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 CallContinously functions	✗	✓	✓
You can choose whether to stop or pause your coroutines with a gameobject or not	✗	✓	✓
Compatable with MEC Multithreaded	✗	✓	✓
Pause and continue running coroutines	✗	✓	✓

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	✗	✗	✓
Can control coroutines directly from their handles	✗	✗	✓
Coroutines can be linked so they stop or pause together	✗	✗	✓
Coroutines can hold execution until a function returns true	✗	✗	✓

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 MEC 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 MEC;
```

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);
```

NOTE: Be sure to use CancelWith (see page 6) with the RunCoroutine call for any coroutines that are moving or changing GameObjects or you will start to notice null reference exceptions when you do things like switch screens in the middle of a transition.

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

tendency 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 like you would call a normal function "_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 best, 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!";
    }
```

Unity's default coroutines allow you to use yield return as a shorthand syntax to create one coroutine and wait for it to finish before continuing:

```
    IEnumerator _EnableHappyFace()
    {
        yield return _CheckForWin();
        HappyFaceObject.SetActive(true);
    }
```

The above function starts the _CheckForWin coroutine function, and the _EnableHappyFace coroutine function remains paused until _CheckForWin actually finishes, then it enables the HappyFaceObject. The yield return statement above is equivalent to "yield return StartCoroutine(_CheckForWin());", but Unity calls StartCoroutine behind the scenes for you.

MEC Pro has a shorthand version that calls RunCoroutine automatically as well, but in MEC Free you have to use the longhand version:

```
    IEnumerator<float> _EnableHappyFace()
    {
        // This line can replace Unity's default coroutines in
    both MEC Free and MEC Pro
        yield return
    Timing.WaitUntilDone(Timing.RunCoroutine(_CheckForWin());
```

```
        // This shorthand version only works in MEC Pro
        yield return Timing.WaitUntilDone(_CheckForWin());

        HappyFaceObject.SetActive(true);
    }
```

CancelWith

In many cases you might make a coroutine that affects game objects in your scene, like one that moves a button from point A to point B over time. MEC coroutines don't automatically stop running when the GameObject that they were created on gets destroyed or disabled like Unity's coroutines do. The reason that this behavior is not included in MEC by default is because it doesn't always make sense for all coroutines. If your coroutine doesn't work on UI elements then at best this check is a waste of processing and at worst it can lead to undesired behavior.

When you're working with UI elements it's a good idea to turn this check on so that you don't get errors when changing screens. With MEC you do that using the CancelWith extension, like this:

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

TLDR: You should use .CancelWith on all coroutines that affect UI elements.

CancelWith *does* increase the size of the unavoidable GC alloc that all coroutines generate by approximately 20 bytes. 20 bytes isn't large, but if you want to avoid every possible GC alloc then you can relatively easily do what CancelWith does without using the function. Just make sure that you make the following check after every yield return statement inside your coroutine:

```
if(gameObject != null && gameObject.activeInHierarchy)
```

Pause and Resume coroutines

MEC coroutines can be paused and later resumed. Unity's coroutines don't do this but the concept is very simple.

```
Timing.PauseCoroutines(handleToACoroutine);
// somewhere else in code...
Timing.ResumeCoroutines(handleToACoroutine);
```

It's fine to pause or resume your coroutines when they are already in that state. For instance you could have a coroutines that moved the camera around but which paused it every frame in which an OnDrag event fired, then call resume once when the drag was released.

If the coroutines has yielded to a WaitForSeconds call then the seconds will not count down while that coroutines is paused.

WaitUntilDone

Unity's default coroutines have several cases where you can yield return some variable. For instance, you can "yield return asyncOperation;". MEC also has a function to do that, and that function is called WaitUntilDone.

```
yield return Timing.WaitUntilDone(wwwObject);
yield return Timing.WaitUntilDone(asyncOperation);
yield return Timing.WaitUntilDone(customYieldInstruction);

// With MEC Pro you can do a little more with WaitUntilDone:
yield return Timing.WaitUntilDone(newCoroutine);
 // The above automatically starts a new coroutine and holds
the current one.

yield return
Timing.WaitUntilTrue(functionDelegateThatReturnsBool);
yield return
Timing.WaitUntilFalse(functionDelegateThatReturnsBool);
```

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 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. Timing.DeltaTime and Timing.LocalTime will always have good values for time inside a MEC coroutines no matter what timing segment you're running in.

You can also change the rate that SlowUpdate runs if you like.

```
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. That's why it's almost always better to use Timing.LocalTime rather than Time.time inside a MEC coroutine.

Additional Functionality

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

- CallDelayed calls the specified action after some number of seconds.
- CallContinously 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 _RunFor5Seconds, 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);

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

Fluid Architecture

Some people prefer a fluid syntax when they write code because it can be easier to read and look more modern. This syntax is a completely optional alternative way to use MEC.

```
// The normal way to run a coroutine
Timing.RunCoroutine(_Foo().CancelWith(gameObject));

// The fluid way to run a coroutine
_Foo().CancelWith(gameObject).RunCoroutine();
```

So here you have “coroutine function”.”modifier”.”modifier”.”RunCoroutine” rather than “RunCoroutine(“coroutine function”.”modifier”);” It’s called fluid because the logic kind of flows from a starting point to a conclusion.

Remember that, like the other way of running your coroutines, if you forget to use RunCoroutine then the compiler won’t complain, but it will not execute the coroutine. This fact can be a little harder to remember when the call to Run is at the end of the line. Also keep in mind that the fluid syntax may confuse other developers who are used to working with Unity’s default coroutines, since Unity’s default coroutine API doesn’t support that syntax.

The end result is exactly the same, and neither way is more performant than the other (that’s not always true of fluid architecture, but it’s true in this case.) It’s just a matter of which syntax you like better.

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 an ideal command, it's better to use "yield return null;". 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. MEC defines the constant Timing.WaitForOneFrame, so with MEC you can use explicit variable names if you want without creating the potential for the visual glitches and decreased performance that using EndOfFrame can cause.

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 Unity coroutine that had been calling WaitForEndOfFrame then the user might see what appears to be the ship blinking for an instant before exploding.

Also, canvases normally recalculate just after the Update segment, so Unity's WaitForEndOfFrame can have a serious negative effect on performance if you use it wrong because it will cause the canvas to have to recalculate twice per frame.

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. The reason yield break is better is because the KillCoroutine command cannot end the coroutines function *that is currently running*, so that function will continue to execute until the next yield command, but yield break doesn't have this problem.

Q: Does MEC have a function for StopAllCoroutines?

A: Yes. Timing.KillCorutines(). You can also use Timing.PauseCorutines() and Timing.ResumeAllCorutines() 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 allocate 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 (unless you use CancelWith or PauseWith to tell them to 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.

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

MEC Pro also has a *ton* of additional features that allow you to run coroutines in fantastic new ways.

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 ()
    {
        CoroutineHandle 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(CoroutineHandle
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(CoroutineHandle
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.
```

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/