LinksPlatform's Platform.Unsafe Class Library

1.1 ./csharp/Platform.Unsafe/ByteArrayExtensions.cs

```csharp
using Platform.Exceptions;
using Platform.Collections;
using System.Runtime.CompilerServices;
using static System.Runtime.CompilerServices.Unsafe;

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

namespace Platform.Unsafe
{
    public unsafe static class ByteArrayExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TStruct ToStructure<TStruct>(this byte[] bytes)
            where TStruct : struct
        {
            Ensure.OnDebug.ArgumentNotEmpty(bytes, nameof(bytes));
            Ensure.OnDebug.ArgumentMeetsCriteria(bytes, HasSameSizeAs<TStruct>, nameof(bytes),
             ↪  "Bytes array should be the same length as struct size.");
            TStruct structure = default;
            fixed (byte* pointer = bytes)
            {
                Copy(ref structure, pointer);
            }
            return structure;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static bool HasSameSizeAs<TStruct>(byte[] array) where TStruct : struct =>
         ↪  array.Length == Structure<TStruct>.Size;
    }
}
```

1.2 ./csharp/Platform.Unsafe/IntPtrExtensions.cs

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

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

namespace Platform.Unsafe
{
    public unsafe static class IntPtrExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void WriteElementValue<TValue>(this IntPtr pointer, long index, TValue
         ↪  value) => Write((byte*)pointer + (SizeOf<TValue>() * index), value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TValue ReadElementValue<TValue>(this IntPtr pointer, long index) =>
         ↪  Read<TValue>((byte*)pointer + (SizeOf<TValue>() * index));
    }
}
```

1.3 ./csharp/Platform.Unsafe/MemoryBlock.cs

```csharp
using System;
using System.Collections.Concurrent;
using System.Runtime.CompilerServices;
using System.Threading.Tasks;
using static System.Runtime.CompilerServices.Unsafe;

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

namespace Platform.Unsafe
{
    public static unsafe class MemoryBlock
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Zero(void* pointer, long capacity)
        {
            // A way to prevent wasting resources due to Hyper-Threading.
            var threads = Environment.ProcessorCount / 2;
            if (threads <= 1)
            {
                ZeroBlock(pointer, 0, capacity);
            }
            else
            {
                // Using 2 threads because two-channel memory architecture is the most available
                 ↪  type.
                // CPUs mostly just wait for memory here.
                threads = 2;
                Parallel.ForEach(Partitioner.Create(0L, capacity), new ParallelOptions {
                 ↪  MaxDegreeOfParallelism = threads }, range => ZeroBlock(pointer, range.Item1,
                 ↪  range.Item2));
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void ZeroBlock(void* pointer, long from, long to)
        {
            var offset = (byte*)pointer + from;
            var length = to - from;
            var uintMaxValue = uint.MaxValue;
            while (length > uintMaxValue)
            {
                InitBlock(offset, 0, uintMaxValue);
                length -= uintMaxValue;
                offset += uintMaxValue;
            }
            InitBlock(offset, 0, unchecked((uint)length));
        }
    }
}
```

1.4 ./csharp/Platform.Unsafe/Structure.cs

```
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using static System.Runtime.CompilerServices.Unsafe;

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

namespace Platform.Unsafe
{
    public static class Structure<TStruct>
        where TStruct : struct
    {
        /// <summary>
        /// <para>
        /// Returns the size of an unmanaged type in bytes.
        /// This property do this without throwing exceptions for generic types as <see
         ↪  cref="Marshal.SizeOf{T}()"/> and <see cref="Marshal.SizeOf(Type)"/> do.
        /// </para>
        /// <para>
        /// Возвращает размер неуправляемого типа в байтах.
        /// Этот свойство делает это без выбрасывания исключений для универсальных типов, как
         ↪  это делают <see cref="Marshal.SizeOf{T}()"/> и <see cref="Marshal.SizeOf(Type)"/>.
        /// </para>
        /// </summary>
        public static int Size
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
        } = SizeOf<TStruct>();
    }
}
```

1.5 ./csharp/Platform.Unsafe/StructureExtensions.cs

```
using System.Runtime.CompilerServices;
using static System.Runtime.CompilerServices.Unsafe;

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

namespace Platform.Unsafe
{
    public unsafe static class StructureExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static byte[] ToBytes<TStruct>(this ref TStruct obj)
            where TStruct : struct
        {
            var bytes = new byte[Structure<TStruct>.Size];
            fixed (byte* pointer = bytes)
            {
                Copy(pointer, ref obj);
            }
            return bytes;
        }
    }
}
```

1.6 ./csharp/Platform.Unsafe.Tests/IntPtrExtensionsTests.cs

```
using System;
using System.Runtime.InteropServices;
using Xunit;
using static System.Runtime.CompilerServices.Unsafe;

namespace Platform.Unsafe.Tests
{
    public unsafe class IntPtrExtensionsTests
    {
        [Fact]
        public void ReadAndWriteOperationsForPointerValuesUnsafeClassMethodsTest()
        {
            void* pointer = (void*)Marshal.AllocHGlobal(sizeof(ulong));
            Write(pointer, 42UL);
            Assert.Equal(42UL, Read<ulong>(pointer));
            Marshal.FreeHGlobal((IntPtr)pointer);
        }

        [Fact]
        public void ElementOffsetOperationsForPointerValuesTest()
        {
            void* pointer = (void*)Marshal.AllocHGlobal(sizeof(ulong) * 10);
            ulong result = (ulong)Add<ulong>(pointer, 5);
            Assert.Equal(5UL * 8UL, result - (ulong)pointer);
            Marshal.FreeHGlobal((IntPtr)pointer);
        }
    }
}
```

1.7 ./csharp/Platform.Unsafe.Tests/SizeOfTests.cs

```
using System.Runtime.InteropServices;
using Xunit;

namespace Platform.Unsafe.Tests
{
    public static class SizeOfTests
    {
        public struct X<T>
        {
            public readonly T F1;
            public readonly T F2;
        }

        [Fact]
        public static void UnsafeClassSizeOfTest()
        {
            var size = System.Runtime.CompilerServices.Unsafe.SizeOf<X<int>>();
            Assert.Equal(8, size);
        }

        [Fact]
        public static void MarshalSizeOfTest()
        {
            var size = Marshal.SizeOf(default(X<int>));
            Assert.Equal(8, size);
        }

        [Fact]
        public static void StructurePropertyTest()
        {
            var size = Structure<X<int>>.Size;
            Assert.Equal(8, size);
        }
    }
}
```

1.8 ./csharp/Platform.Unsafe.Tests/StructAndBytesConversionTests.cs

```
using Xunit;

namespace Platform.Unsafe.Tests
{
    public static class StructAndBytesConversionTests
    {
        [Fact]
        public static void StructToBytesTest()
        {
            ulong source = ulong.MaxValue;
            var result = source.ToBytes();
            for (int i = 0; i < result.Length; i++)
            {
                Assert.Equal(byte.MaxValue, result[i]);
            }
        }

        [Fact]
        public static void BytesToStructTest()
        {
            byte[] bytes = new[] { byte.MaxValue, byte.MaxValue, byte.MaxValue, byte.MaxValue,
             ↪  byte.MaxValue, byte.MaxValue, byte.MaxValue, byte.MaxValue };
            ulong result = bytes.ToStructure<ulong>();
            Assert.Equal(ulong.MaxValue, result);
        }
    }
}
```

1.9 ./csharp/Platform.Unsafe.Tests/ZeroMemoryTests.cs

```
using Xunit;

namespace Platform.Unsafe.Tests
{
    public static unsafe class ZeroMemoryTests
    {
        [Fact]
        public static void ZeroMemoryTest()
        {
            var bytes = new byte[1024];
            for (int i = 0; i < bytes.Length; i++)
            {
                bytes[i] = unchecked((byte)i);
            }
            fixed (byte* pointer = bytes)
            {
                MemoryBlock.Zero(pointer, bytes.Length);
            }
            for (int i = 0; i < bytes.Length; i++)
            {
                Assert.Equal(0, bytes[i]);
            }
        }
    }
}
```

Index

./csharp/Platform.Unsafe.Tests/IntPtrExtensionsTests.cs, 3
./csharp/Platform.Unsafe.Tests/SizeOfTests.cs, 3
./csharp/Platform.Unsafe.Tests/StructAndBytesConversionTests.cs, 3
./csharp/Platform.Unsafe.Tests/ZeroMemoryTests.cs, 4
./csharp/Platform.Unsafe/ByteArrayExtensions.cs, 1
./csharp/Platform.Unsafe/IntPtrExtensions.cs, 1
./csharp/Platform.Unsafe/MemoryBlock.cs, 1
./csharp/Platform.Unsafe/Structure.cs, 2
./csharp/Platform.Unsafe/StructureExtensions.cs, 2