Starting with 5.3, **CMake** is now the default build system for CRYENGINE, replacing WAF. This page will describe the structure of the CMake scripts.

If you are looking on information on the differences between CMake and WAF, please refer to the page **Migrating from WAF to CMake**.

- Bundled CMake Binaries
  - Msvc-android-cmake
  - Win32
- CMake Options
- Adding a Module
  - Specifying Source Files
  - Module Types
  - Other Commands

Bundled CMake Binaries

When building on Windows, we provide pre-compiled binaries for CMake. There are two sets of binaries, used for different platforms. When using the batch files found in **`Tools\CMake`**, the appropriate binaries will be used automatically.

Msvc-android-cmake

These binaries contain customizations by Microsoft to make use of the Android project support in Visual Studio. They are used only for Android builds.

Win32

These binaries are used for all other platforms that can be built from Windows. They contain a small number of bug fixes and customizations to ensure that CRYENGINE builds as intended.

When creating Windows builds, a vanilla CMake install can also be used, but it may require a newer version that what we provide.

CMake Options

When building the engine, you have the ability to customize the solution in various ways. This list gives an overview of the most important options.

Some of these categories and options do not apply on all platforms.

Category/Option	Description
AUDIO_*	Specifies which audio modules will be built.
METADATA_*	Specifies metadata that will be linked into the build, such as version number and project name.
OPTION_CRYMONO	Enable C#/Mono support.
OPTION_ENABLE_BROFILER	Enable Brofiler support for profiling.
OPTION_ENABLE_CRASHRPT	Build the crash reporting used to report errors back to Crytek.
PROJECT_CRYENGINE_GAMESDK	Add GameSDK to the solution.
PROJECT_CRYENGINE_GAMEZERO	Add GameZero to the solution.
OPTION_PCH	Enable precompiled headers when building. Windows only. Not used when building unity/uber files.
OPTION_PGO	• **Off -** regular compilation. • **Generate -** the compiler will add extra code to the executable to gather profiling data while running. This results in extra data files (*.pgc and *.pgd for MSVC), which must be retained for the next step. • **Use -** the compiler uses the gathered data to direct its optimization efforts, which may result in a faster program. **NOTE:** For MSVC, PGO requires LTCG, so this will be turned on automatically if PGO is enabled.
OPTION_PROFILE	Enables various profiling tools even when building release configuration. Generally not necessary.
OPTION_RC	Adds the Resource Compiler as an external project to your solution.
OPTION_SANDBOX	Include Sandbox in the solution. When this is enabled, only Debug and Profile builds can be built; Release builds are not available.

Confidential. © 2009-2015 Crytek GmbH. All Rights Reserved.

OPTION_STATIC_LINKING	Create a monolithic build (single executable).
OPTION_UNITY_BUILD	Use uber files (AKA unity files) when building.
PHYSICS_*	Specifies which physics modules will be built.
PLUGIN_*	Each option here enables an engine plugin, such as VR support plugins or Schematyc.
RENDERER_*	Specifies which renderer modules will be built.

Adding a Module

Standard CMake commands are sufficient for adding new modules. However, the CMake scripts do provide some additional support functionality, implemented as macros and functions in **`Tools\CMake\CommonMacros.cmake`**. Using these macros will greatly simplify the amount of setup you need to do, e.g. by handling uber file generation automatically.

This is not a complete listing of macros and functions provided by that file, but it should cover everything you might need for your own projects. Additional macros used internally by the CMake scripts will not be documented here.

If you are migrating from an older version of CRYENGINE, based on WAF, please refer to the page **Migrating from WAF to CMake** for a more detailed explanation of how to map WAF projects onto CMake.

Once you have created a `CMakeLists.txt` for a module, simply add it with `add_subdirectory` in the corresponding `Build*.cmake` file in `Tools\CMake` (e.g. `BuildEngine.cmake` for an engine module, or `BuildSandbox.cmake` for a sandbox module).

Specifying Source Files

Since 5.3 supports both WAF and CMake as build systems, we have created a few custom macros to allow the same file structure to be represented. These are used by the `waf2cmake` tool, explained in the **Migrating from WAF to CMake** guide, but are also available for newly added CMake modules.

Macro	Description
begin_sources()	Use this before specifying any source files to clear any internal state and prepare for new source files. Should only be used once per file.
end_sources()	Use this after specifying all source files. Should only be used once per file.
sources_platform([OR] platform ... [AND platform ...])	Specifies a set of platforms that should build source files provided after this point. Specify ALL to build for all platforms. Source files specified after this call will be built if "if(T)" is true for at least one token T in `OR`, as well as all tokens T in `AND`. Files that do not get built for a given platform will still appear in the solution, if they exist on disk. Example: **`sources_platform(WIN32 ANDROID AND HAS_FOO)`** means "build if target is Windows or Android, and `HAS_FOO` is true".
add_sources(uberfile [PROJECTS project ...] (SOURCE_GROUP "GroupName" file ...) ...)	Specifies a set of source files to include in the solution. Files will be built if they match the latest sources_platform call. Uberfile signals the file name used for the uber file associated with these source files. Specify NoUberFile to not use uber files with these source files. `PROJECTS` is optional, but should be used when building multiple projects with distinct sets of source files from one directory. If a project is never referenced in `PROJECTS`, it will use all source files given in this directory. Each `SOURCE_GROUP` directive specifies a source group that will be used to group the following files in the Visual Studio solution. The first argument following `SOURCE_GROUP` is the group name, all following arguments will be treated as file names until the next `SOURCE_GROUP` is encountered.

Module Types

As with WAF, modules are categorized according to their type, and different compilation settings get applied automatically as a result. Most of these categories are carried over straight from WAF, simplifying conversion.

The general form for these module declarations take the following form (with `CryEngineModule` being the name given to the category):

```
CryEngineModule(module_name [PCH file] [SOLUTION_FOLDER folder] [extra])
```

`Module_name` must be the first argument; additional arguments may appear in any order.

The most common module types are as follows:

Module Type	Description

Confidential. © 2009-2015 Crytek GmbH. All Rights Reserved.

<table>
<tr><td>CryEngineModule</td><td>Standard engine module.

Additional arguments:

<table><tr><td>FORCE_STATIC</td><td>Forces static linking for this module.</td></tr><tr><td>FORCE_SHARED</td><td>Forces shared library (if supported for the given platform).</td></tr></table></td></tr>
<tr><td>CryEngineStaticModule</td><td>Standard engine module that always gets linked statically.</td></tr>
<tr><td>CryGameModule</td><td>Game DLL module. This module type is new in CMake.</td></tr>
<tr><td>CreateDynamicModule</td><td>Stand-alone DLL that does not depend on any CRYENGINE-specific code.</td></tr>
<tr><td>CryFileContainer</td><td>A collection of files that does not directly participate in the build (e.g. header-only libraries). The <code>PCH</code> argument is not used for this type.</td></tr>
<tr><td>CryPlugin</td><td>Sandbox plugin; automatically links against Qt.

Additional arguments:

<table><tr><td>DISABLE_MFC</td><td>Do not link against MFC for this plugin.</td></tr></table></td></tr>
</table>

Most code should be using `CryEngineModule` or `CryPlugin`.

Other Commands

These additional commands are rarely required, but may be useful in select scenarios.

Command	Description
force_static_crt()	Force static linking against the CRT. Only implemented for Windows.
use_mono()	If you need to link to Mono, use this macro.
use_scaleform()	If you need to link a module against Scaleform, use this macro.
use_xt()	If you write a Sandbox plugin using MFC, use this macro to link against the XT library.

Confidential. © 2009-2015 Crytek GmbH. All Rights Reserved.