LinksPlatform's Platform.Unsafe Class Library

1.1 ./Platform.Unsafe/ByteArrayExtensions.cs

```
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));
            var structureSize = SizeOf<TStruct>();
            Ensure.OnDebug.ArgumentMeetsCriteria(bytes, array => array.Length == structureSize,
             ↪  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;
        }
    }
}
```

1.2 ./Platform.Unsafe/IntPtr.cs

```
using System;
using System.Reflection;
using System.Runtime.InteropServices;
using Platform.Reflection;

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

namespace Platform.Unsafe
{
    /// <remarks>
    /// Please use System.Runtime.CompilerServices.Unsafe instead.
    /// </remarks>
    [Obsolete("Please use System.Runtime.CompilerServices.Unsafe instead.")]
    public static class IntPtr<T>
    {
        public static readonly Func<IntPtr, T> GetValue = CompileGetValueDelegate();
        public static readonly Action<IntPtr, T> SetValue = CompileSetValueDelegate();

        static private Func<IntPtr, T> CompileGetValueDelegate()
        {
            return DelegateHelpers.Compile<Func<IntPtr, T>>(emiter =>
            {
                if (NumericType<T>.IsNumeric)
                {
                    emiter.LoadArgument(0);
                    emiter.LoadIndirect<T>();
                    emiter.Return();
                }
                else
                {
                    emiter.LoadArguments(0);
                    emiter.Call(typeof(Marshal).GetGenericMethod(nameof(Marshal.PtrToStructure),
                     ↪  Types<T>.Array, Types<IntPtr, Type, bool>.Array));
                    emiter.Return();
                }
            });
        }

        static private Action<IntPtr, T> CompileSetValueDelegate()
        {
            return DelegateHelpers.Compile<Action<IntPtr, T>>(emiter =>
            {
                if (NumericType<T>.IsNumeric)
                {
                    emiter.LoadArguments(0, 1);
                    emiter.StoreIndirect<T>();
                    emiter.Return();
                }
                else
                {
                    emiter.LoadArguments(0, 1);
                    emiter.LoadConstant(true);
                    emiter.Call(typeof(Marshal).GetTypeInfo().GetMethod(nameof(Marshal.Structure
                    ↪  ToPtr), Types<object, IntPtr,
                    ↪  bool>.Array));
                    emiter.Return();
                }
            });
        }
    }
}
```

1.3 ./Platform.Unsafe/IntPtrExtensions.cs

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

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

namespace Platform.Unsafe
{
    /// <remarks>
    /// Please use System.Runtime.CompilerServices.Unsafe instead.
    /// </remarks>
    public unsafe static class IntPtrExtensions
    {
        [Obsolete("GetValue method is deprecated, please use
    ↪  System.Runtime.CompilerServices.Unsafe.Read method instead.")]
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TElement GetValue<TElement>(this IntPtr pointer) =>
        ↪  IntPtr<TElement>.GetValue(pointer);

        [Obsolete("SetValue method is deprecated, please use
    ↪  System.Runtime.CompilerServices.Unsafe.Write method instead.")]
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void SetValue<TElement>(this IntPtr pointer, TElement value) =>
        ↪  IntPtr<TElement>.SetValue(pointer, value);

        [Obsolete("GetElement method is deprecated, please use
    ↪  System.Runtime.CompilerServices.Unsafe.Add method instead.")]
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IntPtr GetElement(this IntPtr pointer, int elementSize, int index) =>
        ↪  pointer + (elementSize * index);

        [Obsolete("GetElement method is deprecated, please use
    ↪  System.Runtime.CompilerServices.Unsafe.Add method instead.")]
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IntPtr GetElement(this IntPtr pointer, long elementSize, long index) =>
        ↪  new IntPtr((byte*)pointer.ToPointer() + (elementSize * index));

        [Obsolete("GetElement method is deprecated, please use
    ↪  System.Runtime.CompilerServices.Unsafe.Add method instead.")]
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IntPtr GetElement<TIndex>(this IntPtr pointer, int elementSize, TIndex
        ↪  index) => pointer.GetElement((long)elementSize, (Integer)(Integer<TIndex>)index);

        [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));
    }
}
```

1.4 ./Platform.Unsafe/MemoryBlock.cs

```
using System.Collections.Concurrent;
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
    {
        public static void Zero(void* pointer, long capacity)
        {
            Parallel.ForEach(Partitioner.Create(0, capacity), range =>
            {
                var from = range.Item1;
                var offset = (void*)((byte*)pointer + from);
                var length = (uint)(range.Item2 - from);
                InitBlock(offset, 0, length);
            });
        }
    }
}
```

1.5 ./Platform.Unsafe/Structure.cs

```
using System;
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 { get; } = SizeOf<TStruct>();
    }
}
```

1.6 ./Platform.Unsafe/StructureExtensions.cs

```
using System.Runtime.CompilerServices;
using Platform.Hardware.Cpu;
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 structureSize = SizeOf<TStruct>();
            var bytes = new byte[structureSize];
            fixed (byte* pointer = bytes)
            {
                obj.CopyTo(pointer, structureSize);
            }
            return bytes;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CopyTo<TStruct>(this ref TStruct source, void* destination)
            where TStruct : struct
        {
            var size = SizeOf<TStruct>();
            CopyTo(ref source, destination, size);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void CopyTo<TStruct>(this ref TStruct source, void* destination, int size)
            where TStruct : struct
        {
            if (CacheLine.Size >= size)
            {
                Copy(destination, ref source);
            }
            else
            {
                CopyBlock(destination, AsPointer(ref source), (uint)size);
            }
        }
    }
}
```

1.7 ./Platform.Unsafe.Tests/IntPtrExtensionsTests.cs

```
using System;
using System.Runtime.InteropServices;
using Xunit;
using Xunit.Abstractions;
using Platform.Diagnostics;

namespace Platform.Unsafe.Tests
{
    public unsafe class IntPtrExtensionsTests
    {
        private const int N = 10000000;

        private readonly ITestOutputHelper _output;

        public IntPtrExtensionsTests(ITestOutputHelper output)
        {
            _output = output;
        }

        [Fact]
        public void ReadAndWriteOperationsForPointerValuesDelegatesTest()
        {
            var pointer = Marshal.AllocHGlobal(sizeof(ulong));
            ulong result = default;
            for (var i = 0; i < N; i++)
            {
                result = Delegates(pointer);
            }
            Assert.Equal(42UL, result);
            Marshal.FreeHGlobal(pointer);
        }

        private static ulong Delegates(IntPtr pointer)
        {
            ulong result;
            //IntPtr<ulong>.SetValue(pointer, 42UL);
            System.Runtime.CompilerServices.Unsafe.Write((void*)pointer, 42UL);
            //result = IntPtr<ulong>.GetValue(pointer);
            result = System.Runtime.CompilerServices.Unsafe.Read<ulong>((void*)pointer);
            return result;
        }

        [Fact]
        public void ReadAndWriteOperationsForPointerValuesExtensionMethodsTest()
        {
            var pointer = Marshal.AllocHGlobal(sizeof(ulong));
            ulong result = default;
            for (var i = 0; i < N; i++)
            {
                result = ExtensionMethods(pointer);
            }
            Assert.Equal(42UL, result);
            Marshal.FreeHGlobal(pointer);
        }

        private static ulong ExtensionMethods(IntPtr pointer)
        {
            ulong result;
            //pointer.SetValue(42UL);
            System.Runtime.CompilerServices.Unsafe.Write((void*)pointer, 42UL);
            //result = pointer.GetValue<ulong>();
            result = System.Runtime.CompilerServices.Unsafe.Read<ulong>((void*)pointer);
            return result;
        }

        [Fact]
        public void ReadAndWriteOperationsForPointerValuesUnsafeClassMethodsTest()
        {
            void* pointer = (void*)Marshal.AllocHGlobal(sizeof(ulong));
            ulong result = default;
            for (var i = 0; i < N; i++)
            {
                result = ReadAndWriteMethods(pointer);
            }
            Assert.Equal(42UL, result);
            Marshal.FreeHGlobal((IntPtr)pointer);
        }

        private static ulong ReadAndWriteMethods(void* pointer)
        {
            ulong result;
            System.Runtime.CompilerServices.Unsafe.Write(pointer, 42UL);
            result = System.Runtime.CompilerServices.Unsafe.Read<ulong>(pointer);
            return result;
        }

        [Fact]
        public void ReadAndWriteOperationsComparisionTest()
        {
            var t1 = Performance.Measure(ReadAndWriteOperationsForPointerValuesDelegatesTest);
            var t2 =
             ↪  Performance.Measure(ReadAndWriteOperationsForPointerValuesExtensionMethodsTest);
            var t3 = Performance.Measure(ReadAndWriteOperationsForPointerValuesUnsafeClassMethod⌟
             ↪  sTest);
            var message = $"{t1} {t2} {t3}";
            _output.WriteLine(message);
        }

        [Fact]
        public void ElementOffsetOperationsForPointerValuesExtensionMethods()
        {
            var pointer = Marshal.AllocHGlobal(sizeof(ulong) * 10);
            ulong result = default;
            for (var i = 0; i < N; i++)
            {
                result = GetElementExtensionMethods(pointer);
            }
            Assert.Equal(5UL * 8UL, result - (ulong)pointer);
            Marshal.FreeHGlobal(pointer);
        }

        private static ulong GetElementExtensionMethods(IntPtr pointer)
        {
            ulong result;
            //result = (ulong)pointer.GetElement(8, 5);
            result = (ulong)System.Runtime.CompilerServices.Unsafe.Add<ulong>((void*)pointer, 5);
            return result;
        }

        [Fact]
        public void ElementOffsetOperationsForPointerValuesUnsafeClassMethodsTest()
        {
            void* pointer = (void*)Marshal.AllocHGlobal(sizeof(ulong) * 10);
            ulong result = default;
            for (var i = 0; i < N; i++)
            {
                result = GetElementMethods(pointer);
            }
            Assert.Equal(5UL * 8UL, result - (ulong)pointer);
            Marshal.FreeHGlobal((IntPtr)pointer);
        }

        private static ulong GetElementMethods(void* pointer)
        {
            ulong result;
            result = (ulong)System.Runtime.CompilerServices.Unsafe.Add<ulong>(pointer, 5);
            return result;
        }

        [Fact]
        public void GetElementOperationsComparisionTest()
        {
            var t1 =
             ↪  Performance.Measure(ElementOffsetOperationsForPointerValuesExtensionMethods);
            var t2 = Performance.Measure(ElementOffsetOperationsForPointerValuesUnsafeClassMetho
            ↪  dsTest);
            var message = $"{t1} {t2}";
            _output.WriteLine(message);
        }
    }
}
```

1.8 ./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);
        }
    }
}
```

1.9 ./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);
        }
    }
}
```

1.10 ./Platform.Unsafe.Tests/ZeroMemoryTests.cs

```
using Xunit;

namespace Platform.Unsafe.Tests
```

```csharp
{
    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