Build and Run Low-Level Tests

Learn about the different ways to build and run Low-Level Tests.

PREREQUISITE TOPICS

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

- Low-Level Tests
- Types of Low-Level Tests
- Write Low-Level Tests

You can build and run low-level tests with:

- Visual Studio
- Unreal Build Tool (UBT)
- BuildGraph

You can build and run explicit tests using any of these tools. We recommend you build and run explicit tests with BuildGraph whenever possible.

Test Type Availability	Visual Studio	Unreal Build Tool	BuildGraph
Explicit Tests	Yes	Yes	Yes

At the end of this page, the Example: Foundation Tests section guides you through how to build and run a low-level tests project included with Unreal Engine.

Visual Studio

You can build and run explicit tests directly from Visual Studio on desktop platforms:

1. Install UnrealVS.
    - This is optional, but strongly recommended, as it enhances test discoverability. For more information about UnrealVS, see the UnrealVS Extension documentation.
2. **Build** the test projects from Visual Studio to produce the executables.
    - The Visual Studio built-in test adapter discovers tests in the Catch2 executables. You can build with UnrealVS or directly through Visual Studio's interface.
3. The tests are displayed in the **Test Explorer**. Select **Test > Test Explorer** from the menu. From here, you can run tests and navigate to their source code.
4. If there are no tests in Test Explorer, it's likely that the build at step 2 didn't update the executable. Run **Rebuild** on the test project to remedy this problem.

Unreal Build Tool

Build

Explicit Tests

You can use Unreal Build Tool to build explicit tests. Suppose we build explicit test cases with their target class `MyTestsTarget`:

```
.\Engine\Binaries\DotNET\UnrealBuildTool\UnrealBuildTool.exe MyTestsTarget Development Win64
```

Copy full snippet

The configuration used above is `Development` and the platform is `Win64` for example purposes. All configurations and platforms are supported.

Run

The previous UBT commands build a test executable. The test executable is located in the same base folder that the target normally outputs, such as `Binaries/<PLATFORM>`, but under a folder with the same name as the target. Here is an example that runs a low-level tests executable from the command-line after building them in the manner above with Unreal Build Tool:

```
MyTests.exe --log --debug --sleep=5 --timeout=10 -r xml -# [#MyTestFile][Core] --extra-args -stdout
```

Copy full snippet

This command-line does the following:

LLT arguments:

```
--log --debug --sleep=5 --timeout=10
```

Copy full snippet

- Enable UE logging.
- Print low-level tests debug messages (test start, finish, completion time).
- Wait 5 seconds before running tests.
- Set a per-test timeout of 10 minutes.

Catch2 arguments:

```
-r xml -# [#MyTestFile][Core]
```

Copy full snippet

- Enable XML reporting.
- Use filenames as filter tags and select all tests from the file `MyTestFile` that are tagged `[Core]`.

UE arguments:

```
--extra-args -stdout
```

Copy full snippet

- Set `-stdout` to the UE command-line.

Command-Line Reference

Once built, you can use a test executable for running pre-submit tests or as part of a Continuous Integration/Continuous Delivery (CI/CD) pipeline. The LLT executable supports a range of command line options that cover many use cases.

Argument	Flag or Key-Value Pair	Description
`--global-setup`	Flag	Run global setup that initializes UE core components.
`--no-global-setup`	Flag	Use this to disable global setup.
`--log`	Flag	Enabled UE log output.
`--no-log`	Flag	Disabled UE log output.
`--debug`	Flag	Enable `LowLevelTestsRunner` logger debug messages for current test execution.
`--mt`	Flag	Set `bMultiThreaded=true`. Use this to configure a multithreaded environment.
`--no-mt`	Flag	Set `bMultiThreaded=false`. Use this to configure a single-threaded environment.
`--wait`	Flag	Wait for user input before exiting.
`--no-wait`	Flag	Do not wait for user input before exiting. This is the default behavior.
`--attach-to-debugger`, `--waitfordebugger`	Flag	Application waits for debugger to attach before the global setup phase.

Argument	Flag or Key-Value Pair	Description
`--buildmachine`	Flag	Set the UE global variable `bIsBuildMachine=true`. Used for development to control CI/CD behavior.
`--sleep=<SECONDS>`	Key-Value Pair	Set a sleep period in seconds before the global setup phase. Useful for cases where synchronization demands tests to wait before startup.
`--timeout=<MINUTES>`	Key-Value Pair	Set a per-test timeout in minutes. When the timeout is reached during a single test case, an error message is printed.
`--reporter=` etc. `-r` etc.	Both	Catch2 command-line options. Any command line option that is not one from the above, and it's not after --extra-args, is automatically sent to Catch2. For a full reference of the Catch2 command-line options, see the external Catch2 Command-Line documentation in the Catch2 GitHub repository.
`--extra-args`	Flag	All arguments set after this option are set on UE's `FCommandLine`. Useful in cases where features are enabled from the command-line.

As described in the last two entries above, any argument that is not enumerated in the reference, up to `--extra-args` included, is sent directly to the Catch2 command line argument list.

BuildGraph

The recommended way to build and run tests is through the BuildGraph script. A basic command looks like this:

```
.\RunUAT.bat BuildGraph -Script="Engine/Build/LowLevelTests.xml" -Target="Run Low Level Tests"
```

Copy full snippet

Windows, Mac and Linux can use similar commands.

The script located at `Engine/Build/LowLevelTests.xml` works with test metadata files. To generate metadata files, refer to the Generate BuildGraph Script Metadata files section of the Types of Low-Level Tests documentation. Correct execution of this script is conditional on successfully generated test metadata scripts, so be sure to confirm that files are produced in their expected locations. All the test metadata `.xml` files are included in `LowLevelTests.xml` and this metadata drives the execution of the nodes in the build graph.

Here are some common ways to use the BuildGraph script:

1. Run a test with name `MyTest` on Windows:

```
    .\RunUAT.bat BuildGraph -Script="Engine/Build/LowLevelTests.xml" -Target="MyTest Tests Win64"
```

Copy full snippet

2. Set a specific build configuration other than the default, which is Development. For example, you can set the configuration to **Debug** like this:

```
.\RunUAT.bat BuildGraph -Script="Engine/Build/LowLevelTests.xml" -Target="Foundation Tests Win64" -set:Configuration="Debug"
```

Copy full snippet

3. Build and launch a test in Debug configuration and wait for debugger to attach to it:

```
.\RunUAT.bat BuildGraph -Script="Engine/Build/LowLevelTests.xml" -Target="Foundation Tests Win64" -set:Configuration="Debug" -set:A
```

Copy full snippet

4. If a platform's tooling supports deploying an application onto a device of a given name or IP, you can launch it onto that device. This command can also be used together with `AttachToDebugger` as well if the platform has remote debugging tools:

```
.\RunUAT.bat BuildGraph -Script="Engine/Build/LowLevelTests.xml" -Target="Foundation Tests Win64" -set:Device="<IP_OR_NAME_OF_DEVI
```

Copy full snippet

5. Build Catch2 for a target platform:

```
.\RunUAT.bat BuildGraph -Script="Engine/Build/LowLevelTests.xml" -Target="Catch2 Build Library Win64"
```

Copy full snippet

Example: Foundation Tests

Overview

The Foundation Tests project is located in the Visual Studio **Solution Explorer** under the `Programs/LowLevelTests` folder.

There are lifecycle events defined in `Tests/TestGroupEvents.cpp`. Be mindful of the order of execution of these events, as they impact correct execution of tests. Lack of setup, or incorrectly placed setup, can cause runtime errors, the same goes for teardown events.

Build and Run

Build and run tests from Visual Studio or use BuildGraph.

Visual Studio

To build Foundation Tests from Visual Studio, follow these steps:

1. Ensure that you have installed **UnrealVS** as it makes building Tests easier.
    - See the UnrealVS Extension documentation for more information.
2. Navigate to the Visual Studio menu and find **Solution Configurations**.
3. From the **Solution Configurations** dropdown, select **Tests**.
4. In the UnrealVS toolbar, find the **Startup Project** dropdown, and select **FoundationTests**.
5. In the Visual Studio menu bar, select **Build > Build Solution**.

This builds Foundation Tests and its dependencies. To run Foundation Tests, navigate to the `Engine/Binaries/Win64/FoundationTests` directory from your terminal or command prompt and run the `FoundationTests.exe` executable with `./FoundationTests`.

If everything works correctly, you will see some Log text in your terminal window and, if all tests pass, a dialog at the end that reads "All tests passed...".

BuildGraph

To build and run Foundation Tests, navigate to your project directory and run the command:

```
.\RunUAT BuildGraph -Script="Engine/Build/LowLevelTests.xml" -Target="Foundation Tests Win64"
```

Copy full snippet

You can specify different platforms, build configuration, device target to run tests on or make the tests wait for a debugger to attach.