

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

MIT license

226 stars 7 forks

Star

Unwatch

Code Issues 1 Pull requests Actions Projects Wiki Security Insights Settings

main



JohnBaracuda Updated Packages ...

14 minutes ago 205

[View code](#)

README.md



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, event or method and get its value or state displayed automatically in a customizable and extendable UI.

Also available on the Asset Store!

```
public class PlayerMovement : MonitoredBehaviour
{
    [Monitor]
    private bool _isJumping;

    [Monitor]
    public Vector3 Velocity => _velocity;

    [Monitor]
    private event Action OnPlayerJump;
}
```

The screenshot shows a Unity Editor window displaying a C# script named 'PlayerMovement'. The script is a subclass of 'MonitoredBehaviour'. It contains three fields annotated with the '[Monitor]' attribute: '_isJumping' (a private bool), 'Velocity' (a public Vector3), and 'OnPlayerJump' (a private event). The Unity interface includes a code editor, a timeline at the bottom, and various toolbars and status bars.

Table of Contents

- Getting started
 - Import & Update
 - Setup
 - License
 - Technical Information
 - Feature List
- Monitoring Member
 - Instanced & Static Member
 - Monitoring Fields & Properties
 - Monitoring Events
 - Monitoring Methods
- Attributes
 - Value Processor
 - Value Processor (Global)
 - Conditional Display
 - Update Event
- Monitoring UI
 - UI Formatting
 - UI Filtering
 - Custom UI Controller
- Systems and API
 - Monitoring Manager
 - Monitoring UI
 - Monitoring Settings
 - Monitoring Utility
 - Monitor Unit
 - Monitor Profile
- Compatibility
 - Runtime
 - Platform
- Optimizations
- FAQ: Frequently Asked Questions
- Support Me ❤️

Getting Started

```
// Monitor any field, property, event or method during runtime!

[Monitor]
private int healthPoints;

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

[Monitor]
public int GetHealthPoints() => healthPoints;

[Monitor]
public event Action OnHealthChanged;
```

```

// Monitor static member as well as instance member

[Monitor]
public static string playerName;

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

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

// Use conditions to determine if a member is displayed or not.

[Monitor]
[MShowIf(Condition.CollectionNotEmpty)]
private Stack<string> errorMessages { get; }

// Reduce update overhead by providing an update event.

[Monitor]
[MUpdateEvent(nameof(OnPlayerSpawn))]
public Vector3 LastSpawnPosition { get; set; }

public static event Action<Vector3> OnPlayerSpawn;

// Use processor methods to customize how the value is displayed.

[Monitor]
[MValueProcessor(nameof(IsAliveProcessor))]
public bool IsAlive { get; private set; }

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

// Monitor out parameter value.

[MonitorMethod]
public bool TryGetPlayer(int playerId, out var player)
{
    // ...
}

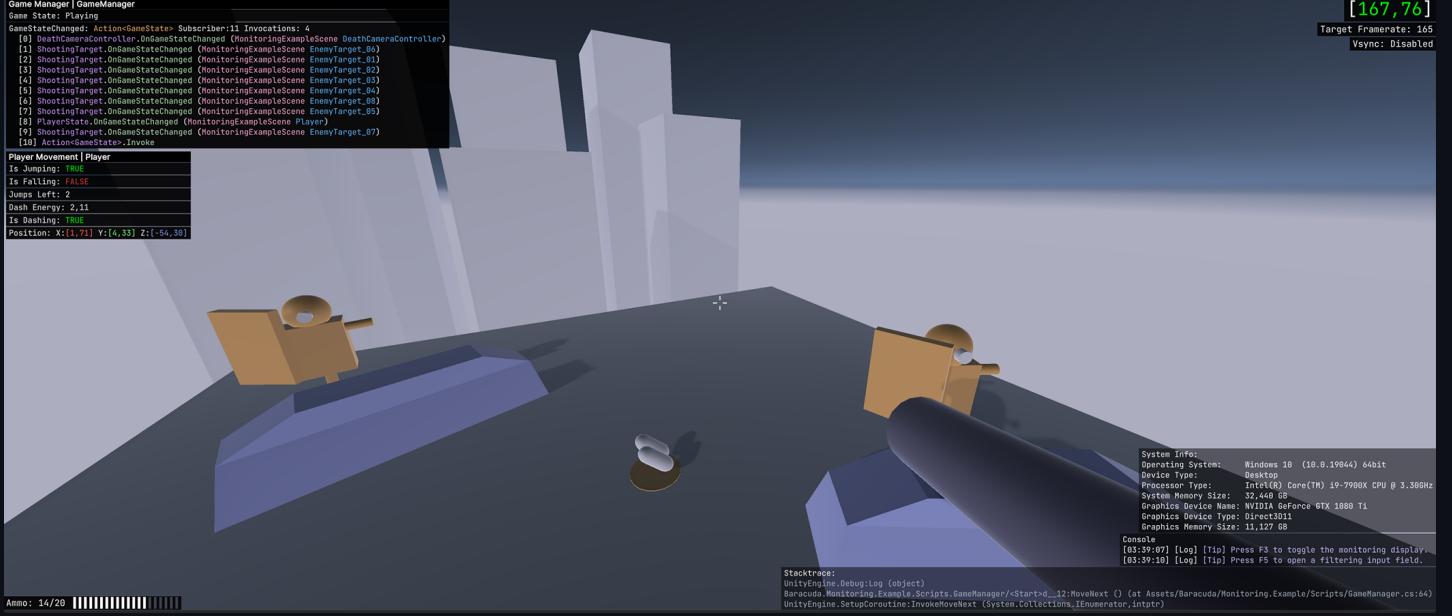
// Register & unregister objects with members you want to monitor.
// This process can be simplified / automated (Take a look at Monitoring Objects)

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

    private void Awake()
    {
        MonitoringSystems.Resolve<IMonitoringManager>().RegisterTarget(this);
    }

    private void OnDestroy()
    {
        MonitoringSystems.Resolve<IMonitoringManager>().UnregisterTarget(this);
    }
}

```



Import and Update

Import this asset into your project as a .unitypackage available at [Runtime-Monitoring/releases](#) or clone this repository and use it directly. You can also download this asset from the [Asset Store!](#). Take a look at [Setup](#) instructions for more information how to import optional packages. (spoiler: via the settings window)

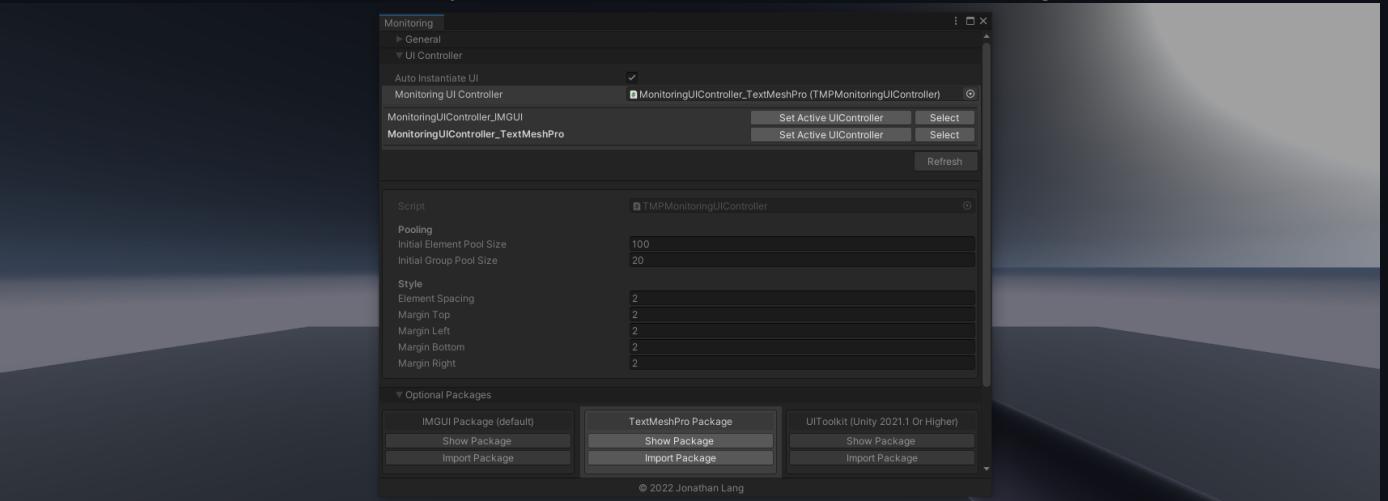
⚠ Important for version updates!

When updating from an older version of this tool it is recommended to perform a clean update. Delete the Baracuda folder and import this asset again. Be aware that this may temporarily cause compile errors. To maintain compatibility when upgrading from an older version multiple files are contained in a special assembly named Assembly-Baracuda-Obsolete. This assembly and all of its contents can be deleted after a successful import. Obsolete files are located at (Baracuda/Monitoring/Internal) & (Baracuda/Monitoring/Interface)

Setup

Download and import Runtime Monitoring. To setup a different UI Controller (IMGUI, TMP or UI Toolkit) follow these optional steps:

- Open the settings by navigating to (menu: Tools > RuntimeMonitoring > Settings).
- Depending on the Unity version and your preferences, import and optional UIController package.
- Use the **Monitoring UI Controller** field in the UI Controller foldout or use the **Set Active UIController** button on a listed element 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.



License

MIT You can use this tool for anything you want, including commercial products, as long as you're not just selling my work or using it for some other morally questionable or condemnable act.

Technical Information

- Unity Version: **2019.4** (for UI Toolkit **2020.1**)
- Scripting Backend: **Mono & IL2CPP**
- API Compatibility: **.NET Standard 2.0 or .NET 4.x**
- Asset Version: **2.0.1**

Features

- Monitor the value of a Field.
- Monitor the return value of a Property.
- Monitor the state of an Event.
- Monitor the return value & out parameter of a Method.
- Monitor static and instance member.
- Display Collections in a readable way. (Not just ToString)
- Choose one of three available UI solution presets.
- IMGUI support (default).
- TextMeshPro based uGUI support.
- UI Toolkit support.
- Detached UI Interface for custom UI solutions.
- Apply filter to displayed units.
- Custom control of how monitored members are displayed.
- Works both asynchronous and synchronous (WebGL supported).
- Mono & IL2CPP support.
- Clean open source code faithful to C# conventions & best practices.
- Global value processor. (like property drawer)
- Drag & drop example modules. (FPSMonitor, ConsoleMonitor etc.)
- Draw conditions. (only show value if true, not null etc.)
- Many more features.

Monitoring Member

Place the `[Monitor]` attribute an a field, property, event or method and get its value or state monitored in a customizable UI during runtime. When monitoring non static (instances), you have to register the monitored target, as shown in the next point. Other than that there is not much to it. Just try it out yourself.

```

Game Manager | GameManager
Game State: Playing
GameStateChanged: Action<GameState> Subscriber:11 Invocations: 2
[0] DeathCameraController.OnGameStateChanged (MonitoringExampleScene Death)
[1] ShootingTarget.OnGameStateChanged (MonitoringExampleScene EnemyTarget)
[2] ShootingTarget.OnGameStateChanged (MonitoringExampleScene EnemyTarget)
[3] ShootingTarget.OnGameStateChanged (MonitoringExampleScene EnemyTarget)
[4] ShootingTarget.OnGameStateChanged (MonitoringExampleScene EnemyTarget)
[5] ShootingTarget.OnGameStateChanged (MonitoringExampleScene EnemyTarget)
[6] ShootingTarget.OnGameStateChanged (MonitoringExampleScene EnemyTarget)
[7] ShootingTarget.OnGameStateChanged (MonitoringExampleScene EnemyTarget)
[8] PlayerState.OnGameStateChanged (MonitoringExampleScene Player)
[9] ShootingTarget.OnGameStateChanged (MonitoringExampleScene EnemyTarget)
[10] Action<GameState>.Invoke

Player Movement | Player
Position: X:[-0,15] Y:[10,62] Z:[-41,89]
Is Jumping: TRUE
Is Dashing: FALSE
Jumps Left: 0
Dash Energy: 1,71

```

```

[Monitor]
private Vector3 Position => transform.position;
[Monitor]
public bool IsJumping { get; private set; }
[Monitor]
public bool IsDashing { get; private set; }

[Monitor]
private int _jumpsLeft;
[Monitor]
private float _dashEnergy;

```

Note that the PlayerMovement class in this example is calling `this.RegisterMonitor()` in its Awake and `this.UnregisterMonitor` in its OnDestory method.

Instanced and Static Member

When monitoring non static member of a class, instances of these classes must be registered when they are created and unregistered when they are destroyed. This process can be automated or simplified, either by creating a custom Factory system that will create/instantiate objects and register them automatically, or by inheriting from a base type that will automatically register and unregister instances. You can use the following predefined base types for this purpose.

- `MonitoredBehaviour : MonoBehaviour`
- `MonitoredSingleton<T> : MonoBehaviour where T : MonoBehaviour`
- `MonitoredScriptableObject : ScriptableObject`
- `MonitoredObject : object, IDisposable`

```

// Monitored instance must be registered / unregistered.
public class Player : MonoBehaviour
{
    [Monitor] private int health;

    private void Awake()
    {
        MonitoringSystems.Resolve<IMonitoringManager>().RegisterTarget(target);
        // Or use this extension method:
        this.RegisterMonitor();
    }

    private void OnDestroy()
    {
        MonitoringSystems.Resolve<IMonitoringManager>().UnregisterTarget(target);
        // Or use this extension method:
        this.UnregisterMonitor();
    }
}

```

```

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

    // Just Remember to call base.Awake and base.OnDestroy if you override these methods.

```

```

protected override void Awake()
{
    base.Awake();
}

protected override void OnDestroy()
{
    base.OnDestroy();
}
}

```

```

public class GameManager
{
    // Static member are always monitored!
    [Monitor]
    public static GameState GameState { get; }
}

```

Monitoring Fields and Properties

Monitoring fields and properties is almost identical, differing only in their technical implementations. Just place the `Monitor`, `MonitorField` or `MonitorProperty` on either a field or a property and get its value displayed [automatically](#). Multiple types like Booleans, Collections, Vectors etc. are also displayed in a readable way. To customize how a monitored value is displayed you can use a [Value Processor](#) and utilize a variety of additional [formatting](#) attributes.

```

[MonitorField]
private int value;

[MonitorProperty]
private int Value { get; }

```

Monitoring Events

Use the `[Monitor]` or `[MonitorEvent]` attributes to monitor the state of an event. The `[MonitorEvent]` attribute accepts additional arguments to customize how the event is displayed.

Property	Description
<code>ShowSubscriberCount</code>	When enabled, the subscriber count of the event is displayed.
<code>ShowInvokeCounter</code>	When enabled, the invoke count of the event is displayed. (Can be incorrect when async profiling is enabled!)
<code>ShowSubscriberInfo</code>	When enabled, every subscribed delegate is displayed.
<code>ShowEventSignature</code>	When enabled, display the signature of the event.
<code>ShowEventHistory</code>	When enabled, the arguments of the last x amount of invocations is displayed. (Planned feature)

```

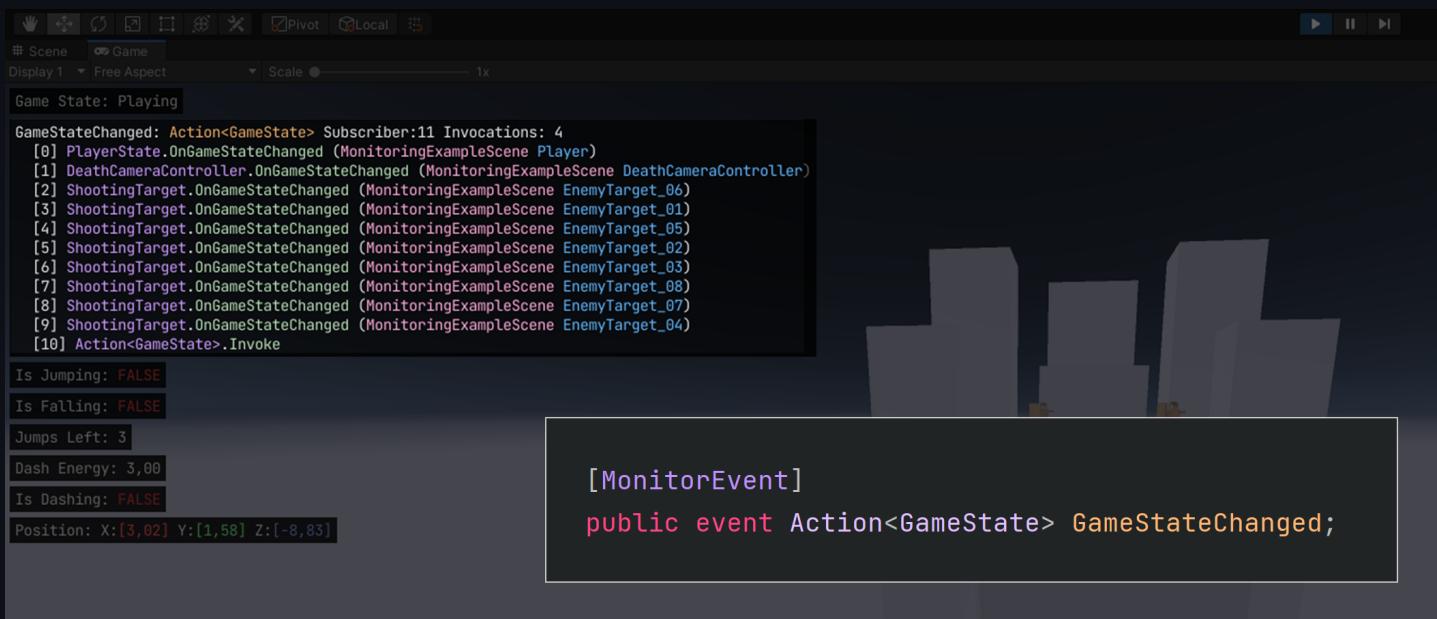
[Monitor]
public event Action OnGameStarted;

[Monitor]
public event Action<T> OnValueChanged;

[Monitor]
public event GameStateDelegate OnGameStateChanged;
public delegate void GameStateDelegate(GameState gameState);

```

```
// If you want to monitor how often an event is invoked without displaying every subscriber.
[MonitorEvent(ShowInvokeCounter = true, ShowSubscriberInfo = false)]
public event Action<Player> OnPlayerSpawn;
```



Monitoring Methods

A method can be monitored like a field or property with the additional feature that their out parameters are monitored too. Default parameter values can be set by passing them to the constructor of the `[MonitorMethod]` attribute. Even methods that return void can be monitored if they have at least one out parameter. Default and [Global Value Processor](#) are applied to monitored out parameters, meaning that collections, vectors etc. assigned via out parameter are displayed in a readable way and not just formatted with `ToString()`.

```
[Monitor]
public string GetName()
{
    return "Hello World"
}

[MonitorMethod(3, 5)]
public int Multiply(int lhs, int rhs)
{
    // Displayed value will be 15;
    return lhs * rhs;
}

[MonitorMethod]
public bool TryGetPlayer(int playerIndex, out Player player)
{
    // Method will be called and both the return value and the out parameter player are monitored.
    //...
}

[MonitorMethod]
public void Populate(out Vector3[] verts)
{
    // verts will be displayed in a readable way.
    //...
}
```

The screenshot shows the Unity Editor interface. In the top bar, there are icons for scene, game, pivot, and local. Below that, it says "Display 1 Free Aspect Scale 1x". The main area shows a 3D scene with platforms. On the left, the Inspector panel shows a "Get Movement Data" method with its parameters and return type. The code editor window below contains the following C# code:

```

Get Movement Data: void
out position: X:[0,64] Y:[0,58] Z:[-57,80]
out hits
[0]: Platform_7 (UnityEngine.MeshCollider)
[1]: Platform_9 (UnityEngine.MeshCollider)
[2]: NULL
[3]: NULL
out grounded: TRUE

GameState: Playing
GameStateCh
[0] Player
[1] Death
[2] Shoot
[3] Shoot
[4] Shoot
[5] Shoot
[6] Shoot
[7] Shoot
[8] Shoot
[9] Shoot
[10] Acti

```

[MonitorMethod]

```

private void GetMovementData(out Vector3 position, out Collider[] hits, out bool grounded)
{
    position = transform.position;
    hits = _raycastHits;
    grounded = !_isFalling && !_isJumping;
}

```

This example additionally shows that out parameters are formatted too.

Attributes

Use Attributes to customize the monitoring process & display of your member. The attributes provided are divided into three broad categories, first the "Monitoring Attributes" to determine which C# member to monitor, second the "Meta Attributes" to customize how a member is monitored and third other attributes used for various purposes.

Monitoring Attributes

Attribute	Base Type	Description
[Monitor]	Attribute	Monitor a field, property, event or method
[MonitorValue]	MonitorAttribute	Monitor a field or property
[MonitorProperty]	MonitorValueAttribute	Monitor a property
[MonitorField]	MonitorValueAttribute	Monitor a field
[MonitorEvent]	MonitorAttribute	Monitor an event
[MonitorMethod]	MonitorAttribute	Monitor a method

Meta Attributes

Attribute	Description
[MUpdateEvent]	Set an event that will trigger an refresh/update (more)
[MValueProcessor]	Set a method that will process the value before it is displayed as a string (more)
[MShowIf]	Set custom validation logic (more)
[MEnabled]	Set the default enabled state of the monitored member.
[MTag]	Set optional tags used for filtering

Meta Attributes (Formatting)

Attribute	Description
[MOptions]	Contains (almost) all of the options below.

Attribute	Description
[MFormat]	Custom format string used to display the members value if possible.
[MLabel]	Custom label for the member (otherwise humanized name).
[MFontSize]	Set the font size for the monitored member.
[MFontName]	Pass the name of a custom font style that will be used for the monitored member.
[MGroupName]	Set the group for the monitored member.
[MGroupElement]	Whether or not the monitored member should be wrapped in a group.
[MShowIndexer]	If the monitored member is a collection, determine if the index of individual elements should be displayed or not.
[MElementIndent]	The indent of individual elements of a displayed collection.
[MPosition]	The preferred position of an individual monitored member on the canvas.
[MTTextAlign]	Horizontal text align.
[MOrder]	Relative vertical order of the monitored member.
[MGroupOrder]	Relative vertical order of the group of the monitored member.
[MRichText]	Override local RichText settings.
[MTextColor]	Set the text color for the element.
[MBackgroundColor]	Set the background color for the monitored member.
[MGroupColor]	Set the background color for the group of the monitored member.
[MStyle]	UIToolkit only. Provide optional style names.

Other Attributes

Attribute	Description
[GlobalValueProcessor]	Declare a method as a global value processor for a specific type (more)
[DisableMonitoring]	Disable monitoring for the target class or assembly

Value Processor

You can add the MValueProcessorAttribute to a monitored field, porperty or method to gain more control of its string representation. Use the attribute 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.

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

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

```
[MValueProcessor(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;
}

```

This example demonstrates how to create a simple progress bar with the help of a value processor. Note that this code is already relatively optimized.

```

Is Jumping: FALSE
Is Falling: FALSE
Jumps Left: FALSE
Dash Enabled: FALSE
Is Dashed: FALSE
Position: (0.0, 0.0, 0.0)
[SerializeField] private int ammunition = 15;

[MonitorField]
[MFontSize(16)]
[MGroupElement(false)]
[MPosition(UIPosition.MiddleLeft)]
[MFontName("JetBrainsMono-Regular")]
[MValueProcessor(nameof(CurrentAmmunitionProcessor))]
private int _currentAmmunition;

```

Ammo: 15/20 

Ammo: 15/20 

```

private string CurrentAmmunitionProcessor(int current)
{
    var sb = new StringBuilder();
    sb.Append("Ammo: ");
    sb.Append(current.ToString("00"));
    sb.Append('/');
    sb.Append(ammunition.ToString("00"));
    sb.Append(' ');

    sb.Append('|', current);
    var gray = new Color(r:.25f, g:.25f, b:.3f);
    sb.Append(gray.ToRichText());
    sb.Append('|', ammunition - current);
    sb.Append("</color>");

    return sb.ToString();
}

```

Static Value Processor

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

```

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

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

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

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

```

```

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

```

```

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

// IEnumerable<T> ValueProcessor
[MValueProcessor(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")}";
}

```

Global Value Processor

You can declare a static method as a global value processor that is then used to process the value for every monitored member of the given type (instanced value processors will still be preferred). The value processor method must have a valid signature, meaning that it has to accept the monitored type as a first or second argument, can optionally accept an IFormatData object as a first argument and must return a string.

```

// Custom type
public struct Version
{
    public readonly short Major;
    public readonly short Minor;
}

[GlobalValueProcessor]
private static string GlobalValueProcessorVersion(Version version)
{
    return $"Version: {version.Major}.{version.Minor}";
}

[GlobalValueProcessor]
private static string GlobalValueProcessorVersion(IFormatData ctx, Version version)
{
    return $"{ctx.Label}: {version.Major}.{version.Minor}";
}

```

Conditional Display

You can use the MShowIfAttribute to define conditions that control if a monitored value is displayed or not. Note that the value will still be monitored but not drawn, meaning that fields, properties, events & methods will still be accessed. There are three different ways to validate if the targeted member is displayed or not.

Validated by Condition

Set a condition for the monitored value that when met will display the monitored value. This is useful to display important debugging information only when it is necessary.

```

// Queue will only be displayed if there are errors. (Queue is not empty)
[Monitor]

```

```

[MShowIf(Condition.CollectionNotEmpty)]
private Queue<string> errorCache;

// Property will only be displayed if Network is not available. (false)
[Monitor]
[MShowIf(Condition.False)]
private bool NetworkAvailable { get; }

// Property will only be displayed if MainCamera is not available or set. (null)
[Monitor]
[MShowIf(Condition.Null)]
private Camera MainCamera { get; }

```

Validated by Comparison

Very similar to [Validated by Condition](#) but more dynamic. Pass in another object value and determine a comparison type that is then used on the current value of the monitored member and the value passed as an argument. If the comparison evaluates to be true, the member will be displayed.

```

// Will only be displayed if the errorCode is 404
[Monitor]
[MShowIf(Comparison.Equals, 404)]
private int errorCode;

// Will only be displayed more than one player is active.
[Monitor]
[MShowIf(Comparison.Greater, 1)]
private int ActivePlayerCount { get; }

```

Validated by Member

Very dynamic way of determining if a member is displayed or not. Pass in the name of a field, property, method or event.

- Passed fields, properties and methods must return a Boolean that determines if the member is displayed or not.
- Passed methods can also accept the current value of the monitored member and use its for a more dynamic evaluation.
- Passed events must be `Action<bool>` and can be used to toggle the display of the member

```

// Will only be displayed if the field 'monitor' is true.
[Monitor]
[MShowIf(monitor)]
private bool IsAlive {get; }

[SerializeField] private bool monitor = true;

// Will only be displayed if the property 'IsDebug' is true.
[Monitor]
[MShowIf(monitor)]
private bool IsAlive {get; }

public static bool IsDebug { get; }

// Will only be displayed if the method 'Validate' returns true.
[Monitor]
[MShowIf(Validate)]
private Entity ActiveTarget {get; }

private bool Validate(Entity target)
{
    return target != null && this.IsActive;
}

```

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.

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

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

```
[Monitor]
[MUpdateEvent(nameof(OnGameStateChanged))]
private bool isGamePaused;

public event Action OnGameStateChanged;

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

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

Monitoring UI

Use the `IMonitoringUI` API to access the active MonitoringUIController and set properties like visibility, filtering etc.

Member	Description
<code>void Show()</code>	Show the monitoring UI overlay.
<code>void Hide()</code>	Hide the monitoring UI overlay.
<code>void ToggleDisplay()</code>	Toggle the visibility of the active monitoring display.
<code>bool IsVisible()</code>	True if monitoring UI is visible.
<code>T GetActiveUIController<T>()</code>	Get the currently active MonitoringUIController casted to a concrete implementation.

Member	Description
<code>void CreateMonitoringUI()</code>	Create a MonitoringUIController instance if there is none already. Disable 'Auto Instantiate UI' in the Monitoring Settings and use this method for more control over the timing in which the MonitoringUIController is instantiated.
<code>void ApplyFilter(string filter)</code>	Filter displayed units by their name, tags etc. more
<code>void ResetFilter()</code>	Reset active filter. more
<code>event Action<bool> VisibleStateChanged()</code>	Event invoked when the monitoring UI became visible or invisible.

UI Formatting

Formatting attributes can be used to apply custom styling options on how a monitored member is displayed. There are multiple ways to [reduce boiler plate code](#).

Attribute	Description
<code>[MOptions]</code>	Contains (almost) all of the options below.
<code>[MFormat]</code>	Custom format string used to display the members value if possible.
<code>[MLabel]</code>	Custom label for the member (otherwise humanized name).
<code>[MFontSize]</code>	Set the font size for the monitored member.
<code>[MFontName]</code>	Pass the name of a custom font style that will be used for the monitored member.
<code>[MGroupName]</code>	Set the group for the monitored member.
<code>[MGroupElement]</code>	Whether or not the monitored member should be wrapped in a group.
<code>[MShowIndexer]</code>	If the monitored member is a collection, determine if the index of individual elements should be displayed or not.
<code>[MElementIndent]</code>	The indent of individual elements of a displayed collection.
<code>[MPosition]</code>	The preferred position of an individual monitored member on the canvas.
<code>[MTextAlign]</code>	Horizontal text align.
<code>[MOrder]</code>	Relative vertical order of the monitored member.
<code>[MGroupOrder]</code>	Relative vertical order of the group of the monitored member.
<code>[MRichText]</code>	Override local RichText settings.
<code>[MTextColor]</code>	Set the text color for the element.
<code>[MBackgroundColor]</code>	Set the background color for the monitored member.
<code>[MGroupColor]</code>	Set the background color for the group of the monitored member.
<code>[MStyle]</code>	UIToolkit only. Provide optional style names.
<code>[MAttributeCollection]</code>	Can be used to create custom preset attributes to reduce boiler plate code.

The `[MOptions]` attribute contains almost all of the other options. Individual attributes will override settings passed with the `[MOptions]` attribute but depending on your preferences you can either use multiple attributes or just the `[MOptions]` attribute.

```
// Will be displayed as "Value: 3.141"
[Monitor]
[MOptions(Format = "0.000", FontSize = 16)]
```

```

private float pi = 3.14159265359;

// Effectively the same as
[Monitor]
[MFormat("0.000")]
[MFontSize(16)]
private float pi = 3.14159265359;

```

Reducing Boiler Plate Code

- Creating a custom attribute and inheriting from `MOptionsAttribute` will let you use the constructor of the custom attribute to set multiple values. The attribute can then be used on multiple monitored members to apply its settings.
- Any formatting or meta attribute applied to a class is also applied to every monitored member declared within it. Attributes directly applied to a member within such a class will always override the class scoped values meaning that you can still apply individual options to members within such a class. The `[MTag]` attribute is the only attribute that will not be overridden but added. This means that tags applied to a class scope will apply to every monitored member of that class even if these members have a custom tag attribute themselves.
- Any formatting or meta attribute applied to a custom attribute that inherits from `[MAttributeCollection]` will be added to monitored member with that custom attribute. This is a similar way like the first option of this list but does not require overriding a constructor. It acts more like a bridge or proxy.

The code segments below all do the same but with different approaches.

```

class MyClass
{
    [Monitor]
    [MFormat("0.000")]
    [MFontSize(16)]
    [MPosition(UIPosition.UpperRight)]
    [MGroupElement(false)]
    [MTag("Gameplay")]
    public int value1;

    [Monitor]
    [MFormat("0.000")]
    [MFontSize(16)]
    [MPosition(UIPosition.UpperRight)]
    [MGroupElement(false)]
    [MTag("Gameplay")]
    public int value2;

    [Monitor]
    [MFormat("0.000")]
    [MFontSize(16)]
    [MPosition(UIPosition.UpperRight)]
    [MGroupElement(false)]
    [MTag("Gameplay")]
    public int value3;
}

```

```

public class MyFormatting : MOptionsAttribute
{
    public MyFormatting()
    {
        Format = "0.00";
        FontSize = 16;
        Position = UIPosition.UpperRight;
        GroupElement = false;
        Tags = new string[] {"Gameplay"};
    }
}

class MyClass
{
    [Monitor]

```

```
[MyFromatting]
public int value1;

[Monitor]
[MyFromatting]
public int value2;

[Monitor]
[MyFromatting]
public int value3;
}
```

```
[MFormat("0.000")]
[MFontSize(16)]
[MPosition(UIPosition.UpperRight)]
[MGroupElement(false)]
[MTag("Gameplay")]
class MyFromatting : MAttributeCollection
{

class MyClass
{
    [Monitor]
    [MyFromatting]
    public int value1;

    [Monitor]
    [MyFromatting]
    public int value2;

    [Monitor]
    [MyFromatting]
    public int value3;
}
```

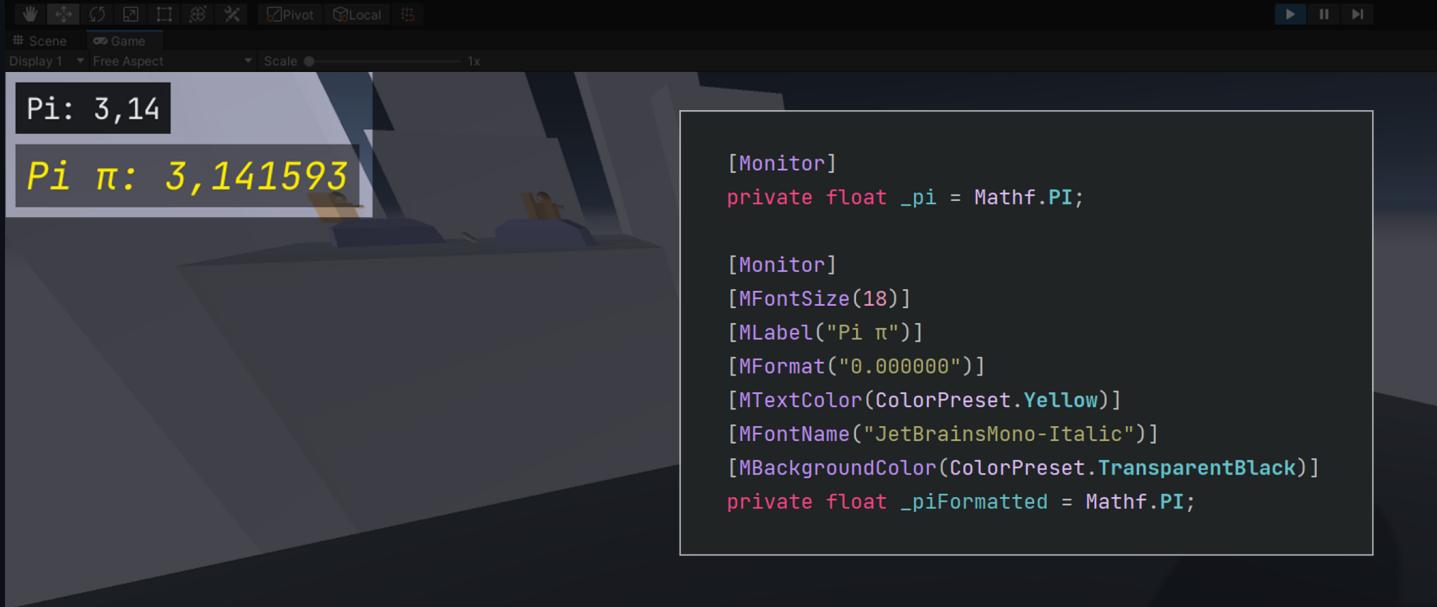
```
[MFormat("0.000")]
[MFontSize(16)]
[MPosition(UIPosition.UpperRight)]
[MGroupElement(false)]
[MTag("Gameplay")]
class MyClass
{
    [Monitor]
    public int value1;

    [Monitor]
    public int value2;

    [Monitor]
    public int value3;
}
```

When applying multiple formatting attributes either directly on the monitored member or via proxy, it may happen that the same attribute is applied with different values. In this case there is a clear structure which attribute will be used.

1. Directly applied to the monitored member.
2. Applied to the monitored member by `[MOptions]`
3. Applied to the monitored member by a custom `[MAttributeCollection]`
4. Directly applied to the monitored members class.
5. Applied to the monitored members class by `[MOptions]`
6. Applied to the monitored members class by a custom `[MAttributeCollection]`



Note that this is for display purposes only. There is no need to apply that many formatting options to every monitored member. Formatting is 100% optional.

UI Filtering

You can filter the currently displayed elements using the monitoring filter API from `IMonitoringUI`. Filtering will check for a variety of matches. If you want more explicit filtering you can disable most of these checks by navigating to (menu: Tools > Monitoring > Settings > Filtering). You can also use the settings to determine if filtering should be case sensitive or case insensitive. By default filtering is case insensitive!

Optional Filter	Description
Filter Label	Enable filtering using the displayed label. (case insensitive) This option is the most intuitive.
Filter Static or Instance	Filter for static or non static member with <i>static</i> or <i>instance</i>
Filter Type	Enable filtering using the name of the type of the monitored member. (e.g. <i>bool</i> , <i>int</i> , <i>queue</i> etc.)
Filter Declaring Type	Enable filtering using the name of the members declaring type. (e.g. <i>MonoBehaviour</i> , <i>Player</i> , <i>GameController</i> etc.)
Filter Member Type	Enable filtering using the member type (<i>Field</i> , <i>Property</i> , <i>Event</i> , <i>Method</i>)
Filter Tags	Enable filtering using tags applied by the <code>[MTag]</code> attribute. more

Filter API

```

// Apply filter. (This filter will only show fields and properties)
MonitoringSystems.Resolve<IMonitorUI>().ApplyFilter("Field & Property");

// Reset filter.
MonitoringSystems.Resolve<IMonitorUI>().ResetFilter();

```

Absolute Filtering

Filter starting with a `@` are always case sensitive and only use the actual name of the monitored member.

Tag only Filtering

Combining Filters

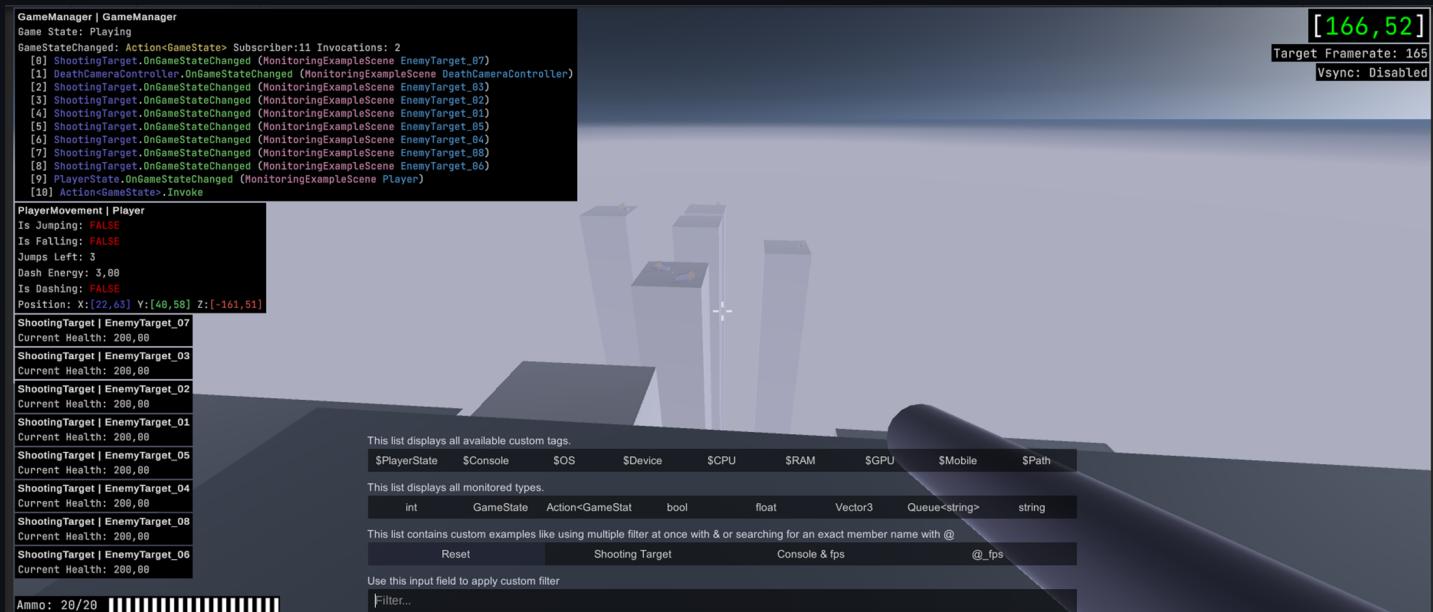
Use a & symbol to combine multiple filters.

Negation Filter

Append an ! to the beginning of a filter to negate it.

You can change all of the symbols mentioned above in the monitoring settings by navigating to (menu: Tools > RuntimeMonitoring > Settings > Filtering).

This example shows a custom filtering setup in the example scene. I would recommend to test it yourself.



Custom UI Controller

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

- Create a new class and inherit from `MonitoringDisplayController`.
- Implement the abstract methods 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.

The following abstract member have to be implemented when inheriting from `MoniotoringUIController`.

Member	Description
<code>bool IsVisible()</code>	Return true if the UI is active and false if it is not active.
<code>void ShowMonitoringUI()</code>	Activate / show the UI.
<code>void HideMonitoringUI()</code>	Deactivate / hide the UI.
<code>void OnUnitCreated(IMonitorUnit unit)</code>	Use to add UI elements for the passed unit.
<code>void OnUnitDisposed(IMonitorUnit unit)</code>	Use to remove UI elements for the passed unit.

Systems and API

Get the interface for a monitoring system by calling `MonitoringSystems.Resolve<TInterface>()`. Interface implementations will not change during runtime meaning it is safe to cache them when necessary.

System Interface	Description
<code>IMonitoringManager</code>	Core access point for the system. Contains initialization and unit registry. more
<code>IMonitoringUI</code>	Access the active monitoring UI. more
<code>IMonitoringSettings</code>	Access current configuration of the plugin.
<code>IMonitoringPlugin</code>	Access static information about the plugin.
<code>IMonitoringUtility</code>	Interface providing various uncategorized utility access points. more

```
//Example how to get / resolve a monitoring system interface.  
IMonitoringUI monitoringUI = MonitoringSystems.Resolve<IMonitoringUI>();
```

Type Interfaces

Important internally used types implement interfaces that should make it more easy to work with and extend this tool. These interfaces are especially used when creating a custom UI controller.

Type Interface	Description
<code>IMonitorUnit</code>	Internally used handle for instances of a monitored member. more
<code>IMonitorProfile</code>	Internally used profile describing a monitored member. more

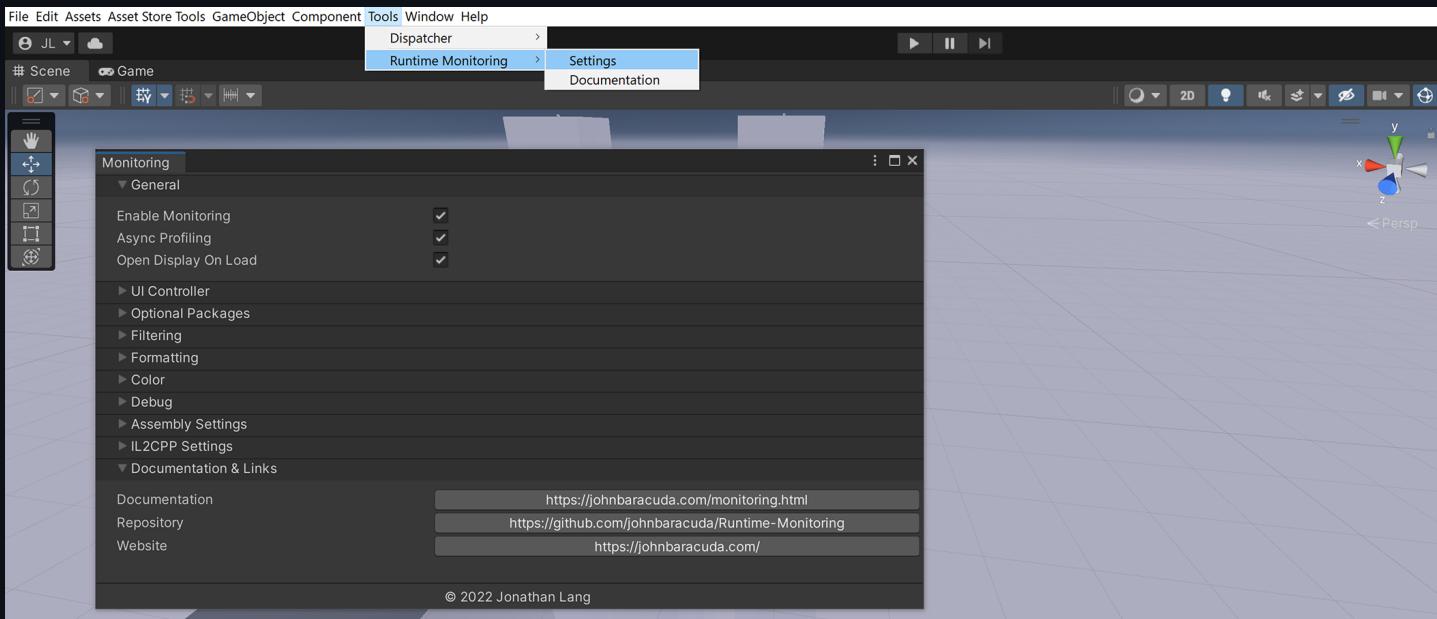
Monitoring Manager

The `IMonitoringManager` interface provides core access points.

Member	Description
<code>bool IsInitialized { get; }</code>	Value indicated whether or not monitoring profiling has completed and monitoring is fully initialized.
<code>event ProfilingCompletedListener ProfilingCompleted</code>	Event is invoked when profiling process for the current system has been completed. Subscribing to this event will instantly invoke a callback if profiling has already completed.
<code>event Action<IMonitorUnit> UnitCreated</code>	Event is called when a new MonitorUnit was created.
<code>event Action<IMonitorUnit> UnitDisposed</code>	Event is called when a MonitorUnit was disposed.
<code>void RegisterTarget<T>(T target) where T : class</code>	Register an object that is monitored during runtime.
<code>void UnregisterTarget<T>(T target) where T : class</code>	Unregister an object that is monitored during runtime.
<code>ReadOnlyList<IMonitorUnit> GetStaticUnits();</code>	Get a list of MonitorUnits for static targets.
<code>ReadOnlyList<IMonitorUnit> GetInstanceUnits();</code>	Get a list of MonitorUnits for instance targets.
<code>ReadOnlyList<IMonitorUnit> GetAllMonitoringUnits();</code>	Get a list of all MonitorUnits.

Monitoring Settings

Use the `IMonitoringSettings` API can be used to access the current configuration set in the monitoring settings window. This API is readonly and mostly used by internal processes. So far it is available for transparency reasons only.



Monitoring UI API

Use the `IMonitoringUI` API to access the active `MonitoringUIController` and set properties like visibility, filtering etc. [more](#)

Member	Description
<code>void Show()</code>	Show the monitoring UI overlay.
<code>void Hide()</code>	Hide the monitoring UI overlay.
<code>void ToggleDisplay()</code>	Toggle the visibility of the active monitoring display.
<code>bool IsVisible()</code>	True if monitoring UI is visible.
<code>T GetActiveUIController<T>()</code>	Get the currently active <code>MonitoringUIController</code> casted to a concrete implementation.
<code>void CreateMonitoringUI()</code>	Create a <code>MonitoringUIController</code> instance if there is none already. Disable 'Auto Instantiate UI' in the Monitoring Settings and use this method for more control over the timing in which the <code>MonitoringUIController</code> is instantiated.
<code>void ApplyFilter(string filter)</code>	Filter displayed units by their name, tags etc. more
<code>void ResetFilter()</code>	Reset active filter. more
<code>event Action<bool> VisibleStateChanged()</code>	Event invoked when the monitoring UI became visible or invisible.

Monitoring Utility

The `IMonitoringUtility` interface provides a variety of uncategorized utility functions.

Member	Description
--------	-------------

Member	Description
<code>bool IsFontHashUsed(int fontHash)</code>	Returns true if the passed hash from the name of a font asset is used by a MFontAttribute and therefore required by a monitoring unit. Used to dynamically load/unload required fonts.
<code>IMonitorUnit[] GetMonitorUnitsForTarget(object target)</code>	Get a list of IMonitorUnits registered to the passed target object.
<code>IReadOnlyCollection<string> GetAllTags()</code>	Get a list of all custom tags, applied by [MTag] attributes that can be used for filtering.

Monitor Unit

The `IMonitorUnit` interface offers access points to the internally used handle for instances of a monitored member.

Interface Member	Description
<code>string Name { get; }</code>	Readable display name.
<code>object Target { get; }</code>	The target object of the monitored member. Null if static.
<code>IMonitorProfile Profile { get; }</code>	Profile describing the monitored member. more
<code>bool Enabled { get; set; }</code>	The active state of the unit. Only enabled units are updated and displayed.
<code>int UniqueID { get; }</code>	Unique Id
<code>event Action<bool> ActiveStateChanged</code>	Event is invoked when the units active state has changed.
<code>event Action<string> ValueUpdated</code>	Event is invoked when the value of the monitored member has changed.
<code>event Action Disposing</code>	Event is invoked when the unit is being disposed.
<code>void Refresh()</code>	Force the monitored member to update its state.
<code>string GetState()</code>	Get the current value or state of the monitored member as a formatted string.

Monitor Profile

The `IMonitorProfile` interface offers access points to the internally used profile describing a monitored member. It contains cached reflection and formatting data.

Interface Member	Description
<code>MonitorAttribute Attribute { get; }</code>	Instance of the attribute that is used to flag the target member to be monitored.
<code>MemberInfo MemberInfo { get; }</code>	Member info of the monitored member.
<code>UnitType UnitType { get; }</code>	The type of the member (either field, property, event or method)
<code>bool ReceiveTick { get; }</code>	True if the unit receives a custom tick event.
<code>Type UnitTargetType { get; }</code>	The declaring type of the monitored member.
<code>Type UnitValueType { get; }</code>	The type of the monitored member.
<code>bool IsStatic { get; }</code>	Indicates if monitored member is static or non static.
<code>IFormatData FormatData { get; }</code>	Contains information about label, font and more formatting data.

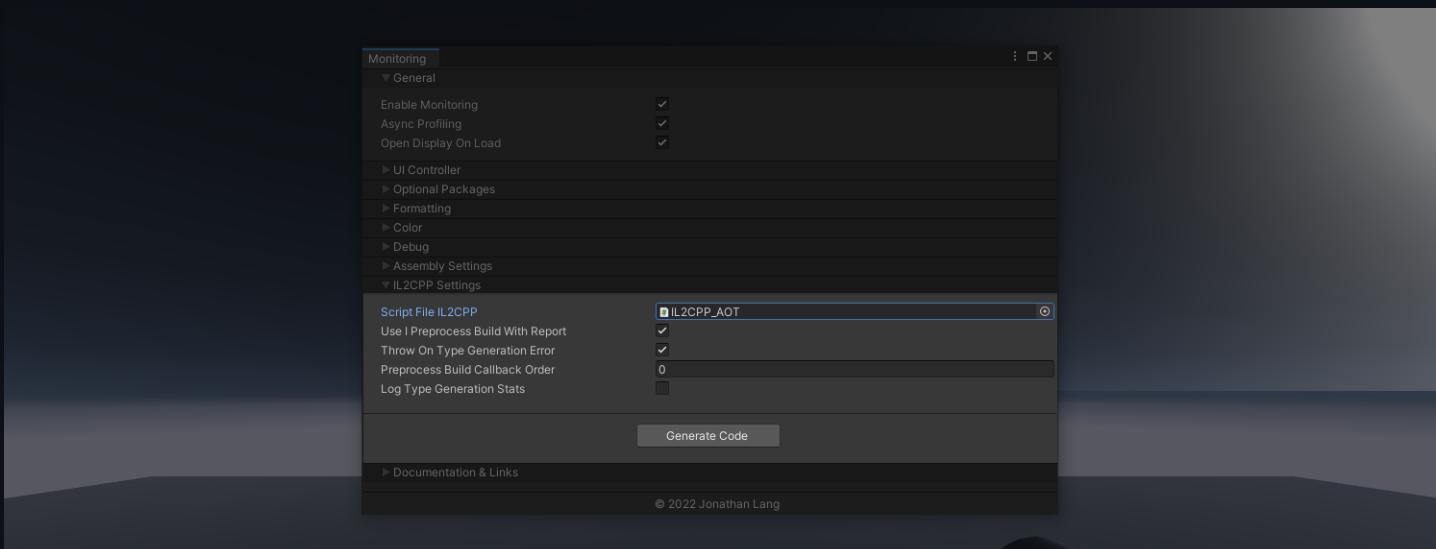
Interface Member	Description
<code>string[] Tags { get; }</code>	Additional meta data for filtering.
<code>T GetMetaAttribute<T>()</code>	The monitoring profiler caches every additional attribute that inherits from MonitoringMetaAttribute on the profile. You can access these custom attributes during runtime using this method without using reflection.
<code>bool TryGetMetaAttribute<T>(out T attribute)</code>	Try to get a MonitoringMetaAttribute.

Compatibility

If you encounter any compatibility issues please create an Issue on GitHub. Runtime Monitoring aims to provide compatibility for every platform, scripting backend & Unity version.

Runtime Compatibility

Both Mono and IL2CPP runtimes are supported. RTM is making extensive use of dynamic type and method creation during its profiling process. In order to create these types, IL2CPP requires AOT compilation (Ahead of time compilation) When using IL2CPP runtime a list of types is generated shortly before a build to give the compiler the necessary information to generate everything it needs during runtime. You can manually create this list form the settings window.



Platform Compatibility

I can't test all of these platforms for compatibility. Let me know if you have tested any platform that is NA or not listed here.

Platform	Compatible	Note
Windows Standalone	✓	
Linux Standalone	NA	
Mac Standalone	NA	
iOS	NA	
Android	NA	
WebGL	✓	Async profiling disabled

Platform	Compatible	Note
Stadia	NA	
XBox One	NA	
XSX	NA	
PlayStation 4	NA	
PlayStation 5	NA	

Optimizations

In general RTM tries to be as optimized as possible by reducing allocations where ever possible and by doing a lot of the heavy work during the initial profiling. However due to the nature of some types and the creation of strings, allocations cannot be prevented. If performance is a sensitive aspect of your project, here are some tips and tricks to keep in mind.

Reference Types & Value Types

Since there is no easy way to check whether the actual value of a reference type has changed when it is evaluated, a monitored Reference Type is processed with `ToString()` each time it is evaluated. This, by default happens either every Update or Tick cycle and may generate a lot of garbage in a very short time, without much scope for automatic optimization. For this reason, it is recommended to pass an `UpdateEvent` for the monitored value whenever possible to reduce memory allocations. Of course, this shouldn't matter much if you're only debugging one value, but could be detrimental if you want to keep monitoring your member in a release or shipped build. The same is not true for monitored Value Types, as these are compared to a cached value to ensure that an update event and a string creation is only triggered when the value has actually changed.

Collections

Because collections are Reference Types the same applies here but on an even greater scale. Pass an update event when ever possible if you intend to monitor a collection over a longer period. Now because the example below requires a lot of boiler plate code I would not recommend this if you just want to quickly debug the values of a collection. I also want to mention that a better solution is planned and WIP.

```
[Monitor]
[MUpdateEvent(nameof(OnNamesChanged))]
public List<string> Names = new List<string>() {"Riebeckite", "Prisoner", "Feldspar"};

private event Action OnNamesChanged;

public void AddName(string name)
{
    Names.Add(name);
    OnNamesChanged();
}

public void RemoveName(string name)
{
    Names.Remove(name);
    OnNamesChanged();
}
```

Transform

Monitored Transforms are another type that have the potential to create a lot of garbage. A simple thing you could do to reduce overhead is to control the `Transform.hasChanged` flag on the Transform itself. The monitoring unit/handler will check the flag and only raise an update event if the flag is set to true. (which it is unless changed manually) Unity is not controlling `Transform.hasChanged`.

Frequently Asked Questions

How can I disable the tool in a release?

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.

How can I uninstall the tool?

You can just remove the plugin by deleting the folder Assets/Baracuda.

Are there any planned features?

- Add to OpenUPM
- Monitoring Editor Window for edit time monitoring.
- Monitored collection type with "dirty" flag. (Optimization)
- Option to create "virtual" units during runtime.
- Tutorial how to use.
- Guide how to create a custom UI Controller.
- Guide how to customize UI.

Assemblies and Modules?

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.UIToolkit	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	✓	

What is Thread Dispatcher?

Thread Dispatcher is another free & open source tool I developed to pass the execution of a Delegate, Coroutine or Task from a background thread to the main thread, and await its completion or result on the calling thread as needed. Runtime Monitoring optionally uses a background thread for initial assembly profiling & setup processes. You can find more information about it on its [GitHub Repository](#), its [Documentation](#) and its page on the [Asset Store](#). I would also appreciate if you would leave a ❤️ or a ★ on its page because it is also something I spent a lot of time working on.

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