

TL; DR

TODO:

## 0.1 Structure

- **EffectableComponents** are **ActorComponents** that allow for delegation (effects). They have predefined places called “**Outlets**” that allow for code modification. Think of **Outlets** like electrical outlets waiting to be plugged into.
  - Let’s use **StatsComponent** as an example. Say we want a Pokémon-style “Adamant” nature (+10% PhA/−10%SpA). One such place for modification is in the function **RecalculateStats**. **TODO: Update picture!**

```
void UStatsComponent::RecalculateStats(const bool bResetCurrent)
{
    for(FStat* Stat : StatsArray)
    {
        ExecuteBeforeRecalculateStats(Stat, bResetCurrent);
        Stat->Update(GetLevel(), bResetCurrent);
        ExecuteAfterRecalculateStats(Stat, bResetCurrent);
    }
}
```

- **Outlet arrays** are variables inside of **EffectableComponents**. They hold **Outlets** whose delegates execute when needed.
  - **TODO: Update this!** Let’s use **StatsComponent**’s **AfterRecalculateStatsArray** in our example. In this case, after stats are recalculated (say, on level-up), the base PhA would increase by 10% and the base SpA would decrease by 10% (additively):

```

// Define "adamant" delegate (+10% PhA/-10% SpA)
UStatsComponent::FRecalculateStatsDelegate AdamantRecalculateDelegate;
AdamantRecalculateDelegate.BindLambda(InFunctor [StatsComponent](FStat* Stat, bool bResetCurrent) -> void
{
    // +10% PhA
    if ( Stat->Name() == StatsComponent->PhysicalAttack.Name())
    {
        Stat->ModifyValue( Modifier: 10, EStatValueType::Permanent, EModificationMode::AddPercentage);
        if (bResetCurrent)
            Stat->ModifyValue( Modifier: 10, EStatValueType::Current, EModificationMode::AddPercentage);
    }

    // -10% SpA
    if ( Stat->Name() == StatsComponent->SpecialAttack.Name())
    {
        Stat->ModifyValue( Modifier: -10, EStatValueType::Permanent, EModificationMode::AddPercentage);
        if (bResetCurrent)
            Stat->ModifyValue( Modifier: -10, EStatValueType::Current, EModificationMode::AddPercentage);
    }
});
StatsComponent->AfterRecalculateStatsArray.Add(AdamantRecalculateDelegate);

```

- **EffectComponents** are **ActorComponents** that plug into **Outlets**. These come in many forms, but an easy example is a **Buff**. **TODO: Describe how this happens with pictures!**

## 0.2 List of EffectableComponents and Outlets

The following tables show all implemented **EffectableComponents** and their delegate arrays. Note the “base name” indicates existence of:

1. the delegate signature **FBaseNameSignature**;
2. the private before/after arrays of **Outlets**:  
**TArray<FBaseNameOutlet> BeforeBaseName**; and
3. a function for each before/after to execute the arrays: **ExecuteBeforeBaseName (...)**.
4. **AddBeforeBaseName**, a function to add an **Outlet** to the private array **BeforeBaseName** (which also puts it in the right order based on priority).

Note that the philosophy applies to what is *probable* rather than what is *possible*. Hence the list meant to be practical rather than exhaustive.

Table 1: Outlets for TEST

ok	ok
• • •	

Table 2:

Outlets  
for  
UAffinitiesComponent

TODO:  
Todo

Table 3: Outlets for ULevelComponent

GetBaseExpYield	
► Before	<code>const float OriginalYield, float&amp; ReturnedYield</code>
► After	<code>const float OriginalYield, const float ReturnedYield</code>
GetCXP	
► Before	<code>const uint32 OriginalCXP, int32&amp; ReturnedCXP</code>
<i>Note:</i>	ReturnedCXP is <code>int32&amp;</code> instead of <code>uint32&amp;</code> for Blueprint compatability.
► After	<code>const uint32 OriginalCXP const int32 ReturnedCXP</code>
<i>Note:</i>	ReturnedCXP is <code>const int32</code> instead of <code>const uint32</code> for Blueprint compatability.
GetExpYield	
► Before	<code>const float OriginalYield, float&amp; ReturnedYield, const uint16 DefeatedLevel, const uint16 VictoriousLevel</code>

Continued on next page

Table 3: **Outlets** for **ULevelComponent** (Continued)

<i>Note:</i>	“Defeated” and “Victorious” levels are provided for flexibility (e.g., in case you want to yield exp differently based on level difference, although technically you could always back-calculate the level difference based on the equation and <b>OriginalYield</b> ).
► After	<pre>const float OriginalYield, const float ReturnedYield, const uint16 DefeatedLevel, const uint16 VictoriousLevel</pre>
<i>Note:</i>	“Defeated” and “Victorious” levels are provided for symmetry with respect to the <b>Before</b> delegate (since <b>ReturnedValue</b> is already calculated, I can’t think of why you would need them, but you never know!).
<b>GetMaxLevel</b>	
► Before	<pre>const uint16 DefaultMax, int32&amp; AttemptedMax</pre>
<i>Note:</i>	<b>DefaultMax</b> is defined in the code. It should normally be 100, but may change for certain subclasses (e.g., a <b>UBossLevelComponent</b> may have a max of 200 instead). Also, <b>AttemptedMax</b> is <b>int32&amp;</b> instead of <b>uint16&amp;</b> for Blueprint compatability.
► After	<pre>const uint16 DefaultMax const int32 ReturnedMax</pre>
<b>GetMinLevel</b>	
► Before	<pre>const uint16 DefaultMin, int32&amp; AttemptedMin</pre>
<i>Note:</i>	<b>DefaultMin</b> is defined in the code. It should normally be 1, but may change for certain subclasses (e.g., a <b>UEggLevelComponent</b> may have a min of 0 instead for whatever reason). Also, <b>AttemptedMin</b> is <b>int32&amp;</b> instead of <b>uint16&amp;</b> for Blueprint compatability.
► After	<pre>const uint16 DefaultMin const int32 ReturnedMin</pre>
<b>SetBaseExpYield</b>	

Continued on next page

Table 3: Outlets for ULevelComponent (Continued)

► Before	<code>const float OldYield,</code> <code>float&amp; AttemptedYield</code>
► After	<code>const float OldYield</code> <code>const float NewYield</code>
<b>SetCXP</b>	
► Before	<code>const uint32 OldCXP,</code> <code>int32&amp; AttemptedCXP</code>
<i>Note:</i>	<code>AttemptedCXP</code> is <code>int32&amp;</code> instead of <code>uint32&amp;</code> for Blueprint compatability.
► After	<code>const uint32 OldCXP</code> <code>const uint32 NewCXP</code>
<i>Note:</i>	<code>UStatsComponent</code> subscribes to this in order to change stats on level change.

Table 4: Outlets for UStatsComponent

<b>RandomizeStats</b>	
► Before	<code>const EStatEnum TargetStat,</code> <code>const FStatRandParams OriginalParams,</code> <code>FStatRandParams&amp; ParamsToBeUsed</code>
► After	<code>const EStatEnum TargetStat,</code> <code>const FStatRandParams OriginalParams,</code> <code>const FStatRandParams UsedParams</code>
<i>Note:</i>	The <code>EStatEnum</code> is not the acutal <code>FStat</code> . To get the <code>FStat</code> (such as <code>FHealth</code> ), use <code>UStatsComponent::GetStat(EStatEnum)</code> .
<b>RecalculateStats</b>	
► Before	<code>const EStatEnum TargetStat,</code> <code>const bool bResetCurrent,</code> <code>const float OriginalCurrent,</code> <code>const float OriginalPermanent</code>

Continued on next page

Table 4: `Outlets` for `UStatsComponent` (Continued)

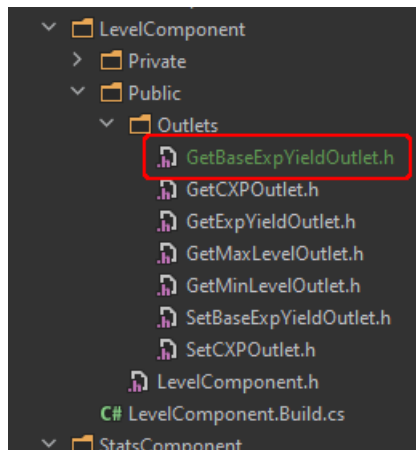
► After	<pre>const EStatEnum TargetStat, const bool bResetCurrent, const float OriginalCurrent, const float OriginalPermanent</pre>
---------	---

Main file is `effects-documentation.tex` and current is `effects-tables.tex`.

### 0.3 Making Your Own Outlet

As an example, let's use `GetBaseExpYield`. (You can imagine that this is an important `Outlet` for tweaking levelling curves.) Here's what to do:

1. **Plan ahead.** I would sincerely recommend you writing down what parameters your `Outlet Before` and `After` delegates take on paper. We go to a few files and it's easy to be inconsistent.
2. **Go to the right directory.** We want to place the `Outlet` inside of `ULevelComponent`, so we'll start with that directory. If yours doesn't contain an "Outlets" directory, create one and place your `Outlet(s)` there.
3. **Copy + paste file.** The easiest way is to copy + paste pre-existing `Outlets`. In this example, we'll copy + paste `SetCXPOutlet.h` and name the new file `GetBaseExpYield.h`.



*Note: this includes both `BeforeGetBaseExpYield` and `AfterGetBaseExpYield` functionality. You don't have to make two different files!*

4. **Replace old name.** Open the new file and you'll still see the base name "SetCXP" everywhere. The easiest way is to do a find+replace "SetCXP" → "GetBaseExpYield".

This replaces everything from the `.generated` include to the delegate signatures. If you're curious, you can look more in-depth and replace instances one-by-one.

5. **Declare delegate signatures.** In this case, we want the `Before` delegate signature to take two arguments: the original, unmodified yield and the one that will be returned from the `GetBaseExpYield` function.

```
DECLARE_DYNAMIC_DELEGATE_TwoParams(FBeforeGetBaseExpYieldSignature,  
    const float, OriginalYield, float&, Yield);
```

You should also set the `After` signature in the same manner. *Note: yours might use more than two parameters or different parameter types. Modify accordingly.*

6. **Declare `Outlet` functions.** In order to be able to call `ExecuteBefore` on your `Outlet`, you need to tell it a few things. The figure below displays a few things in red you should look at:

```
DECLARE_OUTLET_FUNCTIONS_TwoParams(Before, FBeforeGetBaseExpYieldDelegate,  
    BeforeDelegates, Delegate, const float, float&);
```

- Whether it's a `Before` or `After` type `Outlet`. This affects execution based on priority:

## Priorities

The lower the priority, the farther away it is from execution. If two priorities are tied, the older effect is executed first. Order is set externally by `UEffectsComponent` `TODO: fact check this`. Order:

- Intrinsic `Before` delegates (no `UEffect` affiliated)
- `Before` delegates:
  - \* Priority 1
  - \* Priority 2.a (older)
  - \* Priority 2.b (newer)
  - \* ...
- [Function executes]
- `After` delegates:
  - \* ...
  - \* Priority 2.b (newer)
  - \* Priority 2.a (older)
  - \* Priority 1
- Intrinsic `After` delegates (have the final say)

As an example, consider two delegates: one that says you can't take damage no matter what (call the `UBuff` “Invincible”) and another that says damage against you can't be avoided no matter what (call the `UDebuff` “Weakened”). What happens when the target takes damage? Well, it depends on priority:

- They're probably subscribed to the `Before` delegate in `UStatsComponent` called `ModifyStatOutlet` with the target `FStat` being `Health`.
- Note that they're both `Before` delegates.
- Let's say Invincible has Priority 100 and Weakened has Priority 150. The result is the target takes damage because:
  - 1) Invincible first sets the damage to zero.
  - 2) Weakened then sets the damage to no less than its original value.
- If Weakened has lower Priority, the result is flipped and the target takes no damage.



- The parameters you defined in the delegate’s signature. I know, I know—anytime you repeat code, you’re probably doing something wrong. The biggest issue here is the UHT. The main (but not only) issue is that you can’t have **UPROPERTY**s inside macros or the property won’t register. If you have a better way of automating this, *tell me!*
  - Don’t forget the **After** variant’s delegates, which should probably be **const**.
7. **Check number of parameters.** I make a point of this because I find it’s my most common error. Make sure your declared signature *and* declared **Outlet** function macros have the correct number of params (two in our case). Explicitly, you might need to use **DECLARE\_DYNAMIC\_DELEGATE\_FourParams(...)**.
  8. **Declare UPROPERTY.** Inside the **UEffectableComponent** (in this example, **ULevelComponent**), declare the **Outlet** as a variable. Note that it’s custom to have this **UPROPERTY** as public and in the “Outlets” category. It’s also a good idea to comment the **UPROPERTY** with the parameters.

```

FUNCTION(BlueprintCallable, BlueprintPure, Category="Level")
float GetBaseExpYield(); ⑩ 0 blueprint usages

/**
 * Before Parameters:
 * - [const float] original yield prior to modification
 * - [float&] yield that is being set and then returned
 *
 * After Parameters:
 * - [const float] original yield prior to modification
 * - [const float] yield that is being returned
 */
UPROPERTY(VisibleAnywhere, Category="Level Outlets")
FGetBaseExpYieldOutlet GetBaseExpYieldOutlet;

```

*Note:* I use Rider, so it imports **#includes** automatically. Make sure yours does, too.

9. **Implement.** Now it’s time to place your **Outlet** in the appropriate place(s). For our example, it’s pretty simple: place it inside of **GetBaseExpYield** in **ULevelComponent**’s **.cpp** file.

```

float ULevelComponent::GetBaseExpYield()
{
    // Get original for delegates
    const float OriginalBaseExpYield = BaseExpYield;

    // Set up the modifiable return value
    float ReturnedBaseExpYield = BaseExpYield;

    // Call before/after delegates
    GetBaseExpYieldOutlet.ExecuteBefore(OriginalBaseExpYield, [&]ReturnedBaseExpYield);
    GetBaseExpYieldOutlet.ExecuteAfter(OriginalBaseExpYield, ReturnedBaseExpYield);

    // Return for use in other functions
    return ReturnedBaseExpYield;
}

```

Note that you might have to do things like cache original values.

10. **A note on complementary delegates.** If you create a **Before Outlet**, you should also create an **After Outlet**. The biggest difference might be the delegate signature (e.g., reference “&” to **const**).

An example where this would be necessary is an animation delegate. You only want to fire a “bonus exp” animation *after* the amount of exp has been determined, checked, and is now constant.

In some cases, it may not be necessary to have both **Before** and **After** delegates in a function. If you want only one delegate type, or three, or ten, the system is flexible enough to handle it. However, it’s recommended to K.I.S.S.

## 0.4 Making Your Own Effects

Suppose you want to make your own effect from scratch. **TODO: todo**

Main file is effects-documentation.tex and current is effects-documentation.tex.