

Tasks System Reference

A reference to different tasks available in Tasks System.



Tasks System resides in the `UE::Tasks` namespace. To use Tasks System you will need to include the `Tasks/Task.h` library. You can refer to the `Tests/Tasks/TasksTest.cpp` class for usage examples. The reference table below provides a few examples to the key functions of the Tasks System.

Reference	Description
<code>TTask<ResultType>()</code>	<p>A handle of an actual task. It uses reference counting to manage the task's lifetime.</p> <ul style="list-style-type: none">• The task is created when it's launched.• Releasing the last user-held reference doesn't necessarily release the task as the system can still hold an internal reference used to execute the task.• <code>FTask</code> is an alias for tasks that do not return results (<code>TTask</code>);
<code>TTask<ResultType>::IsValid()</code>	<p>The function:</p>

1	<code>bool TTask<ResultType>::IsValid() const;</code>
2	

 Copy full snippet

Returns true if the task handle references a task. A default-constructed task handle is "empty" and so is "not valid". A task is constructed when it's launched. For example:

1	<code>FTask Task;</code>
2	
3	<code>check(!Task.IsValid());</code>
4	
5	<code>Task = Launch(UE_SOURCE_LOCATION, [] {});</code>
6	
7	<code>check(Task.IsValid());</code>
8	
9	<code>Task = {}; <i>// reset the task object</i></code>
1	
0	

Reference

Description

```
1      check(!Task.IsValid());
1
```

 Copy full snippet

Task<ResultType>::Launch

Launches a task for asynchronous execution. In the Code example below launch is used on a task and returns its handle:

```
1      template<typename TaskBodyType>
2
3      TTask<TInvokeResult_T<TaskBodyType>>
4          Launch(
5              const TCHAR* DebugName,
6              TaskBodyType&& TaskBody,
7              LowLevelTasks::ETaskPriority
8              Priority =
9              LowLevelTasks::ETaskPriority::Normal
10             );
```

 Copy full snippet

Prerequisites are other tasks that must be completed before the task that depends on them is executed. Once all

prerequisites are completed, the task is automatically scheduled for execution.

1	<code>template<typename TaskBodyType,</code>
	<code>typename PrerequisitesCollectionType></code>
2	<code>TTask<TInvokeResult_T<TaskBodyType>></code>
	<code>TTask<ResultType>::Launch(</code>
3	<code>const TCHAR* DebugName,</code>
4	<code>TaskBodyType&& TaskBody,</code>
5	<code>PrerequisitesCollectionType&&</code>
	<code>Prerequisites,</code>
6	<code>LowLevelTasks::ETaskPriority</code>
	<code>Priority =</code>
7	<code>LowLevelTasks::ETaskPriority::Normal</code>
8	<code>);</code>
9	

 Copy full snippet

Parameters:

- DebugName - a (preferably) unique name for task identification in debugger and profiler.
UE_SOURCE_LOCATION macro can be used, that generates a string [Filename]:[Lineno] of the source location where it's used.

- TaskBody - a callable object that will be executed asynchronously, e.g. a lambda, a function pointer or a class with operator();
- Prerequisites - an iterable collection of TTask. Their result type is not required to match the result type of the task.
- Priority - task priority that affects the order in which tasks are executed.

Example:

```
1      FTask Prerequisite1 =  
      Launch(UE_SOURCE_LOCATION, []{});  
2  
3      FTask Prerequisite2 =  
      Launch(UE_SOURCE_LOCATION, []{},  
      ETaskPriority::High);  
4  
5      FTask DependentTask =  
      Launch(UE_SOURCE_LOCATION, []{},  
      Prerequisites(Prerequisite1,  
      Prerequisite2));  
6  
7      TTask<bool> BoolTask =  
      Launch(UE_SOURCE_LOCATION, []{ return  
      true; });
```

 Copy full snippet

Reference

Description

`template<typename... TaskTypes>`
`TPrerequisites<TaskTypes...>`
`Prerequisites(TaskTypes&... Tasks);` is a helper function to pass a variable number of prerequisites to `Launch()` and `FTaskEvent::AddPrerequisites()`. For additional examples, you can observe the tasks: `IsCompleted()`, `Wait()`, `GetResult()`.

`TTask<ResultType>::IsCompleted`

Returns true if the task is completed or is not valid.

```
1      bool
      TTask<ResultType>::IsCompleted()
      const;
```

 Copy full snippet

A task is completed **if** its execution is finished and all its nested tasks are completed.

Example:

```
1      FTask Task;
```

```
2
```

```
3      check(Task.IsCompleted());
```

```
4
```

```
5      Task = Launch(UE_SOURCE_LOCATION,
                    []{});
```

Reference

Description

6	
7	<code>Task.Wait();</code>
8	
9	<code>check(Task.IsCompleted());</code>
1	
0	

 Copy full snippet

For additional examples, you can observe the tasks:

`Launch()`, `Wait()` and `GetResult()`.

`TTask<ResultType>::Wait`

Blocks the current thread until the task(s) is completed or waiting timed out. Returns false on timeout. Waiting can take longer than the specified timeout value. The task(s) is completed if `Wait()` returns true. Returns true immediately if the task is not valid:

1	<code>bool</code> <code>TTask<ResultType>::Wait(FTimespan</code> <code>Timeout);</code>
2	<code>template<typename</code> <code>TaskCollectionType></code>
3	
4	<code>bool Wait(const TaskCollectionType&</code> <code>Tasks, FTimespan InTimeout);</code>
5	

 Copy full snippet**Example:**

1	<code>FTask Task;</code>
2	<code>Task.Wait(); // returns immediately</code>
3	<code>Task = Launch(UE_SOURCE_LOCATION, []{});</code>
4	<code>Task.Wait(FTimespan::FromMillisecond(3)); // blocks until the task is completed or waiting timed out</code>
5	
6	<code>FTask AnotherTask = Launch(UE_SOURCE_LOCATION, []{});</code>
7	<code>TArray<FTask> Tasks{ Task, AnotherTask };</code>
8	<code>Wait(Tasks); // blocks until all tasks are completed</code>
9	

 Copy full snippet

If the task execution is not started yet (it's blocked by a prerequisite, or wasn't picked up by a worker thread yet), waiting for it will "retract" the task and execute it locally (inline). As the task execution hasn't started yet, the waiting thread would need to be blocked while a worker thread

executes the task. Executing the task by the waiting thread is not slow process but can be faster, and doesn't occupy a worker thread. Task retraction follows task dependencies, so-called "deep task retraction". If task execution is blocked by prerequisites, task retraction will try to unblock the task by retracting and executing its prerequisites, recursively. If task retraction fails for any reason(the task execution already started) it falls back to blocking waiting.

Example:

```
1      FTask Task1 =
      Launch(UE_SOURCE_LOCATION, []{});

2      FTask Task2 =
      Launch(UE_SOURCE_LOCATION, []{});

3      FTask Task3 =
      Launch(UE_SOURCE_LOCATION, []{},
      Task2);

4      Task3.Wait();

5
```

 Copy full snippet

The sample above launches three tasks, where `Task2` is a prerequisite of `Task3`. Waiting for `Task3` completion may retract `Task3` and/or its prerequisite `Task2` and execute them online, however this will not apply to `Task1`.

`TTask<ResultType>::BusyWait`

Busy waiting for a task is the execution of other unrelated tasks while waiting for the task's completion. This can improve the system throughput but should be used cautiously. Busy waiting can take longer than blocking waiting and could affect latency-sensitive task-chains. In the function below, the task executes other ready for

execution tasks until the task that is waited for is completed. Next, the task is completed after BusyWait Returns.

1	<code>void TTask<ResultType>::BusyWait();</code>
2	

 Copy full snippet

In the code sample below, we execute other ready for execution tasks until the task(s) that is waited for is completed, or waiting timed out. Next, we Return false on timeout. Waiting can take longer than the specified timeout value. The task(s) is completed if BusyWait returns true.

1	<code>bool TTask<ResultType>::BusyWait(FTimespan Timeout);</code>
2	
3	<code>template<typename TaskCollectionType></code>
4	<code>bool BusyWait(const TaskCollectionType& Tasks,</code>
5	<code>FTimespan InTimeout = FTimespan::MaxValue())</code>
6	
7	

 Copy full snippet

Before executing unrelated tasks, busy waiting first tries to retract the task that is waited for.

1	FTask Task;
2	Task.BusyWait(); <i>// returns immediately</i>
3	
4	Task = Launch(UE_SOURCE_LOCATION, []{});
5	Task.BusyWait(); <i>// blocks until the task is completed, can execute other tasks while blocked</i>
6	
7	FTask AnotherTask = Launch(UE_SOURCE_LOCATION, []{});
8	TArray<FTask> Tasks{ Task, AnotherTask };
9	BusyWait(Tasks, FTimespan::FromMilliseconds(1));
10	<i>// preceding line blocks until all tasks are completed or waiting timed out, can execute other tasks while blocked</i>

1

1

 Copy full snippet

TTask<ResultType>::GetResult

Returns a reference to an object returned by the task as a result of its execution (the value returned by task body execution).

```
ResultType& TTask<ResultType>::GetResult();
```

 Copy full snippet



This exists only for tasks with non-void ResultType.

If the task is completed, the call returns immediately. Otherwise, it blocks until the task is completed. The result object gets destroyed when the task object is destroyed, which is when the last reference to the task object is released. The call asserts if the task is not valid.

Example:

```
1      TTask<bool> BoolTask =
      Launch(UE_SOURCE_LOCATION, []{ return
      true; });

2      bool bResult =
      BoolTask.GetResult();

3

4      TTask<int32> IntTask;

5      // IntTask.GetResult(); - asserts,
      the task is invalid as it wasn't
```

Reference

Description

launched

6

Copy full snippet

AddNested()

Registers the given task as "nested" to the "current" task (parent task). The **current task** is a task that is being executed by the current thread.

The **Parent task** is not completed until all nested tasks are completed.


Asserts if is called not from inside another task.

```
1      template<typename TaskType>
2      void AddNested(const TaskType&
                    Nested);
```

Copy full snippet


Example:

```
1      FTask ParentTask =
        Launch(TEXT("Parent Task"),
2
3          []
4          {
            FTask NestedTask =
                Launch(TEXT("Nested Task"), []{});
```

Reference	Description
	<div><div>5AddNested(NestedTask);</div><div>6}</div><div>7);</div><div>8</div></div> <div> Copy full snippet</div>

FTaskEvent

FTaskEvent shares a part of its API with `TTask<ResultType>`. For example, `IsValid()`, `IsCompleted()`, waiting and busy-waiting API is the same. This section describes only FTaskEvent specific API.

Reference Task Event	Description
<div>FTaskEvent Constructor</div>	<p>Creates a task event object with the given debug name. In contrast with TTask, task events are "valid" after construction <code>IsValid() == true</code>, and incomplete <code>IsCompleted() == false</code>. Debug name is used to identify a task event object for debugging purposes.</p> <div><div>1explicit FTaskEvent::FTaskEvent(const TCHAR* DebugName);</div><div>2</div></div> <div> Copy full snippet</div> <p>A task event must be triggered before it's destroyed.</p>

Example:

```
1      FTaskEvent TaskEvent{
      UE_SOURCE_LOCATION };

2      check(TaskEvent.IsValid());

3      check(!TaskEvent.IsCompleted());

4      TaskEvent.Trigger();

5      check(TaskEvent.IsCompleted());
```

 Copy full snippet**FTaskEvent::AddPrerequisites**

Adds other tasks (or task events) as prerequisites. Can be called only before triggering the task event. Only when all prerequisites are completed and the task event is triggered does it become "completed" ("signaling").



```
1      template<typename PrerequisitesType>

2      void
      FTaskEvent::AddPrerequisites(const
      PrerequisitesType& Prerequisites);

3
```


 Copy full snippet**Example:**



1	FTaskEvent TaskEvent{ TEXT("TaskEvent") };
2	
3	TArray<FTask> Prereqs
4	{
5	Launch(TEXT("Task A"), [] {}),
6	Launch(TEXT("Task B"), [] {})
7	};
8	TaskEvent.AddPrerequisites(Prereqs);
9	
1 0	FTask TaskC = Launch(TEXT("Task C"), [] {});
1 1	FTask TaskD = Launch(TEXT("Task D"), [] {});
1 2	TaskEvent.AddPrerequisites(Prerequisites(TaskC, TaskD));
1 3	
1 4	TaskEvent.Trigger();

Reference Task Event	Description
	<div>1</div> <div>5</div> <div>  Copy full snippet </div>
<div>FTaskEvent::Trigger</div>	<p>A task event is incomplete ("non-signaling") until it's triggered. Triggering a task event doesn't necessarily make it signaling, only when all its prerequisites are completed and the task event is triggered does it become completed.</p> <p>Every task event must be triggered. Otherwise, its destructor will assert that the task event is not completed.</p> <div> void FTaskEvent::Trigger(); </div> <div>  Copy full snippet </div>

FPipe

Pipes are lightweight non-copyable and non-movable objects. The construction of a pipe doesn't allocate dynamic memory and doesn't perform costly processing.


Reference Name	Description
<div>FPipe constructor</div>	<p>Constructs a pipe object with the given debug name. Debug name is used for debugging purposes, to identify a pipe object.</p> <div> <div>1</div> <div>FPipe::FPipe(const TCHAR* DebugName);</div> <div>2</div> </div> <div>  Copy full snippet </div>

Reference Name	Description
	Pipes are lightweight non-copyable and non-movable objects. The construction of a pipe doesn't allocate dynamic memory and doesn't do any costly processing
<code>FPipe</code> destructor	Checks if the pipe has any non completed tasks. A pipe must not have any non completed tasks upon destruction.
<code>HasWork()</code>	<p>Checks if the pipe has any non completed tasks. A pipe must not have any non-completed tasks upon destruction.</p> <div> <div> <pre> 1 bool FPipe::HasWork() const; 2 </pre> </div> <div></div> </div> <p> Copy full snippet</p>
<code>WaitUntilEmpty()</code>	<p>This call will block until all pipe's tasks are completed.</p> <div> <div> <pre> 1 void FPipe::WaitUntilEmpty(); 2 </pre> </div> <div></div> </div> <p> Copy full snippet</p> <p>Refer to the function <code>HasWork()</code> for an additional example.</p>
<code>Launch()</code>	<p>Launches a task in the pipe. Tasks launched in the same pipe are executed not concurrently (one after another), but can be executed by different worker threads.</p> <div> <div> <pre> 1 template<typename TaskBodyType> </pre> </div> <div></div> </div>

Reference Name

Description

	2	TTask<TInvokeResult_T<TaskBodyType>> FPipe::Launch(
	3	const TCHAR* InDebugName,
	4	TaskBodyType&& TaskBody,
	5	LowLevelTasks::ETaskPriority Priority = LowLevelTasks::ETaskPriority::Default
	6);
	7	
	8	template<typename TaskBodyType, typename PrerequisitesCollectionType>
	9	TTask<TInvokeResult_T<TaskBodyType>> FPipe::Launch(
	1 0	const TCHAR* InDebugName,
	1 1	TaskBodyType&& TaskBody,
	1 2	PrerequisitesCollectionType&& Prerequisites,
	1 3	LowLevelTasks::ETaskPriority Priority = LowLevelTasks::ETaskPriority::Default
	1 4);

Reference Name	Description
<code>IsInContext()</code>	<p>Returns true if called from inside a task that belongs to this pipe. Can be used to check if it's safe to access a shared resource protected by a pipe, such as when the code doesn't have scope of being executed by a piped task.</p> <pre>bool FPipe::IsInContext() const;</pre> <p> Copy full snippet</p>