

Stats System Overview

The Stats System collects and displays performance data to help optimize Unreal Engine projects



PREREQUISITE TOPICS



In order to understand and use the content on this page, make sure you are familiar with the following topics:

- [Stat Commands](#)

The **Stats System** enables you to collect and display performance data so that it can be used to optimize your game.



For help with Stat Commands, type `stat` into the console, or refer to the `PrintStatsHelpToOutputDevice();` method.

Types

The Stats System supports the following types:

Stats Type	Description
Cycle Counter	A generic cycle counter used to count the number of cycles during the object's lifetime.
Float/Dword Counter	A counter that is cleared every frame.
Float/Dword Accumulator	A counter that is not cleared every frame, being a persistent stat that can be reset.
Memory	A special type of counter that is optimized for memory tracking.

Grouping Stats

Each stat must be grouped, which usually corresponds with displaying the specified stat group. For example, **stat statsystem** displays stats' related data.

To define a stat group, use one of the following methods:

Method	Description
<code>DECLARE_STATS_GROUP(GroupDesc, GroupId, GroupCat)</code>	Declares a stats group that is enabled by default.
<code>DECLARE_STATS_GROUP_VERBOSE(GroupDesc, GroupId, GroupCat)</code>	Declares a stats group that is disabled by default.
<code>DECLARE_STATS_GROUP_MAYBE_COMPILED_OUT(GroupDesc, GroupId, GroupCat)</code>	Declares a stats group that is disabled by default and which may be stripped by the compiler.

Where:

- `GroupDesc` is a text description of the group
- `GroupId` is a `UNIQUE` id of the group
- `GroupCat` is reserved for future use
- `CompileIn`, the compiler may strip it out if it is set to true

 Depending on the usage scope, grouping can be done in the source or header file.

Example Usage

```

1 DECLARE_STATS_GROUP(TEXT("Threading"), STATGROUP_Threading, STATCAT_Advanced);
2 DECLARE_STATS_GROUP_VERBOSE(TEXT("Linker Load"), STATGROUP_LinkLoad, STATCAT_Advanced);
3

```

Declaring and Defining Stats

Now, you can declare and define a stat, but before you do, note that a stat can be used in:

- Only one cpp file
- The function scope
- The module scope
- The whole project

For a Single File

For the scope of a single file, you must use one of the following methods, depending on the stat type:

Method	Description
<code>DECLARE_CYCLE_STAT(CounterName, StatId, GroupId)</code>	Declares a cycle counter stat.
<code>DECLARE_SCOPE_CYCLE_COUNTER(CounterName, StatId, GroupId)</code>	Declares a cycle counter stat and uses it at the same time. Additionally, it is limited to one function scope.
<code>QUICK_SCOPE_CYCLE_COUNTER(StatId)</code>	Declares a cycle counter stat that will belong to a stat group named 'Quick'.

Method	Description
<code>RETURN_QUICK_DECLARE_CYCLE_STAT(StatId, GroupId)</code>	Returns a cycle counter, and is sometimes used by a few specialized classes.
<code>DECLARE_FLOAT_COUNTER_STAT(CounterName, StatId, GroupId)</code>	Declares a float counter, and is based on the double type (8 bytes).
<code>DECLARE_DWORD_COUNTER_STAT(CounterName, StatId, GroupId)</code>	Declares a dword counter, and is based on the qword type (8 bytes).
<code>DECLARE_FLOAT_ACCUMULATOR_STAT(CounterName, StatId, GroupId)</code>	Declares a float accumulator.
<code>DECLARE_DWORD_ACCUMULATOR_STAT(CounterName, StatId, GroupId)</code>	Declares a dword accumulator.
<code>DECLARE_MEMORY_STAT(CounterName, StatId, GroupId)</code>	Declares a memory counter the same as a dword accumulator, but it will be displayed with memory specific units.
<code>DECLARE_MEMORY_STAT_POOL(CounterName, StatId, GroupId, Pool)</code>	Declares a memory counter with a pool.

For Multiple Files

If you want to have stats accessible to the whole project (or for a wider range of files) you need to use extern version. These methods are the same as the previously mentioned but with `_EXTERN` and the end of the name:

```
DECLARE_CYCLE_STAT_EXTERN(CounterName, StatId, GroupId, API)
DECLARE_FLOAT_COUNTER_STAT_EXTERN(CounterName, StatId, GroupId, API)
DECLARE_DWORD_COUNTER_STAT_EXTERN(CounterName, StatId, GroupId, API)
DECLARE_FLOAT_ACCUMULATOR_STAT_EXTERN(CounterName, StatId, GroupId, API)
DECLARE_DWORD_ACCUMULATOR_STAT_EXTERN(CounterName, StatId, GroupId, API)
DECLARE_MEMORY_STAT_EXTERN(CounterName, StatId, GroupId, API) DECLARE_MEMORY_STAT_POOL_EXTERN(CounterName, StatId, GroupId, Pool, API)
```

Then in the source file, you need to define these stats with the following, which defines stats that are declared with `_EXTERN`:

Where:

- `CounterName` is a text description of the stat
- `StatId` is a `UNIQUE` id of the stat
- `GroupId` is the id of the group that the stat will belong to, the `GroupId` from `DECLARE_STATS_GROUP*`
- `Pool` is a platform-specific memory pool
- `API` is the `*_API` of a module, and it can be empty if the stat will only be used in that module

Examples


Custom Memory Stats with Pools

First you need to add a new pool to `enum EMemoryCounterRegion`, which can be global or platform specific:

```
1 enum EMemoryCounterRegion
2 {
3     MCR_Invalid, // not memory
4     MCR_Physical, // main system memory
5     MCR_GPU, // memory directly a GPU (graphics card)
6     MCR_GPUSystem, // system memory directly accessible by a GPU
7     MCR_TexturePool, // pre-sized texture pools
8     MCR_MAX
9 };
10
```

 Copy full snippet

The following is an example that will allow using the pools everywhere (see `CORE_API`).

 The pool name must start with `MCR_`.

Header File Sample

```
1 DECLARE_MEMORY_STAT_POOL_EXTERN(TEXT("Physical Memory Pool [Physical]"), MCR_Physical,
   STATGROUP_Memory, FPlatformMemory::MCR_Physical, CORE_API);
2 DECLARE_MEMORY_STAT_POOL_EXTERN(TEXT("GPU Memory Pool [GPU]"), MCR_GPU, STATGROUP_Memory,
   FPlatformMemory::MCR_GPU, CORE_API);
3 DECLARE_MEMORY_STAT_POOL_EXTERN(TEXT("Texture Memory Pool [Texture]"), MCR_TexturePool,
   STATGROUP_Memory, FPlatformMemory::MCR_TexturePool, CORE_API);
4
```

Source File Sample

```
1  DEFINE_STAT(MCR_Physical);
2  DEFINE_STAT(MCR_GPU);
3  DEFINE_STAT(MCR_TexturePool);
4
5  // This is a pool, so it needs to be initialized - typically in F*PlatformMemory::Init().
6  SET_MEMORY_STAT(MCR_Physical, PhysicalPoolLimit);
7  SET_MEMORY_STAT(MCR_GPU, GPUPoolLimit);
8  SET_MEMORY_STAT(MCR_TexturePool, TexturePoolLimit);
9
10 // Now that we have pools, we can set up memory stats for them.
11
12 // The following are accessible everywhere.
13 DECLARE_MEMORY_STAT_POOL_EXTERN(TEXT("Index buffer memory"), STAT_IndexBufferMemory, STATGROUP_RHI,
    FPlatformMemory::MCR_GPU, RHI_API);
14 DECLARE_MEMORY_STAT_POOL_EXTERN(TEXT("Vertex buffer memory"), STAT_VertexBufferMemory, STATGROUP_RHI,
    FPlatformMemory::MCR_GPU, RHI_API);
15 DECLARE_MEMORY_STAT_POOL_EXTERN(TEXT("Structured buffer memory"),
    STAT_StructuredBufferMemory, STATGROUP_RHI, FPlatformMemory::MCR_GPU, RHI_API);
16 DECLARE_MEMORY_STAT_POOL_EXTERN(TEXT("Pixel buffer memory"), STAT_PixelBufferMemory, STATGROUP_RHI,
    FPlatformMemory::MCR_GPU, RHI_API);
17
18 // The following are only accessible in the module where they were defined.
19 DECLARE_MEMORY_STAT_POOL_EXTERN(TEXT("Pool Memory Size"), STAT_TexturePoolSize, STATGROUP_Streaming,
    FPlatformMemory::MCR_TexturePool, );
```



```
20 DECLARE_MEMORY_STAT_POOL_EXTERN(TEXT("Pool Memory Used"), STAT_TexturePoolAllocatedSize,  
    STATGROUP_Streaming, FPlatformMemory::MCR_TexturePool, );  
21  
22 // And the last thing, we need to update the memory stats.  
23  
24 // Increases the memory stat by the specified value.  
25 INC_MEMORY_STAT_BY(STAT_PixelBufferMemory, NumBytes);  
26 // Decreases the memory stat by the specified value.  
27 DEC_MEMORY_STAT_BY(STAT_PixelBufferMemory, NumBytes);  
28 // Sets the memory stat by the specified value.  
29 SET_MEMORY_STAT(STAT_PixelBufferMemory, NumBytes);  
30
```

 Copy full snippet

Regular Memory Stats Without Pools

```
1 DECLARE_MEMORY_STAT(TEXT("Total Physical"), STAT_TotalPhysical, STATGROUP_MemoryPlatform);  
2 DECLARE_MEMORY_STAT(TEXT("Total Virtual"), STAT_TotalVirtual, STATGROUP_MemoryPlatform);  
3 DECLARE_MEMORY_STAT(TEXT("Page Size"), STAT_PageSize, STATGROUP_MemoryPlatform);  
4 DECLARE_MEMORY_STAT(TEXT("Total Physical GB"), STAT_TotalPhysicalGB, STATGROUP_MemoryPlatform);  
5
```

 Copy full snippet

Or, if you prefer, you can `DECLARE_MEMORY_STAT_EXTERN` in the header file and then `DEFINE_STAT` in the source file.




Updating the memory stats is done the same way as in the version with pools.

Performance Data Using Cycle Counters

First, you need to add cycle counters:


```
1 DECLARE_CYCLE_STAT(TEXT("Broadcast"), STAT_StatsBroadcast, STATGROUP_StatSystem);
2 DECLARE_CYCLE_STAT(TEXT("Condense"), STAT_StatsCondense, STATGROUP_StatSystem);
3
```

 Copy full snippet

Or, you can `DECLARE_CYCLE_STAT_EXTERN` in the header file and then `DEFINE_STAT` in the source file.

Now, you can grab the performance data:

```
1 Stats::Broadcast()
2 {
3     SCOPE_CYCLE_COUNTER(STAT_StatsBroadcast);
4     // ...
5     // a piece of code
6     // ...
7 }
8
```

 Copy full snippet

Sometimes, you do not want to grab the stats every time the function is called, so you can use a conditional cycle counter — it is not very common, but it may be useful:

```
1 Stats::Broadcast(bool bSomeCondition)
2 {
3
4  CONDITIONAL_SCOPE_CYCLE_COUNTER(STAT_StatsBroadcast,bSomeCondition);
5  // ...
6  // a piece of code
7  // ...
8  }
9
```

 Copy full snippet

If you want to grab the performance data from one function, you can use the following construction:

```
1 Stats::Broadcast(bool bSomeCondition)
2 {
3  DECLARE_SCOPE_CYCLE_COUNTER(TEXT("Broadcast"), STAT_StatsBroadcast, STATGROUP_StatSystem);
4  // ...
5  // a piece of code
6  // ...
7  }
8
```

 Copy full snippet

You can also do the following:

```
1 Stats::Broadcast(bool bSomeCondition)
2 {
3     QUICK_SCOPE_CYCLE_COUNTER(TEXT("Stats::Broadcast"));
4     // ...
5     // a piece of code
6     // ...
7 }
8
```

 Copy full snippet



This is mostly used for temporary stats.

All of the previously mentioned cycle counters are used to generate the hierarchy so that you can get more detailed information about performance data. However, there is also the option of setting a flat cycle counter:

```
1 Stats::Broadcast(bool bSomeCondition)
2 {
3     const uint32 BroadcastBeginTime = FPlatformTime::Cycles();
4     // ...
5     // a piece of code
6     // ...
7     const uint32 BroadcastEndTime = FPlatformTime::Cycles();
8     SET_CYCLE_COUNTER(STAT_StatsBroadcast, BroadcastEndTime-BroadcastBeginTime);
9 }
10
```

 Copy full snippet

Performance Data Using GetStatId

A few tasks implemented in Unreal Engine use a different approach in terms of getting the performance data. They implement method `GetStatId()`, and if there is no `GetStatId()`, the code will not compile.

Here is an example:

```
1 class FParallelAnimationCompletionTask
2 {
3 // ...
4 // a piece of code
5 // ...
6 FORCEINLINE TStatId GetStatId() const
7 {
8 RETURN_QUICK_DECLARE_CYCLE_STAT(FParallelAnimationCompletionTask, STATGROUP_TaskGraphTasks);
9 }
10 // ...
11 // a piece of code
12 // ...
13 };
14
```

 Copy full snippet

Logging Performance Data

If you just want to log performance data, we provide the following functionality:


Method

The following method captures the time (passed in seconds), and adds delta time to the variable that is passed in:

```
SCOPE_SECONDS_COUNTER(double& Seconds)
```

Code Sample

```
1 Stats::Broadcast()  
2 {  
3     double ThisTime = 0;  
4     {  
5         SCOPE_SECONDS_COUNTER(ThisTime);  
6         // ...  
7         // a piece of code  
8         // ...  
9     }  
10    UE_LOG(LogTemp, Log, TEXT("Stats::Broadcast %.2f"), ThisTime);  
11 }  
12
```

 Copy full snippet

Utility Class and Methods

Class	Description
<code>FScopeLogTime</code>	Utility class that logs the time (passed in seconds), adding cumulative stats to passed-in variable, and printing the performance data to the log in the destructor.
Method	Description
<code>SCOPE_LOG_TIME(Name, CumulativePtr)</code>	Using the provided name, prints the performance data and gathers cumulative stats.
<code>SCOPE_LOG_TIME_IN_SECONDS(Name, CumulativePtr)</code>	The same functionality as <code>SCOPE_LOG_TIME</code> , but it prints in seconds.
<code>SCOPE_LOG_TIME_FUNC()</code>	Using the function name, it prints the performance data, and it cannot be nested.
<code>SCOPE_LOG_TIME_FUNC_WITH_GLOBAL(CumulativePtr)</code>	The same functionality as <code>SCOPE_LOG_TIME_FUNC</code> , but it gathers cumulative stats.

Code Sample

```

1 double GMyBroadcastTime = 0.0;
2 Stats::Broadcast()
3 {
4     SCOPE_LOG_TIME("Stats::Broadcast", &GMyBroadcastTime);

```

```
5  SCOPE_LOG_TIME_IN_SECONDS("Stats::Broadcast (sec)", &GMyBroadcastTime);
6  // ...
7  // a piece of code
8  // ...
9  }
10
11 Stats::Condense()
12 {
13  SCOPE_LOG_TIME_FUNC(); // The name should be "Stats::Condense()", may differ across compilers
14  SCOPE_LOG_TIME_FUNC_WITH_GLOBAL(&GMyBroadcastTime);
15  // ...
16  // a piece of code
17  // ...
18  }
19
```

 Copy full snippet

Generic Data Using FLOAT or DWORD Counters

Generally, the first thing you need to do is add your counters, such as the following:

```
1  DECLARE_FLOAT_COUNTER_STAT_EXTERN(STAT_FloatCounter, StatId, STATGROUP_TestGroup, CORE_API);
2  DECLARE_DWORD_COUNTER_STAT_EXTERN(STAT_DwordCounter, StatId, STATGROUP_TestGroup, CORE_API);
3  DECLARE_FLOAT_ACCUMULATOR_STAT_EXTERN(STAT_FloatAccumulator, StatId, STATGROUP_TestGroup, CORE_API);
4  DECLARE_DWORD_ACCUMULATOR_STAT_EXTERN(STAT_DwordAccumulator, StatId, STATGROUP_TestGroup, CORE_API);
5
```


 Copy full snippet

Then, to update and manage your counters, you can use the following methods:

Methods to Update Counters

Method	Description
<code>INC_DWORD_STAT(StatId)</code>	Increases a dword stat by 1.
<code>DEC_DWORD_STAT(StatId)</code>	Decreases a dword stat by 1.
<code>INC_DWORD_STAT_BY(StatId, Amount)</code>	Increases a dword stat by the specified value.
<code>DEC_DWORD_STAT_BY(StatId, Amount)</code>	Decreases a dword stat by the specified value.
<code>SET_DWORD_STAT(StatId, Value)</code>	Sets a dword stat to the specified value.
<code>INC_FLOAT_STAT_BY(StatId, Amount)</code>	Increases a float stat by the specified value.
<code>DEC_FLOAT_STAT_BY(StatId, Amount)</code>	Decreases a float stat by the specified value.
<code>SET_FLOAT_STAT(StatId, Value)</code>	Sets a float stat to the specified value.

Helper Methods to Manage Counters

Method	Description
<code>GET_STATID(StatId)</code>	<p>Returns an instance of the <code>TStatId</code> of the stat.</p> <p> This is advanced.</p>
<code>GET_STATDESCRIPTION(StatId)</code>	<p>Returns a description of the stat.</p>