./ConsoleCancellation.cs

```
using System;
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 { get; }

        public CancellationToken Token { get; }

        public bool IsRequested => Source.IsCancellationRequested;

        public bool NotRequested => !Source.IsCancellationRequested;

        public ConsoleCancellation()
        {
            Source = new CancellationTokenSource();
            Token = Source.Token;
            Console.CancelKeyPress += OnCancelKeyPress;
        }

        public void ForceCancellation() => Source.Cancel();

        public void Wait()
        {
            while (NotRequested)
            {
                ThreadHelpers.Sleep();
            }
        }

        protected override void Dispose(bool manual, bool wasDisposed)
        {
            if (!wasDisposed)
            {
                Console.CancelKeyPress -= OnCancelKeyPress;
                Source.DisposeIfPossible();
            }
        }

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

./ConsoleHelpers.cs

```
using System;
using System.Diagnostics;
using Platform.Collections;

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

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

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

        public static string GetOrReadArgument(int index, string readMessage, params string[]
        ↪ args)
```

```
        {
            string result;
            if (args != null && args.Length > index)
            {
                result = args[index];
            }
            else
            {
                Console.Write($"{readMessage}: ");
                result = Console.ReadLine();
            }
            result = (result ?? "").Trim().TrimSingle('"').Trim();
            return result;
        }

        [Conditional("DEBUG")]
        public static void Debug(string format, params object[] args) =>
        ↪  Console.WriteLine(format, args);
    }
}
```

./FileHelpers.cs

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

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

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

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

        public static T ReadFirstOrDefault<T>(string path)
            where T : struct
        {
            using (var fileStream = GetValidFileStreamOrDefault<T>(path))
            {
                return fileStream?.ReadOrDefault<T>() ?? default;
            }
        }

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

        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;
        }

        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>();
            }
        }

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

        public static FileStream Append(string path) => File.Open(path, FileMode.Append,
        ↪  FileAccess.Write);

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

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

./StreamExtensions.cs

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

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

namespace Platform.IO
{
    public static class StreamExtensions
    {
        public static void Write<T>(this Stream stream, T value)
            where T : struct
        {
            var bytes = value.ToBytes();
            stream.Write(bytes, 0, bytes.Length);
        }

        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;
        }

        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;
        }
    }
}
```

Index