Gameplay Effects

# Capabilities of GAS

## What is a gameplay effect?

* Object of type UGameplayEffect
* Used to change attributes and Gameplay Tags
* Tags used to identify things
* Data ONLY
* No logic added
* BPs are created based on Gameplay Effect Classes
* Don’t Subclass UGameplayEffect
* Change attributes through:
* Modifiers
* Executions

## Modifier Op

* Modifiers come in various flavours including
* More complex custom calculations
* For intricate and specific attribute changes
* Specify a type of operation to be performed
* Take a value
* Called a ‘magnitude’
* Uses Magnitude to change attribute in a way that depends on the operation
* Types of Operation
* Add
* Use negative value for subtract
* Multiply
* Divide
* Override
* Change value for a specified value

## Magnitude calculation type

* Scalable float
* Simplest type
* Can specify hard-coded value directly
* Use table to scale based on GamePlayEffect level
* Attribute Based
* Uses another attribute eg “Add value to player Health = player Strength’
* Further customisable eg ‘Add to Health Str x 10’
* Custom Calculation Class (MMC)
* Can Capture other values eg attribute or other variables and use in calculation of arbitrary complexity.
* “Modifier Magnitude Calculations”
* Set by Caller
* Key value pair
* Assign a magnitude associated with a name or gameplay tag
* Useful to set magnitude based on code logic at time Gameplay Effect is created/applied

## Executions

* Aka Gameplay Effect Execution Calculation / ExecCalc
* Can change more than one attribute
* Or anything else we choose to code into them
* Most powerful way to modify attributes

A screenshot of a computer program

Description automatically generated

# Duration Policy

* Instant
* Has Duration
* Infinite
* Affect Attribute until manually removed

Stacking

* Own Policies on how to stack

Add Gameplay Tags

Grant abilities

Can be applied directly but usually we create a more lightweight version of them called a Gameplay Effect Spec

* Spec common concept in GAS
* Contains barebones info to perform modification
* Only instance of the effect class is the Class Default Object

Gameplay Effect class is versatile enough that it never really needs to be subclassed into child classes

Use MMC or ExecCalc for complex calculations!

Effect Spec carries info with it as well

* Gameplay tags
* Effect Context
* Additional class that contains info about the effect being applied
* References to cause and target of effect to allow trigger of gameplay effects

Gameplay Effects often applied from within Gameplay Abilities, but don’t have to be; an actor may have a gameplay effect it wishes to apply to some other actor that overlaps, like the Health potion!

# Instant Gameplay Effects

A screen shot of a computer program

Description automatically generated

A computer screen with green text

Description automatically generated

A computer screen with green text

Description automatically generated

A computer screen with green text

Description automatically generated

Stripping all this out makes it more versatile because the collisions and meshes etc are better set in BP

One possible way to apply effects:

A screen shot of a computer

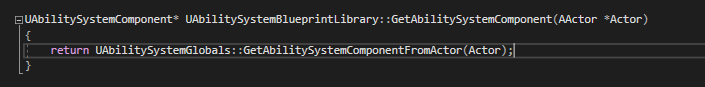
Description automatically generated

Alternatively, static function library



Returns an ability system component

How does this function work? The definition:



It calls AbilitySystemGlobals::GetAbilitySystemFromActor()

That function:

A screenshot of a computer program

Description automatically generated

If the actor passed into this is null, it returns a nullptr. But it casts Actor to Ability system Interface and if that cast succeeds it returns AbilitySystemComponent on that interface

It also has a bool look for component and if that cast fails it uses FindComponentBy Class, so it’s pretty powerful. If there are actors in the world that do not implement that AbilitySystemInterface; in this project there usually are, but it’s possible that there might be some that do not in other projects so this global library handles those cases.

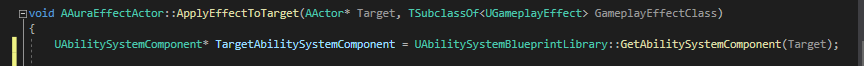
This is a bit more sophisticated than the method in this project, so it’s a great alternative

Reminder of the one in this project:

A screen shot of a computer

Description automatically generated

So, lets use that:



A screenshot of a computer

Description automatically generated

Using ApplyGameplayEffectSpectToSelf() we see the arguments:

A screenshot of a computer

Description automatically generated

1. Const ref to a GameplayEffect of type FGameplayEffectSpec
2. FPredictionKey (lag compensation) (optional)

### How do we make a GameplayEffectSpec?

Since we need a GameplayEffectSpec to pass in we should create that first. We can make one easily, because the AbilitySystemComponent has the ability to do that. We’ll use the target’s ability system component, because any ASC can make an EffectSpec given some gameplayeffect class

So, here’s how to make the Outgoing Spec (commenting out the Apply line until we’re ready for it)

A screenshot of a computer

Description automatically generated

Fn takes a TSubclassOf<UGameplayEffect> as first argument, in this case GameplayEffectClass will work very well, then a float Level (gameplay effects can have their own levels and we’ll add these later, for now 1.f is fine)

A gameplay effect is closely related to an *Effect Context* – what is the context of the effect, what’s the situation, who’s causing it, who’s the target, is it fire/lightning etc. Context is something we associate with the effect spec by passing in an *effect context handle* – a lightweight thing that contains a pointer to the context itself.

### How do we make a context handle?

The ability system component has a function for that! It has functions for everything!

A screenshot of a computer

Description automatically generated

(commenting out the Make Spec line until we’re ready for it!)

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

This does not take any inputs, it’s just a function that makes an effect context and returns a handle to it.

Returns a FGameplayEffectContextHandle; usage:



Let’s examine and understand these new types

Declaration for the FGameplayEffectContextHandle



Definition in GameplayEffectTypes.h :

A screenshot of a computer

Description automatically generated

So, it makes the context or it’s subclass polymorphic and able to replicate, so we can subclass to add fire or lightning or whatever

The Handle is a lightweight wrapper that stores the actual effectcontext as a pointer, called Data

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

As we see, Data is a TShared pointer of type FGameplayEffectContext, so data within this struct is the true context, the handle itself is just a wrapper with a couple of utilities

It can clear the pointer

A computer screen with white text

Description automatically generated

There’s an ISValid function to check it’s valid

A screen shot of a computer code

Description automatically generated

We have a way to get any GameplayTags that EffectContext might have

A screen shot of a computer

Description automatically generated

There’s loads in here

At the heart of the Handle is the context itself – Data

So MakeEffectContext() is just a way to make that wrapper that contains an internal gameplayeffectcontext and we can set things on that context

So we can, for example, take that EffectContextHandle and add a source object.

A screen shot of a computer

Description automatically generated

So if we want to know what caused this effect we can store a source object – looking at the definition for the function we see it uses Data inside and calls AddSourceObject

A screen shot of a computer program

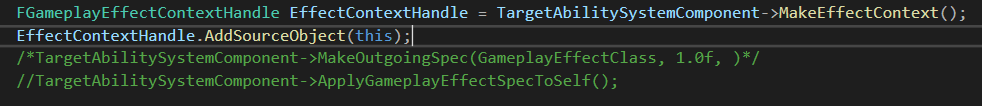
Description automatically generated

And the definition of THAT is

A screen shot of a computer program

Description automatically generated

It takes the UObject and sets an internal member variable called SourceObject (a weak pointer) so it won’t affect garbage collection . So the GameplayEffectContext has the ability to store things related to the context of the gameplay effect . In the case below I’m storing the source object in case it’s needed later



Now we have a context we can uncomment the next line and pass it in



Now MakeOutgoingSpec() returns an FGameplayEffectSpecHandle, so we can store it in a local of that type



A screenshot of a computer program

Description automatically generated

Looking at the definition of the FGameplayEffectSpecHandle we see it allows BPs to generate DES once and then reference by handle to apply multiple times/targets

Not that it has it’s own Data of type GameplayEffectSpec

We now have the Handle, that has the class, level and handle set for it

Now we can uncomment 

This takes an FGameplayEffectSpec, but we only have the FGameplayEffectSpecHandle – we need to get the Data from inside the Handle!

Note that it takes an FGameplayEffectSpec by const reference – not by pointerA screen shot of a computer program

Description automatically generated

We know that the Data is a pointer inside the handle

So we take the handle.data and call Get()



A black screen with white text

Description automatically generated

This returns the raw pointer, but we need a reference. Since we cannot use the pointer we have to dereference the pointer.



That whole function then is:

A computer screen with text

Description automatically generated

## Using the function in Blueprints

A screenshot of a computer

Description automatically generated

Apply Effect To Target is now callable in the Effect Actor BP and children!

Can pass in the Actor and GameplayEffect class:

A screenshot of a computer

Description automatically generated

However, the only current option is the base class, so we need to make some effects

As discussed, we do this by subclassing the GameplayEffect Class

In BP:

A black and grey rectangular object

Description automatically generated with medium confidence

For a health potion, it’s instant and ends, so periodicity is not applicable

A screenshot of a computer

Description automatically generated

Similarly, stacking types are not appliable here, this is also more for ongoing effects

A screenshot of a black screen

Description automatically generated

Theres no granted abilities

A screenshot of a computer

Description automatically generated

This is what we are interested in!

Health potions are not always on (infinite) and not over time, so Instant

A black rectangular object with white lines

Description automatically generated

Modifiers is an array, so add an element to affect an attribute

A black rectangular object with white text

Description automatically generated with medium confidence

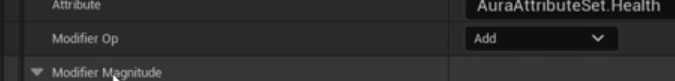
A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

The 4 stats from AuraAttributeSet!



To add to the attribute

And the magnitude:

A screenshot of a computer

Description automatically generated

Scalable Float, at is simplest, can just add a float value and that’s it.

We could also Use Curve Table (currently there are not options for this implemented)

A screenshot of a computer

Description automatically generated

Executions are also here, and there’s a possible curve table driven chance to apply to target system

A screenshot of a computer

Description automatically generated

Back n the potion BP:

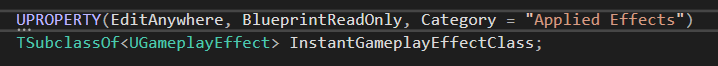
A screenshot of a computer

Description automatically generated

The Instant Potion Effect is not visible because we forgot to expose it in code!

A black screen with white text

Description automatically generated



A screenshot of a video game

Description automatically generated

Now it’s usable!

A screenshot of a computer

Description automatically generated

Now we have a BP for an actor that applies an effect on overlap

Since ‘Target’ is ambiguous in BPs renaming the variable to TargetActor

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

Gameplay effects can be applied directly in BP without having to call the BP callable C++ function

The code function involves getting the ASC for the target simply so we can use it to make a context and apply an effect after making an effect spec

We can also do this in BP quite simply

Take Other Actor and call GetAbilitySystemComponent

A screenshot of a computer

Description automatically generated

If we look at the target it’s AbilitySystemBlueprintLibrary, which is the function we called in the code!



Input is Actor, return is the ASC

Once we got that, we made a context 

And this can be done in BP as well

A screenshot of a computer screen

Description automatically generated

Returns a GameplayEffectContextHandle structure, just like Make Effect context in code

A screenshot of a computer

Description automatically generated

In C++ we called AddSourceObject; the definition is here in GameplayEffectTypes:

A screen shot of a computer program

Description automatically generated

And is inside the FGameplayEffectContextHandle , but this function is not BP callable

A screen shot of a computer program

Description automatically generated

Same with GetSourceObject

However, in BPs we can still use GetSourceObject!

A screenshot of a computer

Description automatically generated

But how to call it if it’s not BP callable?!

Because this is not a function on this struct, it’s a function on AbilitySystemBlueprintLibrary

BP Libraries are a great way to expose to BP certain things you can only do in C++

While we can’t call AddSourceObject, we could make a Static Function Library of our own and create a static function take takes in an effect context and call a function, like AddSourceObject

Then we made an outgoing spec



This is an ability on the ASC, so in BP we drag off of ASC

A screenshot of a computer

Description automatically generated

Notice that this requires a GameplayEffectClass; we have that already (the instant) and can set the level, and it takes a GameplayEffectContextHandle, which is what the Make Effect Context returns

A screenshot of a computer program

Description automatically generated

We now have an outgoing spec

The last thing we did in the code was call ApplyGameplayEffectSpecToSelf on the ASC



We can do that in BP too

A screenshot of a computer

Description automatically generated

A screen shot of a game

Description automatically generated

We can plug in a Spec handle, the outgoing value from the Make Out Going Spec node

A screenshot of a computer

Description automatically generated

It looks a little cleaner in BP that in code, in code we had to get the EffectSpecHandle and get the data pointer wrapper raw pointer and dereference it…it’s easier and quicker in the BP, but C++ has more options as not everything is exposed. Also in C++ we did all that work right inside the callable function

A screen shot of a computer screen

Description automatically generated

So the end result is the BP is cleaner, as there’s only a single node.

In this case, because it’s pretty simple, there’s no significant performance difference between the two. We can even add the check from OtherActor!

A screenshot of a computer

Description automatically generated

# Duration Gameplay Effects

Instant is just one possible option for gameplay effects; there’s also Infinite and Has Duration.

To show this off, we need to make a new BP Class for a duration effect

Set Has Duration and specify the magnitude of the duration, how many seconds is it applied before it removes itself?

A screenshot of a computer

Description automatically generated

A screenshot of a computer

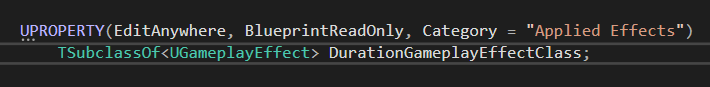
Description automatically generated

A screenshot of a computer

Description automatically generated

In this effect, the MaxHealth will be increased by 100. It’s already at 100, so we should see the globe go DOWN, because the percentage of health is the same value divided by twice as much

New class for this effect:



# Periodic Gameplay Effects

Eg Damage/Heal over time

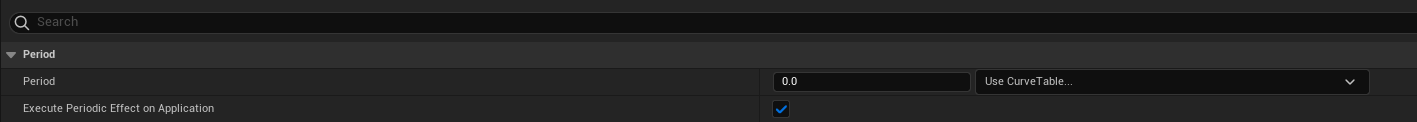
Attributes have a base and a current value.

Temporary Effects affect the current value, when removed this effect is undon and the current returns to the base.

* Instant effects seem to be permanent, because they affect the base with a permanent change.
* When we apply an instant effect we don’t apply to current because we will not undo it later.

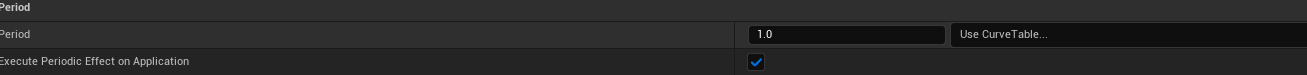
<periodic img>

* This is different to how Duration and Infinite effects work.
* They cause changes to Current and are undone when removed.
* Duration effects remove themselves
* Duration is a set time
* Infinite effects don’t remove themselves
* Have to be removed manually.
* Infinite and Duration effects can be converted into Periodic
* Execute a change to an attribute periodically over their duration
* Periodic are a special type of Duration
* But the changes they execute are treated like Instant
* Change the base value permanently
* When a periodic effect is removed the changes that were made remain intact
* They are not undone by removal of the effect



Non-periodic effect

Adding a period:



Now the effect is applied every 1 seconds



Execute periodic effect on application is a bool; if ticked then as soon as the effect is applied we get the result. Otherwise, we get it at the next period tick (in this case after 1 sec)

This setting will wait 1 sec, then increase MaxHealth to 200, then to 300 a second later



But then it will leave it as it is!

To heal we can set to affect Health and adding 1 per tick with a tick time of 0.1 (10 times a sec) – gives a nice smooth movement of the globe. 0.1 / 0/01 sec is even smoother, BUT remember this is predictive and register these changes with the Ability System each time, so it’s not a trivial setting of a variable – there are potentially other things that happen as a result of the application of Gameplay effects.

You CAN apply them this rapidly…but it’s better to find other ways to get that smoothness!

We can interpolate with tricks on the widget rather than forcing the ability to affect changes so rapidly!

NB with Execute periodic on application ticked, the crystal heals over 2 seconds for 26:

A close-up of a number

Description automatically generated

Turning it off heals correctly for 25

A screenshot of a computer

Description automatically generated

This is because there’s an extra tick for the instant application

A black and white screen with white text

Description automatically generated

Inhibition policy allows you to block execution of a gameplay effect, based on Tags

Stacking type:

A screenshot of a computer

Description automatically generated

What happens when you have multiple effects of the same kind at once?

## Effect Stacking

Because an item has a duration, we can configure it’s stacking options.

A black and white screen

Description automatically generated with medium confidence

Stacking type:

A screenshot of a computer

Description automatically generated

Without stacking, all similar effects are applied at the same time with their own durations

End up with 3x mana as 1 crystal

So we can set stacking. But what behavior do we want? If we pick up 3 mana crystals do we want to see the same mana rate but refresh the duration maybe?

Aggregate by Source:

* Set stack limit count to 2
* Stack Limit count is enforced per source
* One source applies a GE to a target we count it as 1 stack of GE
* Aggregated on the source
* Same source applies effect again
* We count this as second stack
* Aggregated by the source
* Same source applies a 3rd time:
* Exceeds limit, that effect cannot be applied
* We’re aggregating by source so each source keeps a count of the stacks it applies
* A new source applies the same GE
* Starts a new count on stack
* Target now has 2 stacks from source 1, 1 from source 2
* Does not exceed the stack limit count
* Max 2 stacks per source

<stacking 1>

Aggregate by Target:

* Set stack limit count to 2
* Stack Limit count is enforced by target
* One source applies a GE to a target we count it as 1 stack of GE
* Aggregated on the target
* Same source applies effect again
* We count this as second stack
* Aggregated by the target
* Same source applies a 3rd time:
* Exceeds limit, that effect cannot be applied
* We’re aggregating by Target so each Target keeps a count of the stacks it applies
* A new source applies the same GE
* Exceeds limit, that effect cannot be applied

<stacking 2>

# Infinite Gameplay Effect

A screenshot of a computer

Description automatically generated

In this case, overlapping causes -5 health/tick, but it is never removed and drops into the negatives!

We have instant, duration and Infinite gameplay effects, we need to decide how the actor applies and removes these, and we want to change how they are removed from the BP for flexibility

Need to create a policy – an Enum, for application and removal. A setting we can set that determines whether we apply an effect or on end not etc

A screenshot of a computer

Description automatically generated

In the cpp we make a scoped enum (only applies within this class

A black background with white text

Description automatically generated

UENUM and BP type so we can use it

Constants will be:

A screen shot of a computer program

Description automatically generated

If we create a variable of this type for each Effect Class we can set it in BP

A screen shot of a computer program

Description automatically generated

We can also set up an Enum for removal policy:

A computer screen with white text

Description automatically generated

Nb Infinite is the only type that needs a removal policy

A black screen with white text

Description automatically generated

So, how do we handle application and removal?

Eg Instant Effect: check policy, if the policy is DoNoApply then we don’t apply it

DoNotApply is a good default policy

A screen shot of a computer code

Description automatically generated

We can also have functions for Begin and End overlap to call in BP to handle these; all we really need is to receive a target actor; no need to pass in the effect, just have them set and the actor can access it’s own effects

Also: add a bool for actor destroy:

A screen shot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

Nb if we are setting this to apply on Overlap we need to be sure our GameplayEffectClass is set; otherwise there will be a crash thanks to the check in ApplyEffectToTarget(). We cannot set this to an incompatible Enum constant unless we have an instant effect to apply!

Similarly, if the effect is to apply when overlap \*ends\*:

A screen shot of a computer program

Description automatically generated

This is the simplest case.

Duration effects:

Duration effects remove themselves, so no need to check removal policy

We DO need to check the application policy

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

Infinite effect is more complicated because it can be removed! We want it to be moved on end overlap, but only if the effect has already been given to the actor in question. How do we check the actor has that effect and how do we remove it?

## Removing Infinite Effects

So, new goal now is to remove Infinite effects from the target actor is the removal policy is set to EndOverlap. So, how do we do that?

Let’s go back to where we apply the effect, where we have ApplyEffectToTarget

A computer screen with text on it

Description automatically generated

ApplyGameplayEffectSpecToSelf() has a return type – it’s not a void function, it’s FGameplayEffectHandle

A screen shot of a computer program

Description automatically generated

Once you have applied a gameplay effect, that effect becomes active and these apply functions return a handle to that effect, so we can always use that handle for something later, such as removing infinite gameplay effects. So we can store the handle for later; but we need to be careful about how it is stored. For one player character picking up effect actors we could store that active gameplay effect and then OnEndOverlap take that OtherActor and remove the active effect, but things are not that simple. There might be multiple actors overlapping, some of them might have AbilistySystemComponents and some might not, and we cannot assume that TargetActor and EndOverlap is the correct actor for any given effect handle, and if multiple actors are overlapping and we set a single effect handle variable, then we are over-writing it and losing the the effect handle that the variable stored before – needs a lot of care.

One approach is that as we apply a gameplayeffect, if that GE is infinite duration we can store the handle and also the actor it’s applied to – linking the two together.   
  
A Map data structure would be good for this

We **could** make 2 functions; one for Instant and Duration effects and one for Infinite

But in the name of practice let us figure out a way to determine if a given effect is infinite or not!

(so check at runtime the duration policy for a given effect)

To check the duration policy we need to examine the EffectSpecHandle

We can take the EffectSpecHandle 

And we know it has a Data member



So we can access that.

Data is a TSharedPointer

So we can get the raw pointer by calling .Get(), which returns us a pointer to FGameplayEffectSpec

A screenshot of a computer program

Description automatically generated

This is the EffectSpec, and what we want from it is the GameplayEffect itself. The pointer can be dereferenced and the gameplay effect itself is called Def



Def is a TObjectPointer of type UGameplayEffect

Being a TObjectPtr means we have to call .Get(), which gives us the raw pointer needed to get access to all the items on the gameplay effect itself!

A screenshot of a computer

Description automatically generated

One of those items is DurationPolicy



NB: We’re accessing the class default properties here

A screen shot of a computer program

Description automatically generated

So we **don’t** want to use this method to change the duration policy at runtime, for example!

We just want to look at it, and we’re assuming that it doesn’t change at runtime. And we can see if it’s equal to an EGameplayEffectDuration type

A screen shot of a computer

Description automatically generated

Because that’s what the duration policy is, it’s just an EGameplayEffectDurationType, an Enum

If this expression returns an Infinite, we know that as a statement it’s true so we can use it to create a const bool



(Now, if we wanted to be able to remove Duration based effects OnEndOverlap we could also give our Duration based effect it’s own removal policy, just like we have for Infinite )

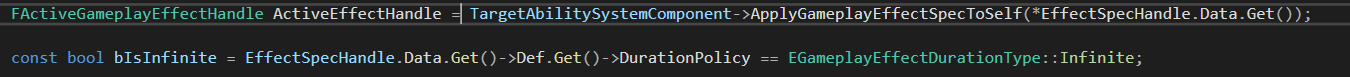
If the effect is an Infinite effect we want to be able to store it somehow

ApplyGameplayEffectToSelf returns a an ActiveGameplayEffectHandle, and that’s the thing we want to store and link to the target actor

A screen shot of a computer program

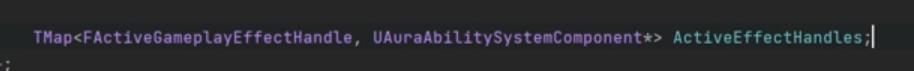
Description automatically generated

So we can make the ActiveGameplayEffectHandle in a local variable



And if bIsInfinite is true, we’ll store it in a map, and the values can be the TargetActor or, even better, the GameplayAbilitySystemComponent – that way we don’t have to dig into the actor to recover the ASC, we have it already!

So the map in .h will be:

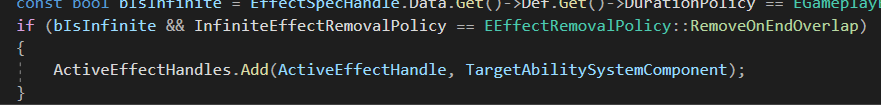


And in the cpp:

A computer screen with text

Description automatically generated

We should only be storing this if we are planning on removing it later though; if there’s no plan to remove the infinite effect, there’s no need to store a handle to the infinite effect. So we should also be checking the removal policy. If it’s a DoNoRemove we shouldn’t bother storing it. If it’s a RemoveOnEndOverlap



So now we need to implement applying and removing for Infinite Effects

A screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generated

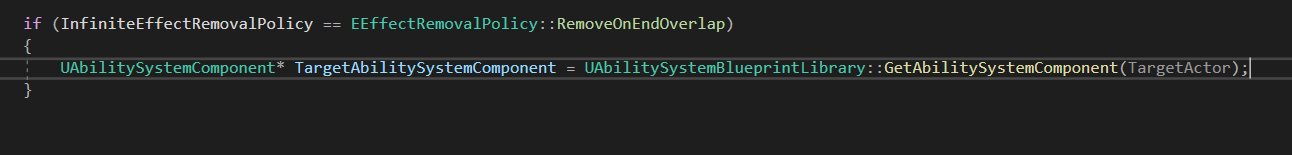
This takes care of applying effects, either on overlap or on end overlap.

Removal:

A screen shot of a computer program

Description automatically generated

First we need to get the TargetActor’s ASC



This is the same line as in ApplyEffectToActor()

Check validity

A screen shot of a computer

Description automatically generated

Then we loop through the Map checking for each key-value pair if there’s an ASC linking an active effect to the ASC that we’re referencing here

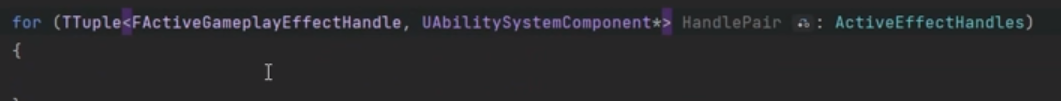
A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated



Long version would be 

So, we take the HandlePair and see if the Value is equal to the ASC

A screenshot of a computer program

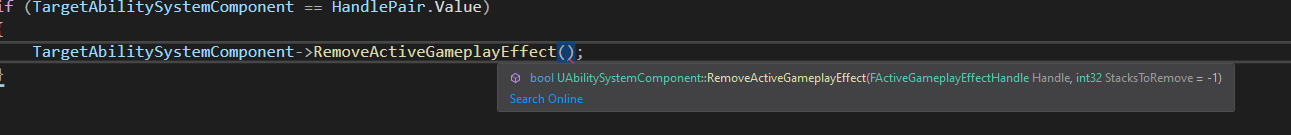
Description automatically generated

If it is, we can remove the gameplay effect

A computer screen with text

Description automatically generated

Requires arguments:



For the FActiveGameplayEffectHandle, we already have one, it’s the key from the map!

A screen shot of a computer

Description automatically generated

We have now removed the gameplay effect! If we have removed the gameplay effect, we should also remove the pair from the Map

However, we are at this point looping through the Map; we cannot remove an element from the container we are currently looping through! That’s bound to cause a crash. So instead as we loop we can accumulate pairs in a container that we will remove when the for loop is completed – make an array

A screen shot of a computer program

Description automatically generated

After an effect is removed it is added to the array, which is scoped outside of the loop but inside the function

To clear the array we need a new loop

A black screen with white text

Description automatically generated

This now removes that ActiveGameplay Effect from the ASC that we linked to it when we added the key-value pair of effect handles to ASC when it was applied.

SO, now the OnOverlap will Apply the Gameplay Effect if the policy says it’s OK, and OnEndOverlap it checks the Map, sees if there’s an ASC that Matches the EndOverlap actor and if it does mark that handle to be removed immediately after!

Change to the Editor:

A screenshot of a computer

Description automatically generated

Set the application policy:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

We can use the default for RemoveOnEndOverlap

Now we just need to be sure to call the BP Callable functions on Begin Overlap and On End Overlap, since we are no longer using ApplyEffectToTarget for this!

A screenshot of a computer program

Description automatically generated

We can also test the stacking for this fire actor:

A black background with white arrow

Description automatically generated

We can have 3 stacks at a time, and as we end overlap we remove those stacks

Now when we overlap 1 volume we get -5 health, 2 -10 and 3 -15.

However, if we leave the 3 volume and move the the 1 volume, we get NO changes to health, when we should be getting -5.

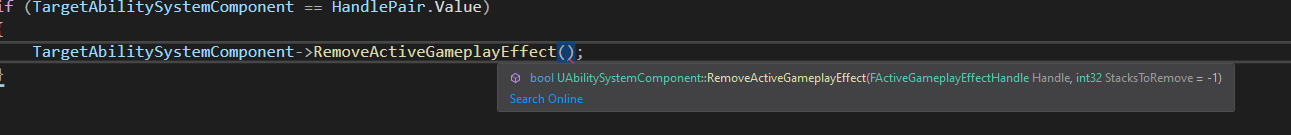
This is because when we exit the x3 stacked overlap we are removing the entire overlap effect!

Back in code:

A screen shot of a computer code

Description automatically generated

As a reminder here was the required arguments:



We did not specify stacks to remove so it removed them all.

If we specify stacks to remove we get the correct behavior:



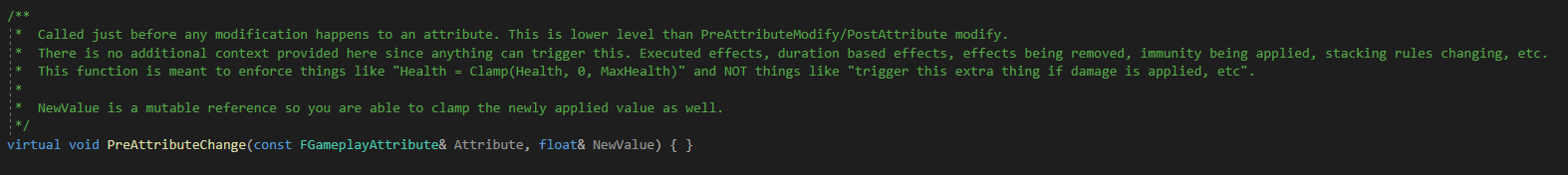
A black and white text

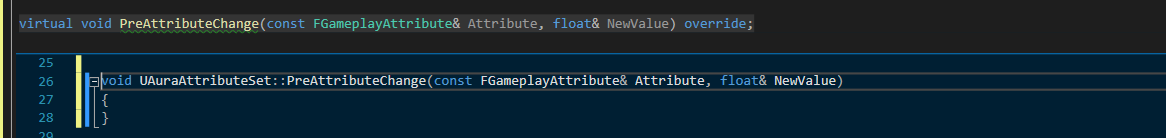
Description automatically generated

# PreAttribute Change

Currently no clamping is being enforced; Mana and Health can go over max and below 0

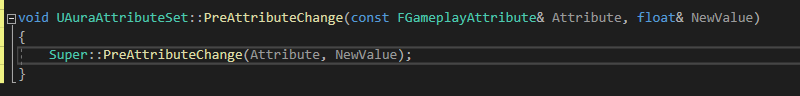
In AuraAttributeSet override one of the inherited functions





Takes in a const reference of FGameplayAttribute (the GameplayAttribute struct) and a float reference NewValue.

Note that NewValue is NOT a const, so it can be changed!



Let’s see what happens in the attribute when we change and attribute.

The cool thing about having an attribute in our argument is we can check that argument and see if it’s equal to any of our current attributes using the accessor functions

A screenshot of a computer

Description automatically generated

Whenever attributes are changed whether from a GE or being set directly this function will kickoff in response. It’s called **Pre**AttributeChange, so it’s started before the change occurs. We can therefore do things pre-emptively (but we shouldn’t be firing events or starting gameplay logic here, it’s recommended for clamping only)

For the sake of education, passing in a UE Log line for the 4 main attributes

A computer screen shot of text

Description automatically generated

The error log shows values as they change. But we can also use Clamp from FMath to restrict changes

A black background with white text

Description automatically generated

A computer screen shot of a program

Description automatically generated

A screenshot of a computer

Description automatically generated

# PostGameplayEffectExecute

Previously we have overridden PreAttributeChange and used it to perform clamping in the attribute set.

PreattributeChange is only really useful for clamping; it’s not great for kicking off logic in response to attribute changes

PostGameplayEffectExecute() is useful for this!

A screen shot of a computer

Description automatically generated

Takes argument Data of type FGameplayEffectModCallbackData const reference

This function is executed after a gameplay effect changes an attribute

The Data parameter allows access to a lot of information, and we’ll be harvesting info from the data parameter!

First we take Data and see **which** attribute has been changed

A screen shot of a computer

Description automatically generated

We ca wrap this in an if and compare:

A screen shot of a computer

Description automatically generated

And in this case log output the current value of the health attribute

We can also get the magnitude of the attribute changed by the gameplay effect:



From the fire:

A screenshot of a computer screen

Description automatically generated

From the health potion:

A screenshot of a computer

Description automatically generated

Nothing from the mana crystal because we’re only getting Health

Health crystal however, yes:

A screenshot of a computer screen

Description automatically generated

With a break at the log for HetHeath Attribute log output:

A screenshot of a computer program

Description automatically generated

In these 3 variables there’s a lot of info; in EffectSpec:

A screenshot of a computer program

Description automatically generated

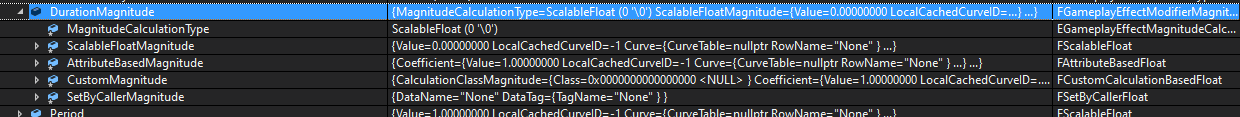
We see the Definition (Def)

A screenshot of a computer

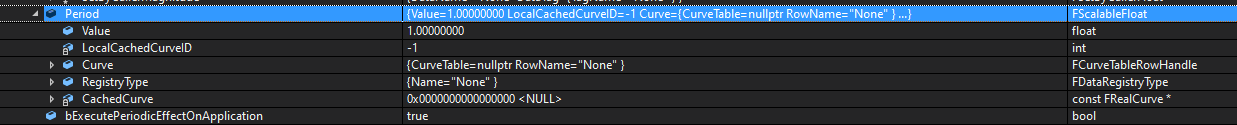
Description automatically generated

The GameplayEffect default object name GE\_FireArea\_C for example

Also visible is the duration magnitude:

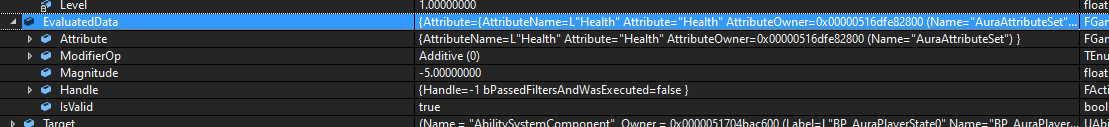


The period and associated info:



We can really get any info we needed from this.

Aside from that we also have Evaluated Data which tells us the attribute, the Modifier operation is additive with magnitude -5,



the Handle is present with it’s FActiveGameplayEffectHandle

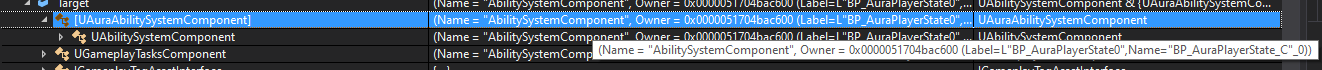


We also have the Target – the AbilitySystemComponent

A screenshot of a computer program

Description automatically generated

And we can see that it’s the specific ASC of the character because it’s owned by the player state



And if we can access the ASC of the target, that means we can access the AbilityActorInfo, both the Owner and Avatar!

So, we can get info about the target and about the source because the effectSpec is a gameplayeffectspec, which has a context, and the context can get info about the source if we set it

Here’s the effect context:

A black screen with white text

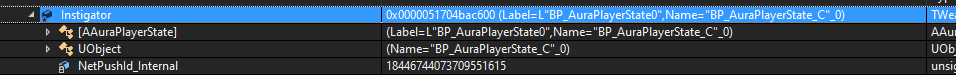
Description automatically generated

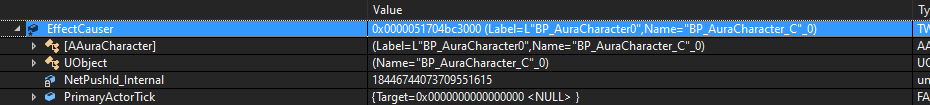
And its Data

A screen shot of a computer

Description automatically generated

We have instigator – in this case the AuraPlayerSTate:



The effect Causer (BPAuraCharacter) 

These are all set in the effect context because the event context was created using the ASC of the overlapped actor in this case

So when you call MakeEffectContext() from an ASC a lot of these data points are set automatically

We just happen to be using the ASC of whatever actor is overlapping our effect actor, so the source is going to be the ASC of that actor, because they made the effect context.

In combat however, when Aura launches a firebolt at an enemy, she’ll be the source and the enemy will be the target. Ultimately in PostGameplayExecute we have access to just about every entity involved in the GameplayEffect being executed, which is a super powerful thing!

So it can be very useful to collect a bunch of data about the gameplay effect and store it. We can create local variables, or even store them all in a structure for this effect execution

This data can therefore be used as a really handy way to respond to gameplay effects being executed!

So, First we get the context:

A black background with white text

Description automatically generated

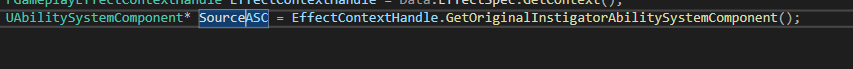
And store in a local variable



Then take that local variable and use it to get the ASC



GetOriginalInstigatorAbilitySystemComponent returns a UAbilitySystemComponent type



Stored in a pointer!

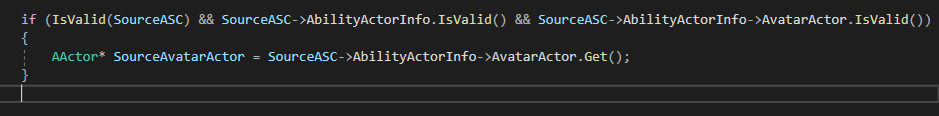
So now we have the source abilitysystemcomponent

We can get lots of things from this, like the owner. Maybe it’s the Avatar Actor?



AvatarActor in this case is a wrapper so we have to call Get() at the end

We’re doing a lot of pointer accessing here any maybe not all actors have a valid ASC, so we should use some checks



So using IsValid for a SourceASC being present, AND it has associated valid ActorInfo, AND it has an avatar set

Itf it’s confusing that we use AbilityActorInfo with the . operator , followed by the -> operator

A screen shot of a computer

Description automatically generated

AbilityActorInfo is a Smart pointer, a TSharedPtr, which has it’s own utilities. AbilityActorInfo.IsValid() is the wrappers’s IsValid() function

After checking the pointer stored in the wrapper is valid, then we use the arrow on it to access it’s Actor and run AvatarActor.IsValid()

Lets get some other objects, eg the player controller



In case the Actor has a null pointer for a player controller we could always fall back on getting the controller from the actor itself by casting to a pawn

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

And GetController() returns an APlayerController so we can just cast

A black screen with white text

Description automatically generated

Or if we want the controller and don’t care if it’s a PC we can rename things a little and cast later if we need it to be a player controller

A screen shot of a computer program

Description automatically generated

And if we cast Source AvatarActor to a pawn we don’t need also to cast the controller (APawn\* Pawn = Cast<APawn>(SourceAvatarActor)

We’re casting here because the source may not be a pawn and may not have a controller, so we’re setting it if we can, if there IS a controller!

We can make some of these consts as well

A screen shot of a computer screen

Description automatically generated

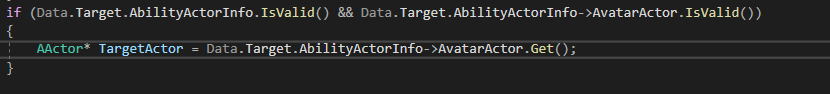
So what we’re doing is we’re first getting the controller from the abilityactorinfo BUT if abilityactorinfo contains a nullptr for the controller then we’ll get it from the actor by casting to a pawn and getting the controller then

We know now we have the SourceAvatarActor, we have the Pawn, so we can also access the source character, if it’s a character. We currently have a source controller that may or may not be null; if it’s not null we can access it’s character and store the character as well

We can get stuff from the Target as well, like to Target AbilitySystemComponent or Avatar



Of course we should run chcks, and we can then get the controller as well



A screen shot of a computer program

Description automatically generated

A screen shot of a computer code

Description automatically generated

There are many possible options.

A screen shot of a computer program

Description automatically generated

The main point is we can get all sorts of things here

But it would be good to refactor and collect items into a struct, rather than clutter this function! Then in postgameplayeffectexecute we can fill in all that data using a function to set the properties based on the effect

A black and gold text on a black background

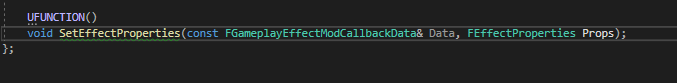
Description automatically generated

Reminder: struct is Name, Gen Body, Default constructor, Declare variables with types.

A screenshot of a computer program

Description automatically generated

## Setting the properties of the struct



Since we want to set these for we should PostGameplayEffectExecute we should use the same argument for that function, and also pass in the struct as a second argument (NON const)

A screenshot of a computer program

Description automatically generated

Then we just move all the code up into the new function and redirect the pointers as Props. References

A computer screen shot of a program

Description automatically generated

Now in the PostGameplayExecute we can access props

A screenshot of a computer program

Description automatically generated