

What is More Effective Coroutines

More Effective Coroutines (MEC) is a free asset on the Unity asset store. If you don't already have it, you can get it [here](#).

More Effective Coroutines is an improved implementation of coroutines that runs about twice as fast as Unity's coroutines do and has zero per-frame memory allocations. It has been tested and refined extensively to maximize performance and create a rock solid platform for coroutines in your app.

Features	Unity's default coroutines	MEC Free	MEC Pro
Uses the yield return structure	✓	✓	✓
Time taken to execute 100,000 empty coroutines	~ 110.53	~ 9.62	~ 9.64
Can run singleton instances of coroutines	✗	✗	✓
Can switch the timing of coroutines mid process	✓	✗	✓
CallDelayed CallPeriodically and CallContinuously functions	✗	✓	✓
You can choose whether to link your coroutines to a gameobject or not	✗	✓	✓
Compatible with MEC Multithreaded	✗	✓	✓

Segments	Unity's default coroutines	MEC Free	MEC Pro
Update	✓	✓	✓
Fixed Update	✓	✓	✓
Late Update	✗	✓	✓
Slow Update	✗	✓	✓
Realtime Update	✓	✗	✓
Editor Update	✗	✗	✓
Editor Slow Update	✗	✗	✓
End Of Frame	✓	✗	✓
Manual Timeframe	✗	✗	✓

How to Change Over

More Effective Coroutines are called slightly differently than Unity's coroutines. The structure is exactly the same, so in most cases it's just a matter of find and replace inside your code.

First, you need to include the two namespaces that MEC uses. MEC coroutines are defined in the MovementEffects namespace and they also rely on the System.Collections.Generic functionality rather than the System.Collections functionality that unity coroutines use. System.Collections is hardly ever used for anything except Unity's coroutines, so an easy way to switch is to do a "Find and Replace," and then just change the lines that have errors. So make sure these two using statements are at the top of every C# script that is using MEC coroutines:

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

Next, replace every instance of StartCoroutine, so this

```
StartCoroutine ( _CheckForWin () ) ;
```

..is replaced with RunCoroutine. (You have to pick the execution loop when you define the process, and it defaults to "Segment.Update".)

```

// To run in the Update segment:
Timing.RunCoroutine(_CheckForWin());
// To run in the FixedUpdate segment:
Timing.RunCoroutine(_CheckForWin(), Segment.FixedUpdate);
// To run in the LateUpdate segment:
Timing.RunCoroutine(_CheckForWin(), Segment.LateUpdate);
// To run in the SlowUpdate segment:
Timing.RunCoroutine(_CheckForWin(), Segment.SlowUpdate);

```

The process' header will then need to be changed as well. It turns from this:

```

IEnumerator _CheckForWin()

{

    ...

}

```

To this:

```

IEnumerator<float> _CheckForWin()

{

    ...

}

```

It is a very good idea to get in the habit of always putting an underscore before all coroutine functions. The reason for this is that coroutines (both Unity and MEC ones) have an annoying tenancy to pretend to run correctly but to actually not execute at all if you try to run the function without using RunCoroutine. The “_” before the function helps you to remember to always use “Timing.RunCoroutine(_CheckForWin());” rather than trying to call it using the line “_CheckForWin();”.

Whenever you want to wait for the next frame you can use “yield return Timing.WaitForOneFrame;” or just “yield return 0;”. Both of those statements evaluate to the same thing, so you can pick the one you like better, but I suggest using WaitForOneFrame since it will make your code more easily understandable to anyone who is not familiar with coroutines and/or Unity. So this,

```
IEnumerator _CheckForWin()
{
    while (_cubesHit < TotalCubes)
    {
        WinText.text = "Have not won yet.";

        yield return null;
    }

    WinText.text = "You win!";
}
```

Would turn into this:

```
IEnumerator<float> _CheckForWin()
{
    while (_cubesHit < TotalCubes)
    {
        WinText.text = "Have not won yet.";

        yield return Timing.WaitForOneFrame;
    }

    WinText.text = "You win!";
}
```

If you want to pause for some number of seconds rather than just one frame then you can use `Timing.WaitForSeconds`. So this,

```
IEnumerator _CheckForWin()
{
    while (_cubesHit < TotalCubes)
    {
        WinText.text = "Have not won yet.";

        yield return new WaitForSeconds(0.1f);
    }
    WinText.text = "You win!";
}
```

Turns into this:

```
IEnumerator<float> _CheckForWin()
{
    while (_cubesHit < TotalCubes)
    {
        WinText.text = "Have not won yet.";

        yield return Timing.WaitForSeconds(0.1f);
    }

    WinText.text = "You win!";
}
```

SlowUpdate

Unity's coroutines don't have a concept of a slow update loop, but MEC coroutines do.

The slow update loop runs (by default) at 7 times a second. It uses absolute timescale, so when you slow down Unity's timescale it will not slow down SlowUpdate. SlowUpdate works great for tasks like displaying text to the user, since if you were to update the value any faster than that then the user wouldn't really be able to see those rapid changes anyway.

There are two major differences between using SlowUpdate and just always yielding with "yield return Timing.WaitForSeconds(1f/7f);". The first is the absolute timescale, and the second is that all SlowUpdate ticks all happen at the same time. This is important, since changing a text box 7 times a second only looks good if all text boxes change during the same frame.

```
Timing.RunCoroutine(_UpdateTime(), Segment.SlowUpdate);

private IEnumerator<float> _UpdateTime()
{
    while(true)
    {
        clock = Timing.LocalTime;

        yield return 0f;
    }
}
```

SlowUpdate also works well for checking temporary debugging variables. For instance, if it takes a long time to rebuild your project you might set up a public bool in your script that will reset the values on that script. You'll need to check if the value of that bool has been set to true periodically, and the perfect period to do that check is on SlowUpdate. When the user checks the checkbox it will feel like it responds immediately, but it will use far less processing in your app

to check it every 1/7th of a second than 30 – 100 times per second (depending on your framerate).

NOTE: Unity's Time.deltaTime variable will not return the correct value while in SlowUpdate, because Unity's Time class knows nothing about this segment. Fortunately you can use Timing.DeltaTime instead when in SlowUpdate.

You can change the rate that SlowUpdate runs.

```
Timing.Instance.TimeBetweenSlowUpdateCalls = 3f;
```

The line above will make SlowUpdate only run once every 3 seconds.

Tags

When you start a coroutine, you have the option of supplying a tag. A tag is a string that identifies that coroutine. When you tag a coroutine or a group of coroutines you can later kill that coroutine or that group using KillCoroutine(tag) or KillAllCoroutines(tag).

```
void Start ()
{
    Timing.RunCoroutine(_shout(1, "Hello"), "shout");
    Timing.RunCoroutine(_shout(2, "World!"), "shout");

    Timing.RunCoroutine(_shout(3, "I"), "shout2");
    Timing.RunCoroutine(_shout(4, "Like"), "shout2");
    Timing.RunCoroutine(_shout(5, "Cake!"), "shout2");

    Timing.RunCoroutine(_shout(6, "Bake"), "shout3");
    Timing.RunCoroutine(_shout(7, "Me"), "shout3");
    Timing.RunCoroutine(_shout(8, "Cake!"), "shout3");

    Debug.Log("Killed " + Timing.KillAllCoroutines("shout2"));
}

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

    Debug.Log(text);
}
```

```
// Output:  
// Killed 3  
// Hello  
// World!  
// Bake  
// Me  
// Cake!
```

LocalTime and DeltaTime

MEC keeps track of the local time inside each segment and keeps the LocalTime and DeltaTime variables updated. The default Time class in Unity will work fine most of the time, but won't work right in the SlowUpdate, and it is completely unavailable in the EditorUpdate and EditorSlowUpdate segments.

Unity's Time class returns time as a float value. Floats store their values inside 4 bytes in a rather complicated format in which the decimal point can float around inside the number. This means that the larger the number you have to the left of the decimal point the more you have to round the number to the right. Unity's time counts the number of seconds since your app started, so the more seconds that pass the less precision you get with the fractions of a second. After about 2 hours of running, most Unity apps will start to show things jumping around on the screen a little because of this rounding. After about 8 hours of running the app can become unplayable.

This is why MEC keeps track of time as a double rather than a float. Doubles use 8 bytes to store the value which means that it takes 32 times as long to get to the same level of rounding error that a float does. For really long running apps, MEC provides the function ResetTimeCount, which will return the counters to zero (only for the Update, FixedUpdate, and LateUpdate segments though). To access time as a double use Timing.Instance.localTime.

Additional Functionality

There are three helper functions that are also included in the Timing object: CallDelayed, CallContinuously, and CallPeriodically.

- CallDelayed calls the specified action after some number of seconds.
- CallContinuously calls the action every frame for some number of seconds.
- CallPeriodically calls the action every "x" number of seconds for some number of seconds.

All three of these could easily be created using coroutines, but this basic functionality ends up being used so often that we've included it in the base module.

```
// This will start the _RunFor5Seconds coroutine, 2 seconds
from now.
Timing.CallDelayed(2f, delegate {
Timing.RunCoroutine(_RunFor5Seconds(handle)); });

// This does the same thing, but without creating a closure.
// (A closure creates a GC alloc.)
// "handle" is being passed in to CallDelayed and CallDelayed
passes
// it back to RunCoroutine as the variable "x".
Timing.CallDelayed<IEnumerator<float>>(handle, 2f, x => {
Timing.RunCoroutine(_RunFor5Seconds(x)); });
```

```
private void PushOnGameObject(Vector3 amount)
{
    transform.position += amount * Time.deltaTime;
}

// This will push this object forward one world unit per
second for 4 seconds.
Timing.CallContinuously(4f, delegate {
PushOnGameObject(Vector3.forward); }, Segment.FixedUpdate);

// CallContinuously also has a non-closure version. It's extra
important to try
// not to make closures on CallContinuously, since it will
result in a GC alloc every frame.
Timing.CallContinuously<Vector3>
    (Vector3.forward, 4f, vector => PushOnGameObject(vector),
Segment.FixedUpdate);

// This line is equalivant to the previous line:
Timing.CallContinuously(Vector3.forward, 4f, PushOnGameObject,
Segment.FixedUpdate);
```

By default MEC prints any exceptions out to the console and quits the coroutine that threw the exception. However, if you want to do something different with exceptions then you can define your own custom error receiver:

```
Timing.Instance.OnError = OnError;
private void OnError(Exception exception)
{
    Debug.LogError("The exception was seen: " +
exception.Message);
}
```


If you decide to run with more than one Timing instance then you can define different error handlers for different instances.

FYI: You'll save yourself some trouble if you never point the OnError function to a lambda expression, since lambda expressions break their links whenever the Unity debugger recompiles.. and you're much better off if your error handling code is solid. Instead of a lambda, use a function that is defined in one of the classes in your app.

One MEC coroutine can yield to another MEC coroutine using the WaitUntilDone function. WaitUntilDone can also wait for the WWW object, or anything that returns an AsyncOperation or inherits from CustomYieldInstruction.

```
yield return Timing.WaitUntilDone(objectToWaitFor);
```

CancelWith

MEC coroutines don't tie themselves to the lifetime of a gameobject by default like Unity's coroutines do. In many cases you might make a coroutine that effects game objects in your scene, like one that moves a button from point A to point B over time. Often times you don't necessarily know that the button that you are moving will not be destroyed (perhaps by switching screens) before you are done moving it.

For that type of coroutine it's a good idea to make your coroutine automatically quit when the object it's working on is destroyed. That way it doesn't throw random exceptions when you switch screens or scenes.

With MEC you do that using the CancelWith extension, like this:

```
Timing.RunCoroutine(_moveMyButton().CancelWith(gameObject));
```

FAQ

[Q: Does MEC have a function for WaitForEndOfFrame?](#)

It is not implemented in MEC Free, but MEC Pro has a segment for it.

NOTE: There is some confusion about what WaitForEndOfFrame actually does. When you just want to yield until the next frame then WaitForEndOfFrame is not really an ideal command, it's better to use "yield return Timing.WaitForOneFrame;". Many people use WaitForEndOfFrame

when using Unity's coroutines because it's the closest thing they can find to `WaitForOneFrame` in Unity's default coroutines and they don't realize that it can cause subtle issues. Now with MEC you can use explicit variable names if you want without creating the potential for the visual glitches described below.

[This page](#) has a graph that shows the timing for each frame. As the graph shows, `WaitForEndOfFrame` executes after all rendering has finished. If you were to use that call in a Unity coroutine to move a button across the screen then the button would always be drawn in the position you had set it to on the previous frame. In most cases the frame rate is high enough that you wouldn't notice the difference visually, but this practice can cause subtle visual glitches that are difficult to explain or debug.

For instance, if you had an enemy ship and an "enemy ship explodes" animation it would be common practice to call `Destroy` on the enemy ship object and `Instantiate` the explosion animation at the same time. However, if you did this in a coroutine that had been calling `WaitForEndOfFrame` then the user might see what appears to be the ship blinking for an instant before exploding.

Q: Does MEC have a function for `StopCoroutine`?

A: Yes. It's called `Timing.KillCoroutines()`. It can either take a handle to a coroutine that was returned by a previous `Timing.RunCoroutine` command, or it can take a tag.

NOTE: `KillCoroutine` is for stopping a coroutine function from a different function. If you want to end a coroutine from inside that coroutine's function then the best command to use is "`yield break;`", which is the equivalent of calling "`return;`" in any other function.

Q: Does MEC have a function for `StopAllCoroutines`?

A: Yes. `Timing.KillCoroutines()`. You can also use `Timing.PauseCoroutines()` and `Timing.ResumeAllCoroutines()` if you would rather stop everything temporarily.

Q: Does MEC have a function to yield one coroutine until another one finishes?

A: Yes. From inside the coroutine that you want to hold you call "`yield return Timing.WaitUntilDone(coroutineHandle);`" The handle is returned whenever you call `Timing.RunCoroutine`.

Q: Does MEC completely remove GC allocs?

A: No. MEC removes all per-frame GC allocs. (unless you allocate memory on the heap inside of your coroutine, but MEC has no control over that.) When a coroutine is first created the function pointer and any variables you pass into it are put on the heap and eventually have to be cleaned up by the garbage collector. This unavoidable allocation happens in both Unity's coroutines and MEC coroutines. MEC coroutines do allocate *less* garbage on average than Unity coroutines.

Q: Are MEC coroutines always more memory efficient than Unity coroutines, or is it only in select cases?

A: MEC coroutines produce less GC allocation than Unity coroutines do in all cases, except if you large strings and assign them as the tag for a coroutine.

Q: Reduced GC allocs are great, but are there any other advantages to MEC coroutines over Unity's coroutines.

A: The MEC infrastructure runs about twice as fast as Unity's coroutine infrastructure. For more information on that please watch the video on the performance of MEC vs Unity coroutines, which is linked to at the end of this document.

Unity's coroutines are attached to the object that you started them on while MEC uses a central object to run all its processes. That means that the following three things are true:

1. Unity's coroutines won't start if the current object is disabled. MEC coroutines don't care.
2. If you disable a GameObject then all of the Unity coroutines that are attached to it will quit running (they do not resume on re-enable.) MEC coroutines don't do this unless you explicitly tell them to.
3. If the GameObject that you started a Unity coroutine on is destroyed then all the attached coroutines will also be killed. MEC coroutines also don't do this.

MEC coroutines allow you to create coroutine groups, which give you the ability to pause/resume or destroy whole groups of coroutines at the same time. Unity's coroutines don't allow you to pause and resume coroutines from the outside, and Unity's coroutines are always grouped by the gameObject that they were started on.

Lastly, MEC coroutines allow you to run the coroutine in the LateUpdate or SlowUpdate segment if you want to. MEC Pro has even more segments.

Q: I heard that it was slow to have a bunch of Update functions in my scripts, but I like update functions. Can I use MEC to make my Update functions faster?

A: Yes. You just need to follow these three steps:

1. Add the following function to any utility class. (If you don't already have one then just make a new static class called Util and put this function in it.)

```
public static IEnumerator<float> _RunFunc(System.Action func,
GameObject go)
{
    yield return Timing.WaitForOneFrame;
    while (go != null)
    {
        if (go.activeInHierarchy)
            func();

        yield return Timing.WaitForOneFrame;
    }
}
```

2. In every class where you want to use update more efficiently you need to change the name of the Update function to something other than "Update" (you can name it anything, I'll assume you named it "MyUpdate").
3. Run the above coroutine at the end of each class you changed's Start function, like this:

```
Timing.RunCoroutine(Util._RunFunc(MyUpdate, gameObject));
```

Advanced Control of Process Lifetime

If you do nothing special the Timing object will add itself to a new object named "Movement Effects". All of the coroutine processes will normally be handed out by that instance.

However, if you want more control over things you can attach the Timing object to one of the GameObjects in your scene yourself. You could even create more than one Timing object if you like and add different coroutines to different objects. The functions for Timing.RunCoroutine() are static so they can be accessed from anywhere, but if you have a handle to an instance of the Timing object then you can call yourTimingInstance.RunCoroutineOnInstance() to run the coroutine on that instance. By creating multiple instances of the Timing object you can effectively create groups of processes that can all be paused or destroyed together.

The OnError delegate and the TimeBetweenSlowUpdateCalls variable are also attached to the Timing instance, so you can set different error handling for different timing objects or you can set SlowUpdate to run at a different rate.

Example

Here is a simple example of using MEC coroutines:

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

public class Testing : MonoBehaviour
{
    void Start ()
    {
        IEnumerator<float> handle =
Timing.RunCoroutine(_RunFor10Seconds());

        handle = Timing.RunCoroutine(_RunFor1Second(handle));

        Timing.RunCoroutine(_RunFor5Seconds(handle));
    }

    private IEnumerator<float> _RunFor10Seconds()
    {
        Debug.Log("Starting 10 second run.");

        yield return Timing.WaitForSeconds(10f);

        Debug.Log("Finished 10 second run.");
    }

    private IEnumerator<float>
_RunFor1Second(IEnumerator<float> waitHandle)
    {
        Debug.Log("Yielding 1s..");

        yield return Timing.WaitUntilDone(waitHandle);

        Debug.Log("Starting 1 second run.");

        yield return Timing.WaitForSeconds(1f);

        Debug.Log("Finished 1 second run.");
    }

    private IEnumerator<float>
_RunFor5Seconds(IEnumerator<float> waitHandle)
    {
```

```

        Debug.Log("Yielding 5s..");

        yield return Timing.WaitUntilDone(waitHandle);

        Debug.Log("Starting 5 second run.");

        yield return Timing.WaitForSeconds(5f);

        Debug.Log("Finished 5 second run.");
    }
}

```

This is the output:

1. Starting 10 second run.
2. Yielding 1s..
3. Yielding 5s..
4. Finished 10 second run.
5. Starting 1 second run.
6. Finished 1 second run.
7. Starting 5 second run.
8. Finished 5 second run.

Here's an example of how to use MEC from UnityScript:

```

import System.Collections.Generic;
import MovementEffects;

function Start() {
    Timing.RunCoroutine(_TestCoroutine());
}

function _TestCoroutine() : IEnumerator.<float> {
    Debug.Log("TestCoroutine: Starting...");

    var handle : IEnumerator.<float> =
Timing.RunCoroutine(_TestWait());

    yield Timing.WaitUntilDone(handle);

    Debug.Log("TestCoroutine: Finished!");
}

function _TestWait() : IEnumerator.<float> {
    for (var i : int = 0; i < 5; i++) {
        Debug.Log("TestCoroutine: " + i);
        yield;
    }
}

```

```
}  
}
```

Output:

```
1. TestCoroutine: Starting...  
2. TestCoroutine: 0  
3. TestCoroutine: 1  
4. TestCoroutine: 2  
5. TestCoroutine: 3  
6. TestCoroutine: 4  
7. TestCoroutine: Finished!
```

If you would like to know more about the performance of MEC vs Unity's coroutines, take a look at this video: <https://www.youtube.com/watch?v=sUYN8XtuUFA>

Also this video: <https://www.youtube.com/watch?v=CqHl7sgam7w>

MEC Pro can be found here: <https://www.assetstore.unity3d.com/en/#!/content/68480>

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