Initializes a <see cref="ConsoleCancellation"/> class instance, using a <see cref="CancellationTokenSource"/> and its token. The <see cref="ConsoleCancellation"/> subscribes to the <see cref="Console.CancelKeyPress"/> event on initialization.</para>
        /// <para>Инициализирует экземпляр класса <see cref="ConsoleCancellation"/>, используя <see cref="CancellationTokenSource"/> и его токен. <see cref="ConsoleCancellation"/> подписывается на событие <see cref="Console.CancelKeyPress"/> при инициализации.</para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ConsoleCancellation()
        {
            Source = new CancellationTokenSource();
            Token = Source.Token;
            Console.CancelKeyPress += OnCancelKeyPress;
        }
```

```

        /// <summary>
        /// <para>Forces cancellation request.</para>
        /// <para>Принудительно запрашивает отмену.</para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ForceCancellation() => Source.Cancel();

        /// <summary>
        /// <para>Suspends the current thread until a cancellation is requested.</para>
        /// <para>Приостанавливает текущий поток до запроса на отмену.</para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Wait()
        {
            while (NotRequested)
            {
                ThreadHelpers.Sleep();
            }
        }

        /// <summary>
        /// <para>Unsubscribes from the <see cref="Console.CancelKeyPress"/> event and attempts
        ↪  to dispose the <see cref="CancellationTokenSource"/>.</para>
        /// <para>Отписывается от события <see cref="Console.CancelKeyPress"/> и пытается
        ↪  высвободить ресурсы, используемые <see cref="CancellationTokenSource"/>.</para>
        /// </summary>
        /// <param name="manual">
        /// <para>A <see cref="Boolean"/> value that determines whether the disposal was
        ↪  triggered manually (by the developer's code) or was executed automatically without
        ↪  an explicit indication from a developer.</para>
        /// <para>Значение типа <see cref="Boolean"/>, определяющие было ли высвобождение
        ↪  вызвано вручную (кодом разработчика) или же выполнилось автоматически без явного
        ↪  указания со стороны разработчика.</para>
        /// </param>
        /// <param name="wasDisposed">
        /// <para>A <see cref="Boolean"/> value that determines whether the <see
        ↪  cref="ConsoleCancellation"/> was released before a call to this method.</para>
        /// <para>Значение типа <see cref="Boolean"/>, определяющие были ли освобождены ресурсы,
        ↪  используемые <see cref="ConsoleCancellation"/> до вызова данного метода.</para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                Console.CancelKeyPress -= OnCancelKeyPress;
                Source.DisposeIfPossible();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void OnCancelKeyPress(object sender, ConsoleCancelEventArgs e)
        {
            e.Cancel = true;
            if (NotRequested)
            {
                Source.Cancel();
            }
        }
    }
}
```

1.2 ./csharp/Platform.IO/ConsoleHelpers.cs

```
using System;
using System.Diagnostics;
using System.Runtime.CompilerServices;
using Platform.Collections;
using Platform.Collections.Arrays;

namespace Platform.IO
{
    /// <summary>
    /// <para>Represents the set of helper methods to work with the console.</para>
    /// <para>Представляет набор вспомогательных методов для работы с консолью.</para>
    /// </summary>
    public static class ConsoleHelpers
    {
```

```
15          /// <summary>
16          /// <para>Requests and expects a user to press any key in the console.</para>
17          /// <para>Запрашивает и ожидает нажатие любой клавиши пользователем в консоли.</para>
18          /// </summary>
19          [MethodImpl(MethodImplOptions.AggressiveInlining)]
20          public static void PressAnyKeyToContinue()
21          {
22              Console.WriteLine("Press any key to continue.");
23              Console.ReadKey();
24          }
25
26          /// <summary>
27          /// <para>Gets an argument's value with the specified <paramref name="index"/> from the
            ↪  <paramref name="args"/> array and if it's absent requests a user to input it in the
            ↪  console.</para>
28          /// <para>Получает значение аргумента с указанным <paramref name="index"/> из массива
            ↪  <paramref name="args"/>, а если оно отсутствует запрашивает его ввод в консоли у
            ↪  пользователя.</para>
29          /// </summary>
30          /// <param name="index">
31          /// <para>The ordinal number of the argument in the array.</para>
32          /// <para>Порядковый номер аргумента в массиве.</para>
33          /// </param>
34          /// <param name="args">
35          /// <para>The argument array passed to the application.</para>
36          /// <para>Массив аргументов переданных приложению.</para>
37          /// </param>
38          /// <returns>
39          /// <para>The value with the specified <paramref name="index"/> extracted from the
            ↪  <paramref name="args"/> array or entered by a user in the console.</para>
40          /// <para>Значение с указанным <paramref name="index"/>, извлечённое из массива
            ↪  <paramref name="args"/>, или введённое пользователем в консоли.</para>
41          /// </returns>
42          [MethodImpl(MethodImplOptions.AggressiveInlining)]
43          public static string GetOrReadArgument(int index, params string[] args) =>
            ↪  GetOrReadArgument(index, $"{index + 1} argument", args);
44
45          /// <summary>
46          /// <para>Gets an argument's value with the specified <paramref name="index"/> from the
            ↪  <paramref name="args"/> array and if it's absent requests a user to input it in the
            ↪  console.</para>
47          /// <para>Получает значение аргумента с указанным <paramref name="index"/> из массива
            ↪  <paramref name="args"/>, а если оно отсутствует запрашивает его ввод в консоли у
            ↪  пользователя.</para>
48          /// </summary>
49          /// <param name="index">
50          /// <para>The ordinal number of the argument in the array.</para>
51          /// <para>Порядковый номер аргумента в массиве.</para>
52          /// </param>
53          /// <param name="readMessage">
54          /// <para>The message's text to a user describing which argument is being entered at the
            ↪  moment. If the <paramref name="args"/> array doesn't contain the element with the
            ↪  specified <paramref name="index"/>, then this message is used.</para>
55          /// <para>Текст сообщения пользователю описывающее какой аргумент вводится в данный
            ↪  момент. Это сообщение используется только если массив <paramref name="args"/> не
            ↪  содержит аргумента с указанным <paramref name="index"/>.</para>
56          /// </param>
57          /// <param name="args">
58          /// <para>The argument array passed to the application.</para>
59          /// <para>Массив аргументов переданных приложению.</para>
60          /// </param>
61          /// <returns>
62          /// <para>The value with the specified <paramref name="index"/> extracted from the
            ↪  <paramref name="args"/> array or entered by a user in the console.</para>
63          /// <para>Значение с указанным <paramref name="index"/>, извлечённое из массива
            ↪  <paramref name="args"/>, или введённое пользователем в консоли.</para>
64          /// </returns>
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          public static string GetOrReadArgument(int index, string readMessage, params string[]
            ↪  args)
67          {
68              if (!args.TryGetElement(index, out string result))
69              {
70                  Console.Write($"{readMessage}: ");
71                  result = Console.ReadLine();
72              }
```

```
            if (string.IsNullOrEmpty(result))
            {
                return "";
            }
            else
            {
                return result.Trim().TrimSingle('"').Trim();
            }
        }

        /// <summary>
        /// <para>Outputs the <paramref name="string"/> to the console.</para>
        /// <para>Выводит <paramref name="string"/> в консоль.</para>
        /// </summary>
        /// <param name="string">
        /// <para>The string to output to the console.</para>
        /// <para>Строка выводимая в консоль.</para>
        /// </param>
        /// <remarks>
        /// <para>The method is only executed if the application was compiled with the DEBUG
        ↪  directive.</para>
        /// <para>Метод выполняется только если приложение было скомпилировано с директивой
        ↪  DEBUG.</para>
        /// </remarks>
        [Conditional("DEBUG")]
        public static void Debug(string @string) => Console.WriteLine(@string);

        /// <summary>
        /// <para>Writes text representations of the specified <paramref name="args"/> array
        ↪  objects to the standard output stream using the specified <paramref name="format"/>,
        ↪  followed by the current line terminator.</para>
        /// <para>Записывает текстовые представления объектов заданного массива <paramref
        ↪  name="args"/>, в стандартный выходной поток с использованием заданного <paramref
        ↪  name="format"/>, за которым следует текущий признак конца строки.</para>
        /// </summary>
        /// <param name="format">
        /// <para>The composite format string.</para>
        /// <para>Строка составного формата.</para>
        /// </param>
        /// <param name="args">
        /// <para>The object array to write to the standard output stream using <paramref
        ↪  name="format"/>.</para>
        /// <para>Массив объектов для записи в стандартный выходной поток с использованием
        ↪  <paramref name="format"/>.</para>
        /// </param>
        /// <remarks>
        /// <para>The method is only executed if the application was compiled with the DEBUG
        ↪  directive.</para>
        /// <para>Метод выполняется только если приложение было скомпилировано с директивой
        ↪  DEBUG.</para>
        /// </remarks>
        [Conditional("DEBUG")]
        public static void Debug(string format, params object[] args) =>
        ↪  Console.WriteLine(format, args);
    }
}
```

1.3 ./csharp/Platform.IO/FileHelpers.cs

```
using System;
using System.IO;
using System.Runtime.CompilerServices;
using Platform.Unsafe;

namespace Platform.IO
{
    /// <summary>
    /// <para>Represents the set of helper methods to work with files.</para>
    /// <para>Представляет набор вспомогательных методов для работы с файлами.</para>
    /// </summary>
    public static class FileHelpers
    {
        /// <summary>
        /// <para>Reads all the text and returns character array from a file at the <paramref
        ↪  name="path"/>.</para>
        /// <para>Читает весь текст и возвращает массив символов из файла находящегося в
        ↪  <paramref name="path"/>.</para>
        /// </summary>
```

```
        /// <param name="path">
        /// <para>The path to a file, from which to read the character array.</para>
        /// <para>Путь к файлу, из которого нужно прочитать массив символов.</para>
        /// </param>
        /// <returns>
        /// <para>The character array from a file at the <paramref name="path"/>.</para>
        /// <para>Массив символов из файла находящегося в <paramref name="path"/>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static char[] ReadAllChars(string path) => File.ReadAllText(path).ToCharArray();

        /// <summary>
        /// <para>Reads and returns all <typeparamref name="T"/> structure values from a file at the <paramref name="path"/>.</para>
        /// <para>Считывает и возвращает все значения структур типа <typeparamref name="T"/> из файла находящегося в <paramref name="path"/>.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The structure type.</para>
        /// <para>Тип структуры.</para>
        /// </typeparam>
        /// <param name="path">
        /// <para>The path to a file, from which to read <typeparamref name="T"/> structure values array.</para>
        /// <para>Путь к файлу, из которого нужно прочитать массив значений структур типа <typeparamref name="T"/>.</para>
        /// </param>
        /// <returns>
        /// <para>The <typeparamref name="T"/> structure values array.</para>
        /// <para>Массив значений структур типа <typeparamref name="T"/>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] ReadAll<T>(string path)
            where T : struct
        {
            using var reader = File.OpenRead(path);
            return reader.ReadAll<T>();
        }

        /// <summary>
        /// <para>Reads and returns the first <typeparamref name="T"/> structure value from a file at the <paramref name="path"/>.</para>
        /// <para>Считывает и возвращает первое значение структуры типа <typeparamref name="T"/> из файла находящегося в <paramref name="path"/>.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The structure type.</para>
        /// <para>Тип структуры.</para>
        /// </typeparam>
        /// <param name="path">
        /// <para>The path to a file, from which to read the first <typeparamref name="T"/> structure value.</para>
        /// <para>Путь к файлу, из которого нужно прочитать значение первой структуры типа <typeparamref name="T"/>.</para>
        /// </param>
        /// <returns>
        /// <para>The <typeparamref name="T"/> structure value if read from a file at the <paramref name="path"/> is successful; otherwise the default <typeparamref name="T"/> structure value.</para>
        /// <para>Значение структуры типа <typeparamref name="T"/> если чтение из файла находящегося в <paramref name="path"/> прошло успешно, иначе значение структуры типа <typeparamref name="T"/> по умолчанию.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T ReadFirstOrDefault<T>(string path)
            where T : struct
        {
            using var fileStream = GetValidFileStreamOrDefault<T>(path);
            return fileStream?.ReadOrDefault<T>() ?? default;
        }

        /// <summary>
        /// <para>Returns the <see cref="FileStream"/> opened for reading from a file at the <paramref name="path"/> if the file exists, not empty and its size is a multiple of the <typeparamref name="TStruct"/> structure size; otherwise <see langword="null"/>.</para>
```

```
79          /// <para>Возвращает <see cref="FileStream"/>, открытый для чтения из файла находящегося
            ↪  в <paramref name="path"/>, если файл существует, не пуст и его размер кратен размеру
            ↪  структуры типа <typeparamref name="TStruct"/>, а иначе <see langword="null"/>.</para>
80          /// </summary>
81          /// <typeparam name="TStruct">
82          /// <para>The structure type.</para>
83          /// <para>Тип структуры.</para>
84          /// </typeparam>
85          /// <param name="path">
86          /// <para>The path to a file to validate.</para>
87          /// <para>Путь к проверяемому файлу.</para>
88          /// </param>
89          /// <returns>
90          /// <para>A <see cref="FileStream"/> opened for reading in the case of successful check;
            ↪  otherwise <see langword="null"/>.</para>
91          /// <para><see cref="FileStream"/>, открытый для чтения в случае успешной проверки, а
            ↪  иначе <see langword="null"/>.</para>
92          /// </returns>
93          /// <exception cref="InvalidOperationException">
94          /// <para>The size of a file at the <paramref name="path"/> is not a multiple of the
            ↪  required <typeparamref name="TStruct"/> structure size.</para>
95          /// <para>Размер файла находящегося в <paramref name="path"/> не кратен требуемому
            ↪  размеру структуры типа <typeparamref name="TStruct"/>.</para>
96          /// </exception>
97          [MethodImpl(MethodImplOptions.AggressiveInlining)]
98          private static FileStream GetValidFileStreamOrDefault<TStruct>(string path) where
            ↪  TStruct : struct => GetValidFileStreamOrDefault(path, Structure<TStruct>.Size);
99
100         /// <summary>
101         /// <para>Returns the <see cref="FileStream"/> opened for reading from a file at the
            ↪  <paramref name="path"/> if the file exists, not empty and its size is a multiple of
            ↪  the required <paramref name="elementSize"/>; otherwise <see langword="null"/>.</para>
102         /// <para>Возвращает <see cref="FileStream"/>, открытый для чтения из файла находящегося
            ↪  в <paramref name="path"/>, если файл существует, не пуст и его размер кратен
            ↪  <paramref name="elementSize"/>, а иначе <see langword="null"/>.</para>
103         /// </summary>
104         /// <param name="path">
105         /// <para>The path to a file to validate.</para>
106         /// <para>Путь к проверяемому файлу.</para>
107         /// </param>
108         /// <param name="elementSize">
109         /// <para>Required size of elements located in a file at the <paramref
            ↪  name="path"/>.</para>
110         /// <para>Требуемый размер элементов, находящихся в файле находящегося в <paramref
            ↪  name="path"/>.</para>
111         /// </param>
112         /// <returns>
113         /// <para>A <see cref="FileStream"/> opened for reading in the case of successful check;
            ↪  otherwise <see langword="null"/>.</para>
114         /// <para><see cref="FileStream"/>, открытый для чтения в случае успешной проверки, а
            ↪  иначе <see langword="null"/>.</para>
115         /// </returns>
116         /// <exception cref="InvalidOperationException">
117         /// <para>The size of a file at the <paramref name="path"/> is not a multiple of the
            ↪  required <paramref name="elementSize"/>.</para>
118         /// <para>Размер файла находящегося в <paramref name="path"/> не кратен требуемому
            ↪  <paramref name="elementSize"/>.</para>
119         /// </exception>
120         [MethodImpl(MethodImplOptions.AggressiveInlining)]
121         private static FileStream GetValidFileStreamOrDefault(string path, int elementSize)
122         {
123             if (!File.Exists(path))
124             {
125                 return null;
126             }
127             var fileSize = GetSize(path);
128             if (fileSize % elementSize != 0)
129             {
130                 throw new InvalidOperationException($"File is not aligned to elements with size
                    ↪  {elementSize}.");
131             }
132             return fileSize > 0 ? File.OpenRead(path) : null;
133         }
134
135         /// <summary>
```

```
136         /// <para>Reads and returns the last <typeparamref name="T"/> structure value from a
            ↪  file at the <paramref name="path"/>.</para>
137         /// <para>Считывает и возвращает последнее значение структуры типа <typeparamref
            ↪  name="T"/> из файла находящегося в <paramref name="path"/>.</para>
138         /// </summary>
139         /// <typeparam name="T">
140         /// <para>The structure type.</para>
141         /// <para>Тип структуры.</para>
142         /// </typeparam>
143         /// <param name="path">
144         /// <para>The path to a file, from which to read the last <typeparamref name="T"/>
            ↪  structure value.</para>
145         /// <para>Путь к файлу, из которого нужно прочитать значение последней структуры типа
            ↪  <typeparamref name="T"/>.</para>
146         /// </param>
147         /// <returns>
148         /// <para>The <typeparamref name="T"/> structure value if read from a file at the
            ↪  <paramref name="path"/> is successful; otherwise the default <typeparamref
            ↪  name="T"/> structure value.</para>
149         /// <para>Значение структуры типа <typeparamref name="T"/> из файла находящегося в
            ↪  <paramref name="path"/> в случае успешного чтения, иначе значение по умолчанию
            ↪  структуры типа <typeparamref name="T"/>.</para>
150         /// </returns>
151         [MethodImpl(MethodImplOptions.AggressiveInlining)]
152         public static T ReadLastOrDefault<T>(string path)
153             where T : struct
154         {
155             var elementSize = Structure<T>.Size;
156             using var reader = GetValidFileStreamOrDefault(path, elementSize);
157             if (reader == null)
158             {
159                 return default;
160             }
161             var totalElements = reader.Length / elementSize;
162             reader.Position = (totalElements - 1) * elementSize; // Set to last element
163             return reader.ReadOrDefault<T>();
164         }
165
166         /// <summary>
167         /// <para>Writes <typeparamref name="T"/> structure value at the beginning of a file at
            ↪  the <paramref name="path"/>.</para>
168         /// <para>Записывает значение структуры типа <typeparamref name="T"/> в начало файла
            ↪  находящегося в <paramref name="path"/>.</para>
169         /// </summary>
170         /// <typeparam name="T">
171         /// <para>The structure type.</para>
172         /// <para>Тип структуры.</para>
173         /// </typeparam>
174         /// <param name="path">
175         /// <para>The path to a file to be changed or created.</para>
176         /// <para>Путь к файлу, который будет изменён или создан.</para>
177         /// </param>
178         /// <param name="value">
179         /// <para>The <typeparamref name="T"/> structure value to be written at the beginning of
            ↪  a file at the <paramref name="path"/>.</para>
180         /// <para>Значение структуры типа <typeparamref name="T"/>, записываемое в начало файла
            ↪  находящегося в <paramref name="path"/>.</para>
181         /// </param>
182         [MethodImpl(MethodImplOptions.AggressiveInlining)]
183         public static void WriteFirst<T>(string path, T value)
184             where T : struct
185         {
186             using var writer = File.OpenWrite(path);
187             writer.Position = 0;
188             writer.Write(value);
189         }
190
191         /// <summary>
192         /// <para>Opens or creates a file at the <paramref name="path"/> and returns its <see
            ↪  cref="FileStream"/> with append mode and write access.</para>
193         /// <para>Открывает или создаёт файл находящийся в <paramref name="path"/> и возвращает
            ↪  его <see cref="FileStream"/> с режимом дополнения и доступом на запись.</para>
194         /// </summary>
195         /// <param name="path">
196         /// <para>The path to a file to open or create.</para>
197         /// <para>Путь к файлу, который нужно открыть или создать.</para>
198         /// </param>
```

```
199         /// <returns>
200         /// <para>The <see cref="FileStream"/> with append mode and write access.</para>
201         /// <para><see cref="FileStream"/> с режимом дополнения и доступом на запись.</para>
202         /// </returns>
203         [MethodImpl(MethodImplOptions.AggressiveInlining)]
204         public static FileStream Append(string path) => File.Open(path, FileMode.Append,
            ↪ FileAccess.Write);
205
206         /// <summary>
207         /// <para>Returns the size of a file at the <paramref name="path"/> if the file exists;
            ↪ otherwise 0.</para>
208         /// <para>Возвращает размер файла находящегося в <paramref name="path"/> если тот
            ↪ существует, иначе 0.</para>
209         /// </summary>
210         /// <param name="path">
211         /// <para>The path to a file to get size.</para>
212         /// <para>Путь к файлу, размер которого нужно получить.</para>
213         /// </param>
214         /// <returns>
215         /// <para>Size of a file at the <paramref name="path"/> if it exists; otherwise 0.</para>
216         /// <para>Размер файла если файл находящийся в <paramref name="path"/> существует, иначе
            ↪ 0.</para>
217         /// </returns>
218         [MethodImpl(MethodImplOptions.AggressiveInlining)]
219         public static long GetSize(string path) => File.Exists(path) ? new FileInfo(path).Length
            ↪ : 0;
220
221         /// <summary>
222         /// <para>Sets the <paramref name="size"/> for a file at the <paramref
            ↪ name="path"/>.</para>
223         /// <para>Устанавливает <paramref name="size"/> файлу находящемуся по пути <paramref
            ↪ name="path"/>.</para>
224         /// </summary>
225         /// <param name="path">
226         /// <para>The path to a file to be resized.</para>
227         /// <para>Путь к файлу, размер которого нужно изменить.</para>
228         /// </param>
229         /// <param name="size">
230         /// <para>The size to assign to a file at the <paramref name="path"/>.</para>
231         /// <para>Размер который будет присвоен файлу находящемуся по пути <paramref
            ↪ name="path"/>.</para>
232         /// </param>
233         [MethodImpl(MethodImplOptions.AggressiveInlining)]
234         public static void SetSize(string path, long size)
235         {
236             using var fileStream = File.Open(path, FileMode.OpenOrCreate);
237             if (fileStream.Length != size)
238             {
239                 fileStream.SetLength(size);
240             }
241         }
242
243         /// <summary>
244         /// <para>Removes all files from the directory at the path <paramref
            ↪ name="directory"/>.</para>
245         /// <para>Удаляет все файлы из директории находящейся по пути <paramref
            ↪ name="directory"/>.</para>
246         /// </summary>
247         /// <param name="directory">
248         /// <para>The path to the directory to be cleaned.</para>
249         /// <para>Путь к директории для очистки.</para>
250         /// </param>
251         [MethodImpl(MethodImplOptions.AggressiveInlining)]
252         public static void DeleteAll(string directory) => DeleteAll(directory, "*");
253
254         /// <summary>
255         /// <para>Removes files from the directory at the path <paramref name="directory"/>
            ↪ according to the <paramref name="searchPattern"/>.</para>
256         /// <para>Удаляет файлы из директории находящейся по пути <paramref name="directory"/> в
            ↪ соотвествии с <paramref name="searchPattern"/>.</para>
257         /// </summary>
258         /// <param name="directory">
259         /// <para>The path to the directory to be cleaned.</para>
260         /// <para>Путь к директории для очистки.</para>
261         /// </param>
262         /// <param name="searchPattern">
```

```
263         /// <para>The search pattern for files to be deleted in the directory at the path
            ↪  <paramref name="directory"/>.</para>
264         /// <para>Шаблон поиска для удаляемых файлов в директории находящейся по пути <paramref
            ↪  name="directory"/>.</para>
265         /// </param>
266         [MethodImpl(MethodImplOptions.AggressiveInlining)]
267         public static void DeleteAll(string directory, string searchPattern) =>
            ↪  DeleteAll(directory, searchPattern, SearchOption.TopDirectoryOnly);
268
269         /// <summary>
270         /// <para>Removes files from the directory at the path <paramref name="directory"/>
            ↪  according to the <paramref name="searchPattern"/> and the <paramref
            ↪  name="searchOption"/>.</para>
271         /// <para>Удаляет файлы из директории находящейся по пути <paramref name="directory"/> в
            ↪  соотвествии с <paramref name="searchPattern"/> и <paramref
            ↪  name="searchOption"/>.</para>
272         /// </summary>
273         /// <param name="directory">
274         /// <para>The path to the directory to be cleaned.</para>
275         /// <para>Путь к директории для очистки.</para>
276         /// </param>
277         /// <param name="searchPattern">
278         /// <para>The search pattern for files to be deleted in the directory at the path
            ↪  <paramref name="directory"/>.</para>
279         /// <para>Шаблон поиска для удаляемых файлов в директории находящейся по пути <paramref
            ↪  name="directory"/> .</para>
280         /// </param>
281         /// <param name="searchOption">
282         /// <para>The <see cref="SearchOption"/> value that determines whether to search only in
            ↪  the current the directory at the path <paramref name="directory"/>, or also in all
            ↪  subdirectories.</para>
283         /// <para>Значение <see cref="SearchOption"/> определяющее искать ли только в текущей
            ↪  директории находящейся по пути <paramref name="directory"/>, или также во всех
            ↪  субдиректориях.</para>
284         /// </param>
285         [MethodImpl(MethodImplOptions.AggressiveInlining)]
286         public static void DeleteAll(string directory, string searchPattern, SearchOption
            ↪  searchOption)
287         {
288             foreach (var file in Directory.EnumerateFiles(directory, searchPattern,
                ↪  searchOption))
289             {
290                 File.Delete(file);
291             }
292         }
293
294         /// <summary>
295         /// <para>Trincates the file at the <paramref name="path"/>.</para>
296         /// <para>Очищает содержимое файла по пути <paramref name="path"/>.</para>
297         /// </summary>
298         /// <param name="path">
299         /// <para>A path to a file to be truncated.</para>
300         /// <para>Путь к файлу для очистки содержимого.</para>
301         /// </param>
302         [MethodImpl(MethodImplOptions.AggressiveInlining)]
303         public static void Truncate(string path) => File.Open(path, FileMode.Truncate).Dispose();
304
305         /// <summary>
306         /// <para>Appends the <paramref name="content"/> to a file at the <paramref
            ↪  name="path"/>.</para>
307         /// <para>Добавляет <paramref name="content"/> в конец файла по пути <paramref
            ↪  name="path"/>.</para>
308         /// </summary>
309         /// <param name="path">
310         /// <para>The path to a file to be appended by the <paramref name="content"/>.</para>
311         /// <para>Путь к файлу для добавления <paramref name="content"/> в конец файла.</para>
312         /// </param>
313         /// <param name="content">
314         /// <para>A content to be appended to a file at the file at the <paramref
            ↪  name="path"/>.</para>
315         /// <para>Содержимое для добавления в конец файла по пути <paramref name="path"/>.</para>
316         /// </param>
317         [MethodImpl(MethodImplOptions.AggressiveInlining)]
318         public static void AppendLine(string path, string content)
319         {
320             using var writer = File.AppendText(path);
```

```
            writer.WriteLine(content);
        }

        /// <summary>
        /// <para>Performs the <paramref name="action"/> for each line of a file at the
        ↪  <paramref name="path"/>.</para>
        /// <para>Выполняет <paramref name="action"/> для каждой строчки файла по пути <paramref
        ↪  name="path"/>.</para>
        /// </summary>
        /// <param name="path">
        /// <para>A path to a file to perform the <paramref name="action"/> for each line of the
        ↪  file.</para>
        /// <para>Путь к файлу для выполнения <paramref name="action"/> для каждой строки
        ↪  файла.</para>
        /// </param>
        /// <param name="action">
        /// <para>An action to be performed for each line of a file at the <paramref
        ↪  name="path"/>.</para>
        /// <para>Действие выполняемое для каждой строчки файла по пути <paramref
        ↪  name="path"/>.</para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void EachLine(string path, Action<string> action)
        {
            using var reader = new StreamReader(path);
            string line;
            while ((line = reader.ReadLine()) != null)
            {
                action(line);
            }
        }
    }
}
```

1.4 ./csharp/Platform.IO/StreamExtensions.cs

```
using System.IO;
using System.Runtime.CompilerServices;
using Platform.Unsafe;

namespace Platform.IO
{
    /// <summary>
    /// <para>Represents the set of extension methods for <see cref="Stream"/> class
    ↪  instances.</para>
    /// <para>Представляет набор методов расширения для экземпляров класса <see
    ↪  cref="Stream"/>.</para>
    /// </summary>
    public static class StreamExtensions
    {
        /// <summary>
        /// <para>Writes a byte sequence that represents the <typeparamref name="T"/> structure
        ↪  <paramref name="value"/> to the <paramref name="stream"/> and moves the current
        ↪  position of the <paramref name="stream"/> by the number of written bytes.</para>
        /// <para>Записывает последовательность байт представляющую <paramref name="value"/>
        ↪  структуры типа <typeparamref name="T"/> в поток <paramref name="stream"/> и
        ↪  перемещает текущую позицию в <paramref name="stream"/> вперёд на число записанных
        ↪  байт.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The structure type.</para>
        /// <para>Тип структуры.</para>
        /// </typeparam>
        /// <param name="stream">
        /// <para>The stream to write to.</para>
        /// <para>Поток, в который осуществляется запись.</para>
        /// </param>
        /// <param name="value">
        /// <para>The <typeparamref name="T"/> structure value to be written to the <paramref
        ↪  name="stream"/>.</para>
        /// <para>Значение структуры типа <typeparamref name="T"/> которое будет записано в
        ↪  поток <paramref name="stream"/>.</para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Write<T>(this Stream stream, T value)
            where T : struct
        {
            var bytes = value.ToBytes();
```

```
            stream.Write(bytes, 0, bytes.Length);
        }

        /// <summary>
        /// <para>Reads a byte sequence that represents the <typeparamref name="T"/> structure value and moves the current position of the <paramref name="stream"/> by the number of read bytes.</para>
        /// <para>Считывает последовательность байт представляющих значение структуры типа <typeparamref name="T"/> и перемещает текущую позицию в потоке <paramref name="stream"/> вперёд на число прочитанных байт.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The structure type.</para>
        /// <para>Тип структуры.</para>
        /// </typeparam>
        /// <param name="stream">
        /// <para>The stream containing the <typeparamref name="T"/> structure value.</para>
        /// <para>Поток, содержащий значение структуры типа <typeparamref name="T"/>.</para>
        /// </param>
        /// <returns>
        /// <para>The <typeparamref name="T"/> structure value, if its bytes from the <paramref name="stream"/> are read; otherwise the default <typeparamref name="T"/> structure value.</para>
        /// <para>Значение структуры типа <typeparamref name="T"/>, если её байты из потока <paramref name="stream"/> были прочитаны, иначе значение структуры типа <typeparamref name="T"/> по умолчанию.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T ReadOrDefault<T>(this Stream stream)
            where T : struct
        {
            var size = Structure<T>.Size;
            var buffer = new byte[size];
            return stream.Read(buffer, 0, size) == size ? buffer.ToStructure<T>() : default;
        }

        /// <summary>
        /// <para>Reads and returns all <typeparamref name="T"/> structure values array from the <paramref name="stream"/>.</para>
        /// <para>Прочитывает и возвращает массив всех значений структур типа <typeparamref name="T"/> из потока <paramref name="stream"/>.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The structure type.</para>
        /// <para>Тип структуры.</para>
        /// </typeparam>
        /// <param name="stream">
        /// <para>The stream containing the <typeparamref name="T"/> structure values.</para>
        /// <para>Поток, содержащий значения структур типа <typeparamref name="T"/>.</para>
        /// </param>
        /// <returns>
        /// <para>The <typeparamref name="T"/> structure values array read from the <paramref name="stream"/>.</para>
        /// <para>Массив с значениями структур типа <typeparamref name="T"/>, прочитанными из потока <paramref name="stream"/>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] ReadAll<T>(this Stream stream)
            where T : struct
        {
            var size = Structure<T>.Size;
            var buffer = new byte[size];
            var elementsLength = stream.Length / size;
            var elements = new T[elementsLength];
            for (var i = 0; i < elementsLength; i++)
            {
                stream.Read(buffer, 0, size);
                elements[i] = buffer.ToStructure<T>();
            }
            return elements;
        }
    }
}
```

1.5 ./csharp/Platform.IO/TemporaryFile.cs

```
using Platform.Disposables;
using System;
```

```
using System.IO;
using System.Runtime.CompilerServices;

namespace Platform.IO
{
    /// <summary>
    /// <para>Represents a self-deleting temporary file.</para>
    /// <para>Представляет самоудаляющийся временный файл.</para>
    /// </summary>
    public class TemporaryFile : DisposableBase
    {
        /// <summary>
        /// <para>Gets a temporary file path.</para>
        /// <para>Возвращает путь к временному файлу.</para>
        /// </summary>
        public readonly string Filename;

        /// <summary>
        /// <para>Converts the <see cref="TemporaryFile"/> instance to <see cref="string"/> using the <see cref="Filename"/> field value.</para>
        /// <para>Преобразует экземпляр <see cref="TemporaryFile"/> в <see cref="string"/> используя поле <see cref="Filename"/>.</para>
        /// </summary>
        /// <param name="file">
        /// <para>A <see cref="TemporaryFile"/> instance.</para>
        /// <para>Экземпляр <see cref="TemporaryFile"/>.</para>
        /// </param>
        /// <returns>
        /// <para>Path to the temporary file.</para>
        /// <para>Путь к временному файлу.</para>
        /// </returns>
        public static implicit operator string(TemporaryFile file) => file.Filename;

        /// <summary>
        /// <para>Initializes a <see cref="TemporaryFile"/> instance.</para>
        /// <para>Инициализирует экземпляр класса <see cref="TemporaryFile"/>.</para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TemporaryFile() => Filename = TemporaryFiles.UseNew();

        /// <summary>
        /// <para>Deletes the temporary file.</para>
        /// <para>Удаляет временный файл.</para>
        /// </summary>
        /// <param name="manual">
        /// <para>A <see cref="Boolean"/> value that determines whether the disposal was triggered manually (by the developer's code) or was executed automatically without an explicit indication from a developer.</para>
        /// <para>Значение типа <see cref="Boolean"/>, определяющие было ли высвобождение вызвано вручную (кодом разработчика) или же выполнилось автоматически без явного указания со стороны разработчика.</para>
        /// </param>
        /// <param name="wasDisposed">
        /// <para>A <see cref="Boolean"/> value that determines whether the <see cref="TemporaryFile"/> was released before a call to this method.</para>
        /// <para>Значение типа <see cref="Boolean"/>, определяющие были ли освобождены ресурсы, используемые <see cref="TemporaryFile"/> до вызова данного метода.</para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                File.Delete(Filename);
            }
        }
    }
}
```

1.6 ./csharp/Platform.IO/TemporaryFiles.cs

```
using System.IO;
using System.Reflection;
using System.Runtime.CompilerServices;

namespace Platform.IO
{
    /// <summary>
    /// <para>Represents the set of helper methods to work with temporary files.</para>
```

```
    /// <para>Представляет набор вспомогательных методов для работы с временными файлами.</para>
    /// </summary>
    public class TemporaryFiles
    {
        private const string UserFilesListFileNamePrefix = ".used-temporary-files.txt";
        private static readonly object UsedFilesListLock = new();
        private static readonly string UsedFilesListFilename =
        ↪  Assembly.GetExecutingAssembly().Location + UserFilesListFileNamePrefix;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void AddToUsedFilesList(string filename)
        {
            lock (UsedFilesListLock)
            {
                FileHelpers.AppendLine(UsedFilesListFilename, filename);
            }
        }

        /// <summary>
        /// <para>Gets a temporary file and adds it to the used files list.</para>
        /// <para>Получает временный файл и добавляет его в список использованных файлов.</para>
        /// </summary>
        /// <returns>
        /// <para>The temporary file path.</para>
        /// <para>Путь временного файла.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string UseNew()
        {
            var filename = Path.GetTempFileName();
            AddToUsedFilesList(filename);
            return filename;
        }

        /// <summary>
        /// <para>Deletes all previously used temporary files and clears the files list.</para>
        /// <para>Удаляет все ранее использованные временные файлы и очищает список
        ↪  файлов.</para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void DeleteAllPreviouslyUsed()
        {
            lock (UsedFilesListLock)
            {
                var listFilename = UsedFilesListFilename;
                if (File.Exists(listFilename))
                {
                    FileHelpers.EachLine(listFilename, File.Delete);
                    FileHelpers.Truncate(listFilename);
                }
            }
        }
    }
}
```

1.7 ./csharp/Platform.IO.Tests/FileHelpersTests.cs

```
using System.IO;
using Xunit;

namespace Platform.IO.Tests
{
    public class FileHelpersTests
    {
        [Fact]
        public void WriteReadTest()
        {
            var temporaryFile = Path.GetTempFileName();
            var originalValue = 42UL;
            FileHelpers.WriteFirst(temporaryFile, originalValue);
            var readValue = FileHelpers.ReadFirstOrDefault<ulong>(temporaryFile);
            Assert.Equal(readValue, originalValue);
            File.Delete(temporaryFile);
        }
    }
}
```

1.8 ./csharp/Platform.IO.Tests/TemporaryFileTests.cs

```
using Xunit;
using System.IO;
```

```
using System.Diagnostics;

namespace Platform.IO.Tests
{
    public class TemporaryFileTests
    {
        [Fact]
        public void TemporaryFileTest()
        {
            using Process process = new();
            process.StartInfo.FileName =
            ↪  Path.GetFullPath(Path.Combine(Directory.GetCurrentDirectory(), "..", "..", "..",
            ↪  "..", "Platform.IO.Tests.TemporaryFileTest", "bin", "Debug", "net5",
            ↪  "Platform.IO.Tests.TemporaryFileTest"));
            process.StartInfo.UseShellExecute = false;
            process.StartInfo.RedirectStandardOutput = true;
            process.Start();
            var path = process.StandardOutput.ReadLine();
            Assert.True(File.Exists(path));
            process.WaitForExit();
            Assert.False(File.Exists(path));
        }

        [Fact]
        public void TemporaryFileTestWithoutConsoleApp()
        {
            string fileName;
            using (TemporaryFile tempFile = new())
            {
                fileName = tempFile;
                Assert.True(File.Exists(fileName));
            }
            Assert.False(File.Exists(fileName));
        }
    }
}
```

Index