Runtime Monitoring

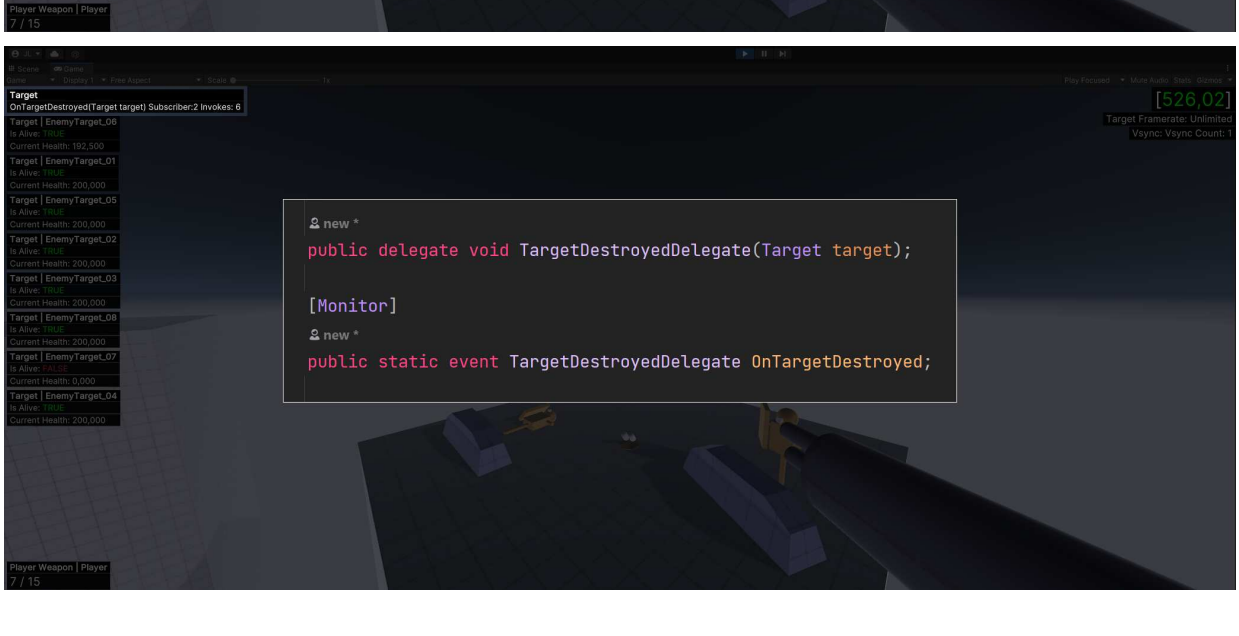
Runtime Monitoring is an easy way for you to monitor the state of your C# classes and objects during runtime. Just add the 'Monitor' attribute to a field, property or event and get its value or state displayed automatically in a customizable and extendable UI.

There are still some aspects I would like to improve or expand (see Planned Features). Especially the optional packages for supporting TextMeshPro & UIToolkit need more work and fine-tuning. However, I can't afford too much time commitment due to my full-time job and other projects I'm working on. For this reason, I would appreciate any feedback and/or support.

Table of Contents

Getting Started

```csharp
// Place the MonitorAttribute on any field, property or event
// to have it automatically displayed during runtime in you UI.
[Monitor]
private int healthPoints;

[Monitor]
public int HealthPoints { get; private set; }

[Monitor]
public event Action OnHealthChanged;

[Monitor]
public static string playerName;

[Monitor]
protected static bool IsPlayerAlive { get; set; }

[Monitor]
internal static event Action<int> OnScoreChanged;

// Determine if and in what quantity the state will be evaluated.
[MonitorField(Update = UpdateOptions.FrameUpdate)]
private float speed;

// Reduce update overhead by providing an update event.
[MonitorProperty(UpdateEvent = nameof(OnPlayerSpawn))]
public bool LastSpawnPosition { get; set; }

[MonitorEvent]
public static event Action<Vector3> OnPlayerSpawn;

// Monitored events display their signature, subscriber count and invokation count.
// These options can be toggled using the MonitorEventAttribute.
[MonitorEvent(ShowSignature = false, ShowSubscriber = true)]
public event OnGameStart;

// Use processor methods to customize how the value is displayed.
[Monitor]
[ValueProcessor(nameof(IsAliveProcessor))]
public bool IsAlive { get; private set; }

private string IsAliveProcessor(bool value) => value? "Alive" : "Dead";

// Register & unregister objects with members you want to monitor.
// This process can be simplified / automated (Take a look at Monitoring Objects)
public class Player : MonoBehaviour
{
    [Monitor]
    private int healthPoints;

    private void Awake()
    {
        MonitoringManager.RegisterTarget(this);
    }

    private void OnDestroy()
    {
        MonitoringManager.UnregisterTarget(this);
    }
}
```



Technical Information

- Unity Version: **2019.4** (for UIToolkit **2020.1**)
- Scripting Backend: **Mono & IL2CPP**
- API Compatibility: **.NET Standard 2.0 or .NET 4.xP**
- Asset Version: **1.0.7**

Import

Import this asset into your project as a .unitypackage available at Runtime-Monitoring/releases or clone this repository and use it directly.

Depending on your needs you may select or deselect individual modules when importing. `Monitoring Example` contains an optional example scene and Monitoring UI contains UI / Display preset based on different UI Systems. Some packages can be imported retroactively using the setup window or by just importing their included packages located at Baracuda/Monitoring.UI

Assembly	Path	Editor	Core
Assembly-Baracuda-Monitoring	Baracuda/Monitoring		✔
Assembly-Baracuda-Editor	Baracuda/Monitoring.Editor	✔	✔
Assembly-Baracuda-Example	Baracuda/Monitoring.Example		
Assembly-Baracuda-Monitoring.GUI	Baracuda/Monitoring.UI/UnityGUI		✔
Assembly-Baracuda-Monitoring.UITookit	Baracuda/Monitoring.UI/UIToolkit		
Assembly-Baracuda-Monitoring.TextMeshPro	Baracuda/Monitoring.UI/TextMeshPro		
Assembly-Baracuda-Pooling	Baracuda/Pooling		✔
Assembly-Baracuda-Threading	Baracuda/Threading		✔
Assembly-Baracuda-Reflection	Baracuda/Reflection		✔

Setup

Everything should work out of the box after a successful import. However, if you want to validate that everything is set up correctly or you want to change for example the active Monitoring UI Controller, the following steps will guide you through that process.

- Open the settings by navigating to (menu: Tools > RuntimeMonitoring > Settings).
- Ensure that both `Enable Monitoring` and `Open Display On Load` are set to `true`.
- If `Enable Monitoring` in the UI Controller foldout is set to `false`, Make sure to call `MonitoringUI.CreateMonitoringUI()` from anywhere in you code.
- Open the setup window (menu: Tools > RuntimeMonitoring > Setup) to import optional UIController packages (recommended).
- Use the `Monitoring UI Controller` field in the UI Controller foldout to set the active UI Controller. The inspector of the set UI Controller object will be inlined and can be edited from the settings window. As of version 1.0.7 available UIController in your project will be listed and can be selected by pressing their corresponding select button.

Monitoring Objects

When monitoring non static member of a class, instances of those classes must be registered when they are created and unregistered when they are destoryed. This process can be automated or simplified, by inheriting from one of the following base types.

- `MonitoredBehaviour`: an automatically monitored `MonoBehaviour`
- `MonitoredSingleton<T>`: an automatically monitored `MonoBehaviour` singleton.
- `MonitoredScriptableObject`: an automatically monitored `ScriptableObject`.
- `MonitoredObject`: an automatically monitored `System.Object`. that implements the `IDisposable` interface. Please make sure to call `Disposable` on those objects when you no longer need them.

```csharp
public class Player : MonoBehaviour
{
    [Monitor]
    private int healthPoints;

    private void Awake()
    {
        MonitoringManager.RegisterTarget(this);
        // Or use the extension method:
        this.RegisterMonitor();
    }

    private void OnDestroy()
    {
        MonitoringManager.UnregisterTarget(this);
        // Or use the extension method:
        this.UnregisterMonitor();
    }
}

// Simplified by inheriting from MonitoredBehaviour.
public class Player : MonitoredBehaviour
{
    [Monitor]
    private int healthPoints;
}

// Just Remember to call base.Awake and base.OnDestroy if you override these methods.
public class Player : MonitoredBehaviour
{
    [Monitor]
    private int healthPoints;

    protected override void Awake()
    {
        base.Awake();
        // Your Awake code.
    }

    protected override void OnDestroy()
    {
        base.OnDestroy();
        // Your OnDestroy code.
    }
}
```

Value Processor

You can add the ValueProcessorAttribute to a monitored field or porperty to gain more controll of its string representation. Use the attibute to pass the name of a method that will be used to parse the current value to a string.

The value processor method must accept a value of the monitored members type, can be both static and non static (when monitoring a non non static member) and must return a string.

```csharp
[ValueProcessor(nameof(IsAliveProcessor))]
[Monitor]
private bool isAlive;

private string IsAliveProcessor(bool isAliveValue)
{
    return isAliveValue ? "Player is Alive" : "Player is Dead!";
}
```

```csharp
[ValueProcessor(nameof(IListProcessor))]
[Monitor] private IList<string> names = new string[] {"Gordon", "Alyx", "Barney"};

private string IListProcessor(IList<string> elements)
{
    var str = string.Empty;
    foreach (var name in elements)
    {
        str += name;
        str += "\n";
    }
    return str;
}
```

Static processor methods can have certain overloads for objects that impliment generic collection interfaces, which allow you to process the value of individual elements of the collection instead of the whole collection all at once.

```csharp
//IList<T> ValueProcessor
[ValueProcessor(nameof(IListProcessor))]
[Monitor] private IList<string> names = new string[] {"Gordon", "Alyx", "Barney"};

private static string IListProcessor(string element)
{
    return $"The name is {element}";
}


[ValueProcessor(nameof(IListProcessorWithIndex))]
[Monitor] private IList<string> Names => names;

private static string IListProcessorWithIndex(string element, int index)
{
    return $"The name at index {index} is {element}";
}
```

```csharp
//IDictionary<TKey, TValue> ValueProcessor
[ValueProcessor(nameof(IDictionaryProcessor))]
[Monitor] private IDictionary<string, bool> isAliveDictionary = new Dictionary<string, b
{
    {"Bondrewd", true},
    {"Lyza", false}
};

private static string IDictionaryProcessor(string name, bool isAlive)
{
    return $"{name} is {(isAlive ? "alive" : "dead")}";
}
```

```csharp
//IEnumerable<T> ValueProcessor
[ValueProcessor(nameof(IEnumerableValueProcessor))]
[Monitor]
private IEnumerable<int> randomNumbers = new List<int>
{
    1, 43, 14, 65, 23, 174, 16, 2, 786, 4, 89
};

private static string IEnumerableValueProcessor(int number)
{
    return $"{number} is {((number & 1) == 0? "Even" : "Odd")}";
}
```

Update Loop

Monitord member are evaluated in an update loop. You can provide an event that will tell monitoring that a value has changed to remove it from the update loop.

```csharp
public enum UpdateOptions
{
    Auto = 0,
    DontUpdate = 1,
    FrameUpdate = 2,
    TickUpdate = 4,
}
```

- `UpdateOtions.Auto`: If an update event is set, the state of the members will only be evaluated when the event is invoked. Otherwise Tick is the preferred update interval.
- `UpdateOtions.DontUpdate`: The members will not be evaluated except once on load. Use this option for constant values.
- `UpdateOtions.FrameUpdate`: The member will be evaluated on every LateUpdate.
- `UpdateOtions.TickUpdate`: The member will be evaluated on every Tick. Tick is a custom update cycle that is roughly called 30 times per second.

Update Event

When monitoring a field or a property (Value units) you can provide an 'OnValueChanged' event that will tell the monitored unit that the state of the member has changed.

This event can either be an `Action` or an `Action<T>`, with T being the type of the monitored field or property. Note that once a valid update event was provided the unit will not be evaluated during an update cycle anymore, unless `UpdateOptions` are explicitly set to `UpdateOptions.Auto` or `UpdateOptions.FrameUpdate`.

Passing an event will slightly reduce performance overhead for values or member that you know will update rarely. It is however not required.

```csharp
private int healthPoints;
public event Action<int> OnHealthChanged;

[Monitor(UpdateEvent = nameof(OnHealthChanged))]
public int HealthPoints
{
    get => healthPoints;
    private set
    {
        healthPoints = value;
        OnHealthChanged?.Invoke(healthPoints);
    }
}
```

```csharp
[Monitor(UpdateEvent = nameof(OnGameStateChanged))]
private bool isGamePaused;

public event Action OnGameStateChanged;

public void PauseGame()
{
    isGamePaused = true;
    OnGameStateChanged?.Invoke();
}

public void ContinueGame()
{
    isGamePaused = false;
    OnGameStateChanged?.Invoke();
}
```

Runtime

The true purpose of this tool is to provide an easy way to debug and monitor build games. Both Mono & IL2CPP runtimes are supported. Mono runtime works without any limitations.

IL2CPP

Monitoring is making extensive use of dynamic type & method creation during its initialization process. This means that the IL2CPP runtime has a hard time because it requires AOT compilation (Ahead of time compilation)

In order to use IL2CPP as a runtime some features are disabled or reduced and some types must be generated during a build process, that can then be used by the IL2CPP runtime as templates. You can configure the IL2CPP AOT type generation from the monitoring settings.

UI Controller

Use the `MonitoringUI` API to toggle the visiblity or active state of the current monitoring UI overlay. `MonitoringUI` is an accesspoint and the bridge between custom code and the active `MonitoringUIController`. This is to offer a layer of abstraction that enables you to switch between multiple either prefabricated or custom UI implimentations / UI Controller.

Note! Not every existing UI controllers (UIToolkit, TextMeshPro and GUI) includes every feature. I would recommend unsing the UIToolkit UI solution if possible.

```csharp
using Baracuda.Monitoring.API;

// Show the monitoring UI overlay.
MonitoringUI.Show();

// Hide the monitoring UI overlay.
MonitoringUI.Hide();

// Toggle the visibility of the active monitoring display.
// This method returns a bool indicating the new visibility state.
MonitoringUI.ToggleDisplay();

// Returns true if the there is an active monitoring display that is also visible.
MonitoringUI.IsVisible();
```

Custom UI Controller

You can create a custom UI controller by follwing the steps below. You can take a look at the existing UI Controller implimentations to get some reference.

- Create a new class and inherit from `MonitoringDisplayController`.
- Impliment the abstract mehtods and create custom UI logic.
- Add the script to a new GameObject and create a prefab of it.
- Make sure to delete the GameObject from your scene.
- Open the settings by navigating to (menu: Tools > Monitoring > Settings).
- Set your prefab as the active controller in the `Moniotoring UI Controller` field.

Assemblies and Modules

Runtime Monitoring is separated into multiple assemblies / modules. Some of those modules are essential while others are not.

Assembly	Path	Core	Note
Assembly-Baracuda-Monitoring	Baracuda/Monitoring	✔	
Assembly-Baracuda-Editor	Baracuda/Monitoring.Editor	✔	Editor
Assembly-Baracuda-Example	Baracuda/Monitoring.Example		
Assembly-Baracuda-Monitoring.GUI	Baracuda/Monitoring.UI/UnityGUI	✔	Default UI
Assembly-Baracuda-Monitoring.UITookit	Baracuda/Monitoring.UI/UIToolkit		Unity 2020.1 or newer
Assembly-Baracuda-Monitoring.TextMeshPro	Baracuda/Monitoring.UI/TextMeshPro		TMP Required
Assembly-Baracuda-Pooling	Baracuda/Pooling	✔	
Assembly-Baracuda-Threading	Baracuda/Threading	✔	Thread Dispatcher
Assembly-Baracuda-Reflection	Baracuda/Reflection	✔	

Miscellaneous

- Use the #define `DISABLE_MONITORING` to disable the internal logic of the tool. Public API will still compile so you don't have to wrap your API calls in a custom #if !DISABLE_MONITORING block.

Planned Features

I would appreciate any help in completing and improving this tool and its features. Feel free to contact me if you have any feedback, suggestions or questions.

- Improved UIController for TextMeshPro & UIToolkit
- Filtering & Grouping (The current UI implimentation is relatively simple and won't work well for huge amounts of monitored members. I would like to address this issue by adding multiple tabs/groups as well as a simple way to filter displayed member.)
- Method monitoring (Properties can be used as a workaround)
- Class scoped monitoring
- Class / object monitoring (Properties returning ToString() can be used as a workaround)
- Improved IL2CPP support / AOT generation.
- Add the option for synchronous profiling.
- Custom update / evaluation loops or more control over the `Tick` loop.

Support Me

I spend a lot of time working on this and other free assets to make sure as many people as possible can use my tools regardless of their financial status. Any kind of support I get helps me keep doing this, so consider leaving a star ⭐ making a donation or follow me on my socials to support me ❤️

- Donation (PayPal.me)
- Linktree
- Twitter
- Itch