Types of Low-Level Tests

Determine which type of test is best for your use case.

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

The **Low-Level Tests (LLTs)** framework recognizes the following types of tests:

- **Explicit**: Self-contained tests defined by a module build-target pair.

Explicit Tests

Explicit Tests are self-contained tests defined by a module build-target pair. Explicit tests are designed to be lightweight in terms of compilation time and run time. They are called *explicit* because they require explicit UE module build and target files. This means explicit tests require both a `.Build.cs` and a `.Target.cs` file.

Create an Explicit Test

Follow these steps to create your explicit test:

1. In the `Source/Programs` directory, create a new directory with the same name as the module you want to test and add the `.Build.cs` file in this directory.

   > To see an example, the directory `Engine/Source/Programs/LowLevelTests` contains an explicit test target named Foundation Tests.

2. Inherit your module class from `TestModuleRules`.
   - If you are writing a test for a plugin, place the new module inside a `Tests` directory at the same level as the plugin's `Source` directory.
   - If you are building a test module that does not use Catch2, inherit the base constructor with the second parameter set to false: `base(Target, false)`.
3. Call `UpdateBuildGraphPropertiesFile` with a new `Metadata` object argument.
   - This information is used to generate BuildGraph script test metadata.
   - For more information about BuildGraph script generation, see the Generate BuildGraph Script Metadata section.
4. Suppose that you have an explicit test module titled `UEModuleTests`. Your explicit test `.Build.cs` file should look similar to this:

   UEModuleTests/UEModuleTests.Build.cs

```
public class UEModuleTests : TestModuleRules
{
public UEModuleTests(ReadOnlyTargetRules Target) : base(Target)
{
PrivateIncludePaths.AddRange(
// Any private include paths
);
PrivateDependencyModuleNames.AddRange(
// Any private dependencies to link against
);

// Other types of dependencies or module specific logic

UpdateBuildGraphPropertiesFile(new Metadata("UEModule", "UE Module"));
}
}
```

Copy full snippet

5. Add a test target file (`.Target.cs`) with a class that inherits from `TestTargetRules`.
6. Override the default compilation flags if necessary.
    - Aim for a minimal, testable module free of default features that don't add value to low-level testing.
    - You can specify the supported platforms individually. The default platforms are: Win64, Mac, Linux, and Android.
    - You can enable project-specific global definitions and set Catch2 definitions, such as those needed for benchmarking support.
7. Your explicit tests `.Target.cs` file should look similar to this:

    UEModuleTests/UEModuleTests.Target.cs

```
[SupportedPlatforms(UnrealPlatformClass.All)]
public class UEModuleTestsTarget : TestTargetRules
```

```
{
public UEModuleTestsTarget(TargetInfo Target) : base(Target)
{
// Setup like any other target: set compilation flags, global definitions etc.
GlobalDefinitions.Add("CATCH_CONFIG_ENABLE_BENCHMARKING=1");
}
}
```

Copy full snippet

Next Steps

Now you can write C++ test files in the `Private` folder of the module and write Catch2 tests in these files. For testing tips and best practices, see the Write Low-Level Tests documentation. Lastly, learn how to build and run your tests. There is more than one way to build and execute low-level tests. See the Build and Run Low-Level Tests documentation to select the best method for your development needs.

Generate BuildGraph Script Metadata Files

If you want to build and run your tests with BuildGraph, you need to enable generation of BuildGraph script metadata files for explicit tests. When generating the IDE solution via `GenerateProjectFiles.bat`, the explicit test modules generate BuildGraph `.xml` files.

An engine configuration setting conditions this generation. You can set this configuration in `Engine/Config/BaseEngine.ini`:

```
[LowLevelTestsSettings]
bUpdateBuildGraphPropertiesFile=true
```

Copy full snippet

When you run `GenerateProjectFiles.bat`, test metadata `.xml` files are generated in the `Build/LowLevelTests/<TEST_NAME>.xml` folder for each test target, where `<TEST_NAME>` is the name of your test target. For NDA platforms, these files are generated under `Platforms/<PLATFORM_NAME>/Build/LowLevelTests/<TEST_NAME>.xml`. An additional `General.xml` file is optionally present next to the test files containing global properties.

If the files already exist, they are updated according to the C#-described `Metadata` object. The folders and files that are accessed by project file generation must be writable. Typically, these files are read-only when under source control, so check them out or make them writeable before generation.

To see an example, the directory `Engine/Build/LowLevelTests` contains an `.xml` file named Foundation. This is the generated BuildGraph metadata for the Foundation Tests.

Explicit Tests Reference

Test Module Rules Reference

The `TestModuleRules` class extends `ModuleRules` with `UpdateBuildGraphPropertiesFile`. `UpdateBuildGraphPropertiesFile` accepts a `Metadata` object which generates BuildGraph test metadata `.xml` files. With a `Metadata` object, you can set the following properties:

Field	Description
`TestName`	The name of your tests used by the BuildGraph script to generate test-specific properties. This field cannot contain spaces.
`TestShortName`	The short name of your tests used for display in the build system. This field can contain spaces.

Field	Description
`ReportType`	The Catch2 report type. The most common report types are console and xml. For more information about Catch2 report types, see the external Catch2 documentation.
`Disabled`	Whether the test is disabled. If true, this test is excluded from the BuildGraph graph.
`InitialExtraArgs`	Command-line arguments that are prepended in front of other arguments for the `RunLowLevelTests` Gauntlet command. These are typically Gauntlet feature-enabling arguments that only apply to some tests. For example, `-printreport`, which prints the report to `stdout` at the end of test execution.
`HasAfterSteps`	If true, tests must provide a BuildGraph `Macro` with the name `<TEST_NAME>AfterSteps` that include any cleanup or steps necessary to run after the test execution. For example, this could be running a database script that cleans up any leftover test data.
`UsesCatch2`	This property allows you to choose your test framework. Some tests don't use Catch2; they might use `GoogleTest` for example. If you choose your own test framework, ensure that you implement support for reporting and other features in the `RunLowLevelTests` Gauntlet command.
`PlatformTags`	Platform-specific list of tags. For example, use this to exclude unsupported tests on a given platform.
`PlatformCompilationExtraArgs`	Any extra compilation arguments that a platform might require.
`PlatformsRunUnsupported`	Add an exception and can serve as a compilation safety net in the BuildGraph script until running support is implemented. For example, if a platform only supports compilation but lacks low-level test running capabilities.

`TestModuleRules` overrides many default UBT flags from its base class `ModuleRules`. This reduces compilation bloat and minimizes compilation times for most tests out of the box. You can always override these defaults in your `TestModuleRules` derived class, but they should not be changed in `TestModuleRules` directly.

Test Target Rules Reference

The `TestTargetRules` class extends `TargetRules` with the following:

Flag	Description
`bUsePlatformFileStub`	This causes the platform-dependent `FPlatformFile` runtime instance to be replaced with a mock that disables IO operations. Use this to disable asset loading when testing against the engine module. Setting this property changes the value of `UE_LLT_USE_PLATFORM_FILE_STUB`, which tests can use to perform additional IO mocking. The `FPlatformFile` is saved using `SaveDefaultPlatformFile` and restored with `UseDefaultPlatformFile`, both of which require `#include "TestCommon/Initialization.h"`.
`bMockEngineDefaults`	When testing with the engine module, certain resources are managed by default or loaded from asset files. These operations require cooking assets. Use this for tests that don't need to load assets; the effect is to mock engine default materials, world objects, and other resources.

Flag	Description
	Setting this property changes the value of `UE_LLT_WITH_MOCK_ENGINE_DEFAULTS`.
`bNeverCompileAgainstEngine`	The default behavior of the LLT Framework is to automatically set `bCompileAgainstEngine = true` whenever the engine module is in the build dependency graph. This property can stop this behavior such that we're not compiling with the engine even if the engine module is in the graph.
`bNeverCompileAgainstCoreUObject`	Same as `bNeverCompileAgainstEngine`, but for `CoreUObject`.
`bNeverCompileAgainstApplicationCore`	Same as `bNeverCompileAgainstEngine`, but for `ApplicationCore`.
`bNeverCompileAgainstEditor`	Same as `bNeverCompileAgainstEngine`, but for `UnrealEd`.

Just like `TestModuleRules`, `TestTargetRules` sets default UBT flags. Notably it disables UE features such as UObjects, localizations, stats, and others.

Engine tests

In this type of explicit test, the LLT framework compiles and runs explicit tests that include the engine module. Because loading assets requires cooking for most platforms, the engine module cannot be used, so engine tests only work with the following flags set in the `.Target.cs` file:

```
1 public UEModuleTestsTarget(TargetInfo Target) : base(Target)
```

```
{
bUsePlatformFileStub = true;
bMockEngineDefaults = true;
}
```

Copy full snippet

Next Step

Once you have decided which test is right for your needs, see the Write Low-Level Tests documentation to learn how to write Low-Level Tests in Unreal Engine.

Write Low-Level Tests

Learn how to write Low-Level Tests in Unreal Engine, including naming conventions and best practices.