

# What is More Effective Coroutines Pro?

More Effective Coroutines Pro runs on the exact same super fast and super optimized core that the free version runs on, and also adds a host of additional methods, most of which can't be found in Unity's coroutines or anywhere else. [MEC Pro can be found on the Unity asset store here](#).

## Editor Update

Unity's coroutines don't run in the editor when the app is not playing, but MEC coroutines can.

Editor coroutines are generally used to give the developer a preview of an effect before the game runs, or to perform update or networking sync tasks that need to be done before the actual app is compiled. Editor coroutines will all be destroyed when the user hits the play button, and will not run while in play mode. `Time.time` and `Time.deltaTime` will not work while in editor mode, instead you should use `Timing.LocalTime` and `Timing.DeltaTime`.

Be conservative when using editor coroutines: An infinite loop or a very processor intensive function here can destroy your entire project. Always make sure to have a backup of your project available whenever you mess with these. Also keep in mind that any changes you make to an object in your scene in the editor will change the initial state of that object when you run or compile the app. For example, you could make an editor coroutine that made your character wink at the developer before play mode, but this would have the side effect of making the character randomly start mid-wink occasionally when they started or compiled the app.

In order to run in the editor, set the Segment to `EditorUpdate` or `EditorSlowUpdate`. Make sure your class has the `[ExecuteInEditMode]` tag attached, and then use them normally. Here is an example:

```
using UnityEngine;
using System.Collections.Generic;
using MovementEffects;

[ExecuteInEditMode]
public class EditorTesting : MonoBehaviour
{
    void OnEnable()
    {
        Timing.RunCoroutine( _RunOverAndOver() ,
Segment.EditorUpdate);
    }

    IEnumerator<float> _RunOverAndOver()
```

```

    {
        while(true)
        {
            if(!enabled)
                yield break;

            Timing.RunCoroutine(_MoveThisObject(),
Segment.EditorUpdate);

            yield return Timing.WaitForSeconds(1f);
        }
    }

IEnumerator<float> _MoveThisObject()
{
    double startTime = Timing.LocalTime;
    Vector3 direction = Random.onUnitSphere;

    while(Timing.LocalTime - startTime < 7d)
    {
        Vector3 tmp = transform.position;
        tmp += direction * Timing.DeltaTime;
        transform.position = tmp;

        yield return 0f;
    }
}
}

```

## Realtime Update

Another timing segment you can use in MEC Pro is RealtimeUpdate. This segment is just like update, but Timing.LocalTime and Timing.DeltaTime ignore Unity's timescale. This can be useful while controlling menus that happen while your game is paused. (If you pause by turning the timescale to 0.)

## Layers and Tags

The free version of MEC only has tags, but MEC Pro adds layers. Collectively tags and layers are called graffiti because they are both just ways to identify a particular instance of a coroutine. The only real difference between a layer and a tag is that a layer is an integer and a tag is a string, but in MEC Pro you have the option of supplying one, the other, or both. Once you supply

graffiti for an instance, you can pause/resume, kill, or use that graffiti for `RunCoroutineSingleton`.

In Unity every `GameObject` is assigned a unique number, which can be accessed by calling `gameObject.GetInstanceID()`. The instance id can change every time you run the application, but it is guaranteed that no other `gameObject` instance will be assigned the same id during a single run. So even if you have a swarm of enemies that all have the same scripts attached to them you can still run them on a layer which is the instance id of the `gameObject` they were created on and then kill all coroutines attached to one particular enemy using

```
Timing.KillCoroutines(enemy.gameObject.GetInstanceID());
```

If you had, say, an amnesia gun you could graffiti all your enemy's AI logic with both the instance id and a tag:

```
Timing.RunCoroutine(_EnemyAI(), gameObject.GetInstanceID(), "AI");  
// Somewhere else in code:  
void HitEnemyWithAmnesiaGun(EnemyController enemy)  
{  
    Timing.KillCoroutines(enemy.gameObject.GetInstanceID(), "AI");  
}
```

`KillCoroutines` only kills the instance that match all the graffiti that you pass in, so if you coded it right the above code would make the enemy stand there and do nothing, but would also leave any other coroutines that might be running on that particular character untouched.

Unity's coroutines will always cancel all of the coroutines that are associated with a `GameObject` whenever you disable that object. Sometimes you would rather just pause a coroutine while it's disabled and then resume it as soon as the object is re-enabled. That sort of pattern is really easy to set up in MEC like this:

```
void Start ()  
{  
    Timing.RunCoroutine(_MoveUpAndDown(), gameObject.GetInstanceID());  
}  
void OnEnable()  
{  
    Timing.ResumeCoroutines(gameObject.GetInstanceID());  
}  
void OnDisable()  
{  
    Timing.PauseCoroutines(gameObject.GetInstanceID());  
}
```

# RunCoroutineSingleton

You often want to run a coroutine, but you don't want to run multiple copies of that coroutine.

One common use case is to have a button that visibly pops onto the screen and back out as it is enabled and disabled. You might write a coroutine that expands the transform of that button every frame in a very pleasing lerp effect, but find that if the user clicks back and forth very quickly then more than one coroutine can be started at the same time in a very unpleasing way.

RunCoroutineSingleton allows you to fix that. The normal way to use it is to apply a unique tag to every coroutine in the set (for this example that would be every coroutine that moves that button). You can then call `Timing.RunCoroutineSingleton(_ButtonPopIn(button.transform, "HealthButtonMovementSet"))`;

Tags do result in a memory alloc, so if you are trying to be as efficient as possible with your GC allocs then you might want to use a handle to define your singleton.

```
class Foo
{
    CoroutineHandle handle;

    void Start()
    {
        handle = Timing.RunCoroutineSingleton(_MoveThatThing(),
        handle, SingletonBehavior.Overwrite);
    }

    ...
}
```

The third parameter that you pass in is an enum that determines how any conflicts that are found will be handled. Abort will fail to run the current coroutine if one is already defined that matches your parameters. Overwrite will kill any matching coroutines and then run yours. Wait will automatically keep your coroutine paused until all the matches have finished and then it will execute.

One thing that you can do with the wait behavior is to set up a sequence of coroutines to run one after the other by giving them all the same tag and setting them to wait. Each coroutine will only wait for the coroutines that exist with that tag at the moment it's created, later additions with the same tag will be ignored.

Below are some examples of usage:

```
Timing.RunCoroutineSingleton(_ButtonPopIn(button.transform),
    "HealthButtonMovementSet", SingletonBehavior.Overwrite);
Timing.RunCoroutineSingleton(_ButtonPopIn(button.transform),
    "HealthButtonMovementSet", SingletonBehavior.Wait);

CoroutineHandle handle;
handle = Timing.RunCoroutineSingleton
    (_ButtonPopIn(button.transform), handle);
handle = Timing.RunCoroutineSingleton
    (_ButtonPopIn(button.transform), Segment.FixedUpdate, handle);
```

## Pause and Resume

In More Effective Coroutines Pro you can pause all coroutines that have a particular tag or layer and resume them later.

This could be useful, for instance, if you had a two layer menu system. If you tagged every movement coroutine with the string “layer1” and opened a menu on layer 2 you could easily stop all movement on layer1 with the command “Timing.PauseCoroutines(“layer1”);”. Later, when your layer 2 menu closed you could resume all movement on layer 1 right where it left off by calling “Timing.ResumeCoroutines(“layer1”);”

Remember that with tags the string has to match exactly every time, so “layer1” is not the same as “Layer1”, “layer 1”, or “laier1”. Always make sure you can spell all tags consistently and watch out for capitalization and spaces (both are ok, but you have to use them the same way every time).

## Controlling Coroutines by Handle

Whenever you call Timing.RunCoroutine(...) a CoroutineHandle object is returned. That object can be used to make one coroutine wait for another using “yield return Timing.WaitUntilDone(handle);”. However, in MEC Pro the CoroutineHandle can do more.

**Using Tag or Layer you can set or retrieve the associated graffiti.**

```
CoroutineHandle handle = Timing.RunCoroutine(_Foo());  
  
// ....  
  
string oldTag = handle.Tag;  
handle.Tag = "newTag";  
  
if(handle.Layer == null) // No layer is assigned  
    handle.Layer = gameObject.GetInstanceID();
```

Note: Layer is a *nullable int* (int?) so it may require casting into a regular int in your code, and you should check whether the value is null when you query either tags or layers (a null value means that there is no tag or layer assigned.)

**Segment can retrieve or change the timing segment that the coroutine is running in.**

```
handle.Segment = Segment.SlowUpdate;
```

### **IsRunning, IsPaused, and IsValid**

IsRunning returns true until the coroutine terminates, and then it returns false. Paused and/or locked coroutines are considered to be running.

IsPaused returns true if the coroutine is paused or locked.

IsValid returns true if the coroutine handle has ever pointed to a valid coroutine (regardless of whether that coroutine is currently running).

### **Linking coroutine handles**

Handles can be linked to one another using the LinkCoroutine function. This means that the “master” coroutine will send a kill command to the “slave” coroutine when it ends (either from a kill command or just by ending the function). It will also copy any pause or resume commands (but not any calls to WaitForSeconds).

The reason you would want to link coroutines is so you can start two or more coroutines independently but treat the whole group like it’s a single coroutine with the same scope as the master coroutine. A simple example could be if you have a loading progress bar with a coroutine that updates the progress bar’s position. Let’s say that you find that loading can sometimes get stuck and you want to create a second coroutine to watch the progress bar and make the words “sorry for the delay” march around on the screen when it gets stuck. Let’s say there’s also a

cancel button which kills your first coroutine and a pause button which pauses it. You could turn your single reference into a list, but that starts getting messy fast. The simple way is to just link the two coroutines in MEC and then you can treat them like a single one. The slave will always be terminated or paused along with the master.

### Getting the current coroutine handle

This is a useful convenience that allows you to retrieve your own handle from within a coroutine. You can use your own handle to set up links on the fly, change your own tags, coordinate static lists of different instances of the same coroutine, or all kinds of things.

```
private IEnumerator<float> _shout(float time, string text)
{
    yield return Timing.WaitForSeconds(time);

    CoroutineHandle myHandle = new CoroutineHandle();
    yield return Timing.GetMyHandle(x => myHandle = x);

    Debug.Log(myHandle.Tag + ": " + text);
}
```

NOTE: GetMyHandle is a special function that does not cause a one frame delay even though it's in a yield return statement.

## Extension Methods

Extension methods can be extremely powerful tools. They allow you to run the same coroutine, but change the way it runs while running it. That probably sounds confusing, so here's an example:

## AddDelay

AddDelay runs the coroutine after a delay.

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections.Generic;
using MovementEffects;

public class ButtonExample : MonoBehaviour
{
    public Button button1;
    public Button button2;
    public Button button3;

    void Start ()
    {
        Timing.RunCoroutine(_MoveBackAndForth(button1).AddDelay(1f));
        Timing.RunCoroutine(_MoveBackAndForth(button2).AddDelay(2f));
        Timing.RunCoroutine(_MoveBackAndForth(button3).AddDelay(3f));
    }

    private IEnumerator<float> _MoveBackAndForth(Button
myButton)
    {
        Vector3 myPos = myButton.transform.localPosition;
        float time = 0f;

        while(time < 2f)
        {
            myPos.x += Timing.DeltaTime;
            time += Timing.DeltaTime;
            transform.localPosition = myPos;
            yield return 0f;
        }

        while (time < 4f)
        {
            myPos.x -= Timing.DeltaTime;
            time += Timing.DeltaTime;
            transform.localPosition = myPos;
            yield return 0f;
        }
    }
}
```



The code above will simply move three buttons, first 2 units to the right over 2 seconds, and then 2 units to the left. The interesting part is that during the `Timing.RunCoroutine` calls we used the `AddDelay` function to add a different delay to each call, so `button1` will start moving after one second, `button2` will start moving after 2 seconds, and `button3` after 3 seconds. This is far cleaner than passing in the delay and putting it at the top of the function.

## CancelWith

This can take either a `GameObject` or a function that returns a `bool`. If you pass in a `GameObject` then the coroutine will automatically be terminated if the `GameObject` is destroyed or disabled. You might want to use this if your coroutine is moving some UI element and that UI element might occasionally end up being destroyed before the movement completes. `CancelWith` will ensure that the coroutine quits cleanly without throwing any exceptions. If you want to safely modify two or three game objects there are overloads to pass in two or three at a time. If you want to be safe from more than three game objects going out of scope then you can chain calls to `CancelWith`.

If you pass in a function then the coroutine will be terminated as soon as that function returns `false`.

```
int framesLeft = 100;
GameObject obj1;
GameObject obj2;
GameObject obj3;
GameObject obj4;

void Start()
{
    Timing.RunCoroutine(_Coroutine().CancelWith(obj1, obj2,
obj3).CancelWith(obj4));

    Timing.RunCoroutine(_Coroutine().CancelWith(CancelFunction));
}

bool CancelFunction()
{
    if(framesLeft <= 0)
        return false;

    framesLeft--;
    return true;
}
```

## Append and Prepend

These two functions can be used to chain two (or more) coroutines together. So this would turn right and then turn left:

```
Timing.RunCoroutine(_TurnRight().Append(_TurnLeft()));
```

They can also be used to append a delegate to the end of a coroutine. So this might be used to run a coroutine and then destroy the object once it was done:

```
Timing.RunCoroutine(_MoveToFinalPosition(obj1).Append(delegate  
{ Destroy(obj1); }));
```

Of course in many cases you could have also just added the line to `Destroy(obj1)` to the end of the movement coroutine, but if you are using the same move function in several places and you don't normally want to destroy the object at the end then this can be a clean way to add that functionality without fracturing your code base.

There are also many cases where you might want to call some event once a coroutine has finished. If that event is related more to how you are calling the coroutine than what the coroutine is doing then it is better to encapsulate the code for the event in the place that is calling it, and this pattern does that.

## Superimpose

This quirky function superimposes two coroutines into a single handle. The combined coroutine won't finish until both of its contributing coroutines are done. This can be combined with the `WaitUntilDone` function to make a coroutine wait until both functions finish before continuing. Here's an example of that:

```
void Start()  
{  
    var handle =  
Timing.RunCoroutine(_NetworkStream1().Superimpose(_NetworkStream2()));  
    Timing.RunCoroutine(_RunWhenDone(handle));  
}  
  
IEnumerator<float> _NetworkStream1()  
{  
    // Networking stuff happens here.  
}  
  
IEnumerator<float> _NetworkStream2()  
{  
    // Other networking stuff happens here.  
}
```

```
IEnumerator<float> _RunWhenDone(IEnumerator<float> handle)
{
    yield return Timing.WaitUntilDone(handle);
    // This part gets run as soon as both networking streams
    are done.
}
```

## Hijack

This fun little function alters the return value of a coroutine. This can be useful for cutscenes or replays where you want the same code to be executed but in slow motion.

```
float slowdown = 0.1f;

Timing.RunCoroutine(_MoveButton().Hijack(input =>
{
    if(input <= Timing.LocalTime)
        input = (float)Timing.LocalTime;
    return input + slowdown;
}));
```

All extension functions are very lightweight additions, so feel free to use them liberally or chain them together if you like.

For the latest documentation, FAQ, or to contact the author visit <http://trinary.tech/category/mec/> and <http://trinary.tech/category/mec/mec-pro/>