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

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

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

namespace Platform.IO
{
    public class ConsoleCancellation : DisposableBase
    {
        public CancellationTokenSource Source
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public CancellationToken Token
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        }

        public bool IsRequested
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => Source.IsCancellationRequested;
        }

        public bool NotRequested
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get => !Source.IsCancellationRequested;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ConsoleCancellation()
        {
            Source = new CancellationTokenSource();
            Token = Source.Token;
            Console.CancelKeyPress += OnCancelKeyPress;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ForceCancellation() => Source.Cancel();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Wait()
        {
            while (NotRequested)
            {
                ThreadHelpers.Sleep();
            }
        }

        [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;

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

namespace Platform.IO
{
    public static class ConsoleHelpers
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void PressAnyKeyToContinue()
        {
            Console.WriteLine("Press any key to continue.");
            Console.ReadKey();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string GetOrReadArgument(int index, params string[] args) =>
        ↪  GetOrReadArgument(index, $"{index + 1} argument", args);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string GetOrReadArgument(int index, string readMessage, params string[]
        ↪  args)
        {
            if (!args.TryGetElement(index, out string result))
            {
                Console.Write($"{readMessage}: ");
                result = Console.ReadLine();
            }
            if (string.IsNullOrEmpty(result))
            {
                return "";
            }
            else
            {
                return result.Trim().TrimSingle('"').Trim();
            }
        }

        [Conditional("DEBUG")]
        public static void Debug(string @string) => Console.WriteLine(@string);

        [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;

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

namespace Platform.IO
{
    public static class FileHelpers
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static char[] ReadAllChars(string path) => File.ReadAllText(path).ToCharArray();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] ReadAll<T>(string path)
            where T : struct
        {
            using var reader = File.OpenRead(path);
            return reader.ReadAll<T>();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T ReadFirstOrDefault<T>(string path)
            where T : struct
        {
```

```
            using var fileStream = GetValidFileStreamOrDefault<T>(path);
            return fileStream?.ReadOrDefault<T>() ?? default;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static FileStream GetValidFileStreamOrDefault<TStruct>(string path) where
        ↪  TStruct : struct => GetValidFileStreamOrDefault(path, Structure<TStruct>.Size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static FileStream GetValidFileStreamOrDefault(string path, int elementSize)
        {
            if (!File.Exists(path))
            {
                return null;
            }
            var fileSize = GetSize(path);
            if (fileSize % elementSize != 0)
            {
                throw new InvalidOperationException($"File is not aligned to elements with size
                ↪  {elementSize}.");
            }
            return fileSize > 0 ? File.OpenRead(path) : null;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T ReadLastOrDefault<T>(string path)
            where T : struct
        {
            var elementSize = Structure<T>.Size;
            using var reader = GetValidFileStreamOrDefault(path, elementSize);
            if (reader == null)
            {
                return default;
            }
            var totalElements = reader.Length / elementSize;
            reader.Position = (totalElements - 1) * elementSize; // Set to last element
            return reader.ReadOrDefault<T>();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WriteFirst<T>(string path, T value)
            where T : struct
        {
            using var writer = File.OpenWrite(path);
            writer.Position = 0;
            writer.Write(value);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static FileStream Append(string path) => File.Open(path, FileMode.Append,
        ↪  FileAccess.Write);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long GetSize(string path) => File.Exists(path) ? new FileInfo(path).Length
        ↪  : 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void SetSize(string path, long size)
        {
            using var fileStream = File.Open(path, FileMode.OpenOrCreate);
            if (fileStream.Length != size)
            {
                fileStream.SetLength(size);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void DeleteAll(string directory) => DeleteAll(directory, "*");

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void DeleteAll(string directory, string searchPattern) =>
        ↪  DeleteAll(directory, searchPattern, SearchOption.TopDirectoryOnly);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void DeleteAll(string directory, string searchPattern, SearchOption
        ↪  searchOption)
        {
            foreach (var file in Directory.EnumerateFiles(directory, searchPattern,
            ↪  searchOption))
```

```
            {
                File.Delete(file);
            }
        }
    }
}
```

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

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

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

namespace Platform.IO
{
    public static class StreamExtensions
    {
        [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);
        }

        [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;
        }

        [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.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);
        }
    }
}
```

Index