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
            {
```

```csharp
                    // 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

```csharp
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

```csharp
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
{
    /// <summary>
    /// <para>defines code as unsafe</para>
    /// <para>Определяет код, как небезопасный</para>
    /// </summary>
    public unsafe static class StructureExtensions
    {
        /// <summary>
        /// <para>this process does something</para>
        /// <para>этот процесс что-то делает</para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
19          public static byte[] ToBytes<TStruct>(this ref TStruct obj)
20              where TStruct : struct
21          {
22              var bytes = new byte[Structure<TStruct>.Size];
23              fixed (byte* pointer = bytes)
24              {
25                  Copy(pointer, ref obj);
26              }
27              return bytes;
28          }
29      }
30  }
```

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

```
1   using System;
2   using System.Runtime.InteropServices;
3   using Xunit;
4   using static System.Runtime.CompilerServices.Unsafe;
5
6   namespace Platform.Unsafe.Tests
7   {
8       public unsafe class IntPtrExtensionsTests
9       {
10          [Fact]
11          public void ReadAndWriteOperationsForPointerValuesUnsafeClassMethodsTest()
12          {
13              void* pointer = (void*)Marshal.AllocHGlobal(sizeof(ulong));
14              Write(pointer, 42UL);
15              Assert.Equal(42UL, Read<ulong>(pointer));
16              Marshal.FreeHGlobal((IntPtr)pointer);
17          }
18
19          [Fact]
20          public void ElementOffsetOperationsForPointerValuesTest()
21          {
22              void* pointer = (void*)Marshal.AllocHGlobal(sizeof(ulong) * 10);
23              ulong result = (ulong)Add<ulong>(pointer, 5);
24              Assert.Equal(5UL * 8UL, result - (ulong)pointer);
25              Marshal.FreeHGlobal((IntPtr)pointer);
26          }
27      }
28  }
```

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

```
1   using System.Runtime.InteropServices;
2   using Xunit;
3
4   namespace Platform.Unsafe.Tests
5   {
6       public static class SizeOfTests
7       {
8           public struct X<T>
9           {
10              public readonly T F1;
11              public readonly T F2;
12          }
13
14          [Fact]
15          public static void UnsafeClassSizeOfTest()
16          {
17              var size = System.Runtime.CompilerServices.Unsafe.SizeOf<X<int>>();
18              Assert.Equal(8, size);
19          }
20
21          [Fact]
22          public static void MarshalSizeOfTest()
23          {
24              var size = Marshal.SizeOf(default(X<int>));
25              Assert.Equal(8, size);
26          }
27
28          [Fact]
29          public static void StructurePropertyTest()
30          {
31              var size = Structure<X<int>>.Size;
32              Assert.Equal(8, size);
33          }
34      }
35  }
```

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

```csharp
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

```csharp
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