NAME

cmake-generators − CMake Generators Reference

INTRODUCTION

A *CMake Generator* is responsible for writing the input files for a native build system. Exactly one of the *CMake Generators* must be selected for a build tree to determine what native build system is to be used. Optionally one of the *Extra Generators* may be selected as a variant of some of the *Command−Line Build Tool Generators* to produce project files for an auxiliary IDE.

CMake Generators are platform−specific so each may be available only on certain platforms. The **cmake(1)** command−line tool **−−help** output lists available generators on the current platform. Use its **−G** option to specify the generator for a new build tree. The **cmake−gui(1)** offers interactive selection of a generator when creating a new build tree.

CMAKE GENERATORS

Command−Line Build Tool Generators

These generators support command−line build tools. In order to use them, one must launch CMake from a command−line prompt whose environment is already configured for the chosen compiler and build tool.

Makefile Generators

Borland Makefiles

Generates Borland makefiles.

MSYS Makefiles

Generates makefiles for use with MSYS (Minimal SYStem) **make** under the MSYS shell.

Use this generator in a MSYS shell prompt and using **make** as the build tool. The generated makefiles use **/bin/sh** as the shell to launch build rules. They are not compatible with a Windows command prompt.

To build under a Windows command prompt, use the **MinGW Makefiles** generator.

MinGW Makefiles

Generates makefiles for use with **mingw32−make** under a Windows command prompt.

Use this generator under a Windows command prompt with MinGW (Minimalist GNU for Windows) in the **PATH** and using **mingw32−make** as the build tool. The generated makefiles use **cmd.exe** as the shell to launch build rules. They are not compatible with MSYS or a unix shell.

To build under the MSYS shell, use the **MSYS Makefiles** generator.

NMake Makefiles

Generates NMake makefiles.

NMake Makefiles JOM

Generates JOM makefiles.

New in version 3.8: **CodeBlocks** generator can be used as an extra generator.

Unix Makefiles

Generates standard UNIX makefiles.

A hierarchy of UNIX makefiles is generated into the build tree. Use any standard UNIX−style make program to build the project through the **all** target and install the project through the **install** (or **install/strip**) target.

For each subdirectory **sub/dir** of the project a UNIX makefile will be created, containing the following targets:

all Depends on all targets required by the subdirectory.

install Runs the install step in the subdirectory, if any.

install/strip
Runs the install step in the subdirectory followed by a **CMAKE_STRIP** command, if any.

The **CMAKE_STRIP** variable will contain the platform's **strip** utility, which removes symbols information from generated binaries.

test Runs the test step in the subdirectory, if any.

package
Runs the package step in the subdirectory, if any.

Watcom WMake
Generates Watcom WMake makefiles.

Ninja Generators

Ninja
Generates **build.ninja** files.

A **build.ninja** file is generated into the build tree. Use the ninja program to build the project through the **all** target and install the project through the **install** (or **install/strip**) target.

For each subdirectory **sub/dir** of the project, additional targets are generated:

sub/dir/all
New in version 3.6: Depends on all targets required by the subdirectory.

sub/dir/install
New in version 3.7: Runs the install step in the subdirectory, if any.

sub/dir/install/strip
New in version 3.7: Runs the install step in the subdirectory followed by a **CMAKE_STRIP** command, if any.

The **CMAKE_STRIP** variable will contain the platform's **strip** utility, which removes symbols information from generated binaries.

sub/dir/test
New in version 3.7: Runs the test step in the subdirectory, if any.

sub/dir/package
New in version 3.7: Runs the package step in the subdirectory, if any.

Fortran Support
New in version 3.7.

The **Ninja** generator conditionally supports Fortran when the **ninja** tool is at least version 1.10 (which has the required features).

Swift Support
New in version 3.15.

The Swift support is experimental, not considered stable, and may change in future releases of CMake.

See Also
New in version 3.17: The **Ninja Multi−Config** generator is similar to the **Ninja** generator, but generates multiple configurations at once.

Ninja Multi−Config
New in version 3.17.

Generates multiple **build−<Config>.ninja** files.

This generator is very much like the **Ninja** generator, but with some key differences. Only these differences will be discussed in this document.

Unlike the **Ninja** generator, **Ninja Multi−Config** generates multiple configurations at once with **CMAKE_CONFIGURATION_TYPES** instead of only one configuration with **CMAKE_BUILD_TYPE**. One **build−<Config>.ninja** file will be generated for each of these configurations (with **<Config>** being the configuration name.) These files are intended to be run with **ninja −f build−<Config>.ninja**. A **build.ninja** file is also generated, using the configuration from either **CMAKE_DEFAULT_BUILD_TYPE** or the first item from **CMAKE_CONFIGURATION_TYPES**.

cmake −−build . −−config <Config> will always use **build−<Config>.ninja** to build. If no **−−config** argument is specified, **cmake −−build .** will use **build.ninja**.

Each **build−<Config>.ninja** file contains **<target>** targets as well as **<target>:<Config>** targets, where **<Config>** is the same as the configuration specified in **build−<Config>.ninja** Additionally, if cross−config mode is enabled, **build−<Config>.ninja** may contain **<target>:<OtherConfig>** targets, where **<OtherConfig>** is a cross−config, as well as **<target>:all**, which builds the target in all cross−configs. See below for how to enable cross−config mode.

The **Ninja Multi−Config** generator recognizes the following variables:

CMAKE_CONFIGURATION_TYPES
Specifies the total set of configurations to build. Unlike with other multi−config generators, this variable has a value of **Debug;Release;RelWithDebInfo** by default.

CMAKE_CROSS_CONFIGS
Specifies a semicolon−separated list of configurations available from all **build−<Config>.ninja** files.

CMAKE_DEFAULT_BUILD_TYPE
Specifies the configuration to use by default in a **build.ninja** file.

CMAKE_DEFAULT_CONFIGS
Specifies a semicolon−separated list of configurations to build for a target in **build.ninja** if no **:<Config>** suffix is specified.

Consider the following example:

```
cmake_minimum_required(VERSION 3.16)
project(MultiConfigNinja C)
```

```
add_executable(generator generator.c)
add_custom_command(OUTPUT generated.c COMMAND generator generated.c)
add_library(generated ${CMAKE_BINARY_DIR}/generated.c)
```

Now assume you configure the project with **Ninja Multi−Config** and run one of the following commands:

```
ninja -f build-Debug.ninja generated
# OR
cmake --build . --config Debug --target generated
```

This would build the **Debug** configuration of **generator**, which would be used to generate **generated.c**, which would be used to build the **Debug** configuration of **generated**.

But if **CMAKE_CROSS_CONFIGS** is set to **all**, and you run the following instead:

```
ninja -f build-Release.ninja generated:Debug
# OR
cmake --build . --config Release --target generated:Debug
```

This would build the **Release** configuration of **generator**, which would be used to generate **generated.c**, which would be used to build the **Debug** configuration of **generated**. This is useful for running a release−optimized version of a generator utility while still building the debug version of the targets built with the generated code.

Custom Commands

New in version 3.20.

The **Ninja Multi−Config** generator adds extra capabilities to **add_custom_command()** and **add_custom_target()** through its cross−config mode. The **COMMAND**, **DEPENDS**, and **WORKING_DIRECTORY** arguments can be evaluated in the context of either the "command config" (the "native" configuration of the **build−<Config>.ninja** file in use) or the "output config" (the configuration used to evaluate the **OUTPUT** and **BYPRODUCTS**).

If either **OUTPUT** or **BYPRODUCTS** names a path that is common to more than one configuration (e.g. it does not use any generator expressions), all arguments are evaluated in the command config by default. If all **OUTPUT** and **BYPRODUCTS** paths are unique to each configuration (e.g. by using the **$<CONFIG>** generator expression), the first argument of **COMMAND** is still evaluated in the command config by default, while all subsequent arguments, as well as the arguments to **DEPENDS** and **WORKING_DIRECTORY**, are evaluated in the output config. These defaults can be overridden with the **$<OUTPUT_CONFIG:...>** and **$<COMMAND_CONFIG:...>** generator−expressions. Note that if a target is specified by its name in **DEPENDS**, or as the first argument of **COMMAND**, it is always evaluated in the command config, even if it is wrapped in **$<OUTPUT_CONFIG:...>** (because its plain name is not a generator expression).

As an example, consider the following:

```
add_custom_command(
  OUTPUT "$<CONFIG>.txt"
  COMMAND generator "$<CONFIG>.txt" "$<OUTPUT_CONFIG:$<CONFIG>>" "$<COMMAND_CO
  DEPENDS tgt1 "$<TARGET_FILE:tgt2>" "$<OUTPUT_CONFIG:$<TARGET_FILE:tgt3>>" "$
  )
```

Assume that **generator**, **tgt1**, **tgt2**, **tgt3**, and **tgt4** are all executable targets, and assume that **$<CONFIG>.txt** is built in the **Debug** output config using the **Release** command config. The **Release** build of the

generator target is called with **Debug.txt Debug Release** as arguments. The command depends on the **Release** builds of **tgt1** and **tgt4**, and the **Debug** builds of **tgt2** and **tgt3**.

PRE_BUILD, **PRE_LINK**, and **POST_BUILD** custom commands for targets only get run in their "native" configuration (the **Release** configuration in the **build−Release.ninja** file) unless they have no **BYPRODUCTS** or their **BYPRODUCTS** are unique per config. Consider the following example:

```
add_executable(exe main.c)
add_custom_command(
  TARGET exe
  POST_BUILD
  COMMAND ${CMAKE_COMMAND} −E echo "Running no−byproduct command"
  )
add_custom_command(
  TARGET exe
  POST_BUILD
  COMMAND ${CMAKE_COMMAND} −E echo "Running separate−byproduct command for $<C
  BYPRODUCTS $<CONFIG>.txt
  )
add_custom_command(
  TARGET exe
  POST_BUILD
  COMMAND ${CMAKE_COMMAND} −E echo "Running common−byproduct command for $<CON
  BYPRODUCTS exe.txt
  )
```

In this example, if you build **exe:Debug** in **build−Release.ninja**, the first and second custom commands get run, since their byproducts are unique per−config, but the last custom command does not. However, if you build **exe:Release** in **build−Release.ninja**, all three custom commands get run.

IDE Build Tool Generators

These generators support Integrated Development Environment (IDE) project files. Since the IDEs configure their own environment one may launch CMake from any environment.

Visual Studio Generators

Visual Studio 6

Removed. This once generated Visual Studio 6 project files, but the generator has been removed since CMake 3.6. It is still possible to build with VS 6 tools using the **NMake Makefiles** generator.

Visual Studio 7

Removed. This once generated Visual Studio .NET 2002 project files, but the generator has been removed since CMake 3.6. It is still possible to build with VS 7.0 tools using the **NMake Makefiles** generator.

Visual Studio 7 .NET 2003

Removed. This once generated Visual Studio .NET 2003 project files, but the generator has been removed since CMake 3.9. It is still possible to build with VS 7.1 tools using the **NMake Makefiles** generator.

Visual Studio 8 2005

Removed. This once generated Visual Studio 8 2005 project files, but the generator has been removed since CMake 3.12. It is still possible to build with VS 2005 tools using the **NMake Makefiles** generator.

Visual Studio 9 2008

Generates Visual Studio 9 2008 project files.

Platform Selection

The default target platform name (architecture) is **Win32**.

New in version 3.1: The **CMAKE_GENERATOR_PLATFORM** variable may be set, perhaps via the **cmake(1) −A** option, to specify a target platform name (architecture). For example:

- **cmake −G "Visual Studio 9 2008" −A Win32**
- **cmake −G "Visual Studio 9 2008" −A x64**
- **cmake −G "Visual Studio 9 2008" −A Itanium**
- **cmake −G "Visual Studio 9 2008" −A <WinCE−SDK>** (Specify a target platform matching a Windows CE SDK name.)

For compatibility with CMake versions prior to 3.1, one may specify a target platform name optionally at the end of the generator name. This is supported only for:

Visual Studio 9 2008 Win64
Specify target platform **x64**.

Visual Studio 9 2008 IA64
Specify target platform **Itanium**.

Visual Studio 9 2008 <WinCE−SDK>
Specify target platform matching a Windows CE SDK name.

Visual Studio 10 2010
Deprecated. Generates Visual Studio 10 (VS 2010) project files.

NOTE:
This generator is deprecated and will be removed in a future version of CMake. It will still be possible to build with VS 10 2010 tools using the **Visual Studio 11 2012** (or above) generator with **CMAKE_GENERATOR_TOOLSET** set to **v100**, or by using the **NMake Makefiles** generator.

For compatibility with CMake versions prior to 3.0, one may specify this generator using the name **Visual Studio 10** without the year component.

Project Types
Only Visual C++ and C# projects may be generated (and Fortran with Intel compiler integration). Other types of projects (Database, Website, etc.) are not supported.

Platform Selection
The default target platform name (architecture) is **Win32**.

New in version 3.1: The **CMAKE_GENERATOR_PLATFORM** variable may be set, perhaps via the **cmake(1) −A** option, to specify a target platform name (architecture). For example:

- **cmake −G "Visual Studio 10 2010" −A Win32**
- **cmake −G "Visual Studio 10 2010" −A x64**
- **cmake −G "Visual Studio 10 2010" −A Itanium**

For compatibility with CMake versions prior to 3.1, one may specify a target platform name optionally at the end of the generator name. This is supported only for:

Visual Studio 10 2010 Win64
Specify target platform **x64**.

Visual Studio 10 2010 IA64
Specify target platform **Itanium**.

Toolset Selection
The **v100** toolset that comes with Visual Studio 10 2010 is selected by default. The**CMAKE_GENERATOR_TOOLSET** option may be set, perhaps via the **cmake(1) −T** option, to specify another toolset.

Visual Studio 11 2012
Generates Visual Studio 11 (VS 2012) project files.

For compatibility with CMake versions prior to 3.0, one may specify this generator using the name "Visual Studio 11" without the year component.

Project Types
Only Visual C++ and C# projects may be generated (and Fortran with Intel compiler integration). Other types of projects (JavaScript, Database, Website, etc.) are not supported.

Platform Selection
The default target platform name (architecture) is **Win32**.

New in version 3.1: The **CMAKE_GENERATOR_PLATFORM** variable may be set, perhaps via the **cmake(1) −A** option, to specify a target platform name (architecture). For example:

- **cmake −G "Visual Studio 11 2012" −A Win32**
- **cmake −G "Visual Studio 11 2012" −A x64**
- **cmake −G "Visual Studio 11 2012" −A ARM**
- **cmake −G "Visual Studio 11 2012" −A <WinCE−SDK>** (Specify a target platform matching a Windows CE SDK name.)

For compatibility with CMake versions prior to 3.1, one may specify a target platform name optionally at the end of the generator name. This is supported only for:

Visual Studio 11 2012 Win64
Specify target platform **x64**.

Visual Studio 11 2012 ARM
Specify target platform **ARM**.

Visual Studio 11 2012 <WinCE−SDK>
Specify target platform matching a Windows CE SDK name.

Toolset Selection
The **v110** toolset that comes with Visual Studio 11 2012 is selected by default. The**CMAKE_GENERATOR_TOOLSET** option may be set, perhaps via the **cmake(1) −T** option, to specify another toolset.

Visual Studio 12 2013
Generates Visual Studio 12 (VS 2013) project files.

For compatibility with CMake versions prior to 3.0, one may specify this generator using the name "Visual Studio 12" without the year component.

Project Types
Only Visual C++ and C# projects may be generated (and Fortran with Intel compiler integration). Other types of projects (JavaScript, Powershell, Python, etc.) are not supported.

Platform Selection
The default target platform name (architecture) is **Win32**.

New in version 3.1: The **CMAKE_GENERATOR_PLATFORM** variable may be set, perhaps via the **cmake(1) −A** option, to specify a target platform name (architecture). For example:

- **cmake −G "Visual Studio 12 2013" −A Win32**
- **cmake −G "Visual Studio 12 2013" −A x64**
- **cmake −G "Visual Studio 12 2013" −A ARM**

For compatibility with CMake versions prior to 3.1, one may specify a target platform name optionally at the end of the generator name. This is supported only for:

Visual Studio 12 2013 Win64
Specify target platform **x64**.

Visual Studio 12 2013 ARM
Specify target platform **ARM**.

Toolset Selection
The **v120** toolset that comes with Visual Studio 12 2013 is selected by default. The**CMAKE_GENERATOR_TOOLSET** option may be set, perhaps via the **cmake(1) −T** option, to specify another toolset.

New in version 3.8: For each toolset that comes with this version of Visual Studio, there are variants that are themselves compiled for 32−bit (**x86**) and 64−bit (**x64**) hosts (independent of the architecture they target). By default this generator uses the 32−bit variant even on a 64−bit host. One may explicitly request use of either the 32−bit or 64−bit host tools by adding either **host=x86** or **host=x64** to the toolset specification. See the**CMAKE_GENERA TOR_TOOLSET** variable for details.

New in version 3.14: Added support for **host=x86** option.

Visual Studio 14 2015
New in version 3.1.

Generates Visual Studio 14 (VS 2015) project files.

Project Types
Only Visual C++ and C# projects may be generated (and Fortran with Intel compiler integration). Other types of projects (JavaScript, Powershell, Python, etc.) are not supported.

Platform Selection
The default target platform name (architecture) is **Win32**.

The **CMAKE_GENERATOR_PLATFORM** variable may be set, perhaps via the **cmake(1) −A** option, to specify a target platform name (architecture). For example:

- **cmake −G "Visual Studio 14 2015" −A Win32**
- **cmake −G "Visual Studio 14 2015" −A x64**
- **cmake −G "Visual Studio 14 2015" −A ARM**

For compatibility with CMake versions prior to 3.1, one may specify a target platform name optionally at the end of the generator name. This is supported only for:

Visual Studio 14 2015 Win64
Specify target platform **x64**.

Visual Studio 14 2015 ARM
Specify target platform **ARM**.

Toolset Selection
The **v140** toolset that comes with Visual Studio 14 2015 is selected by default. The**CMAKE_GENERATOR_TOOLSET** option may be set, perhaps via the **cmake(1) −T** option, to specify another toolset.

New in version 3.8: For each toolset that comes with this version of Visual Studio, there are variants that are themselves compiled for 32−bit (**x86**) and 64−bit (**x64**) hosts (independent of the architecture they target). By default this generator uses the 32−bit variant even on a 64−bit host. One may explicitly request use of either the 32−bit or 64−bit host tools by adding either **host=x86** or **host=x64** to the toolset

specification. See the**CMAKE_GENERATOR_TOOLSET** variable for details.

New in version 3.14: Added support for **host=x86** option.

Windows 10 SDK Maximum Version for VS 2015
New in version 3.19.

Microsoft stated in a "Windows 10 October 2018 Update" blog post that Windows 10 SDK versions (15063, 16299, 17134, 17763) are not supported by VS 2015 and are only supported by VS 2017 and later. Therefore by default CMake automatically ignores Windows 10 SDKs beyond **10.0.14393.0**.

However, there are other recommendations for certain driver/Win32 builds that indicate otherwise. A user can override this behavior by either setting the **CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION_MAXIMUM** to a false value or setting the **CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION_MAXIMUM** to the string value of the required maximum (e.g. **10.0.15063.0**).

Visual Studio 15 2017
New in version 3.7.1.

Generates Visual Studio 15 (VS 2017) project files.

Project Types
Only Visual C++ and C# projects may be generated (and Fortran with Intel compiler integration). Other types of projects (JavaScript, Powershell, Python, etc.) are not supported.

Instance Selection
New in version 3.11.

VS 2017 supports multiple installations on the same machine. The **CMAKE_GENERATOR_INSTANCE** variable may be set as a cache entry containing the absolute path to a Visual Studio instance. If the value is not specified explicitly by the user or a toolchain file, CMake queries the Visual Studio Installer to locate VS instances, chooses one, and sets the variable as a cache entry to hold the value persistently.

When CMake first chooses an instance, if the **VS150COMNTOOLS** environment variable is set and points to the **Common7/Tools** directory within one of the instances, that instance will be used. Otherwise, if more than one instance is installed we do not define which one is chosen by default.

Platform Selection
The default target platform name (architecture) is **Win32**.

The **CMAKE_GENERATOR_PLATFORM** variable may be set, perhaps via the **cmake(1) −A** option, to specify a target platform name (architecture). For example:

- **cmake −G "Visual Studio 15 2017" −A Win32**
- **cmake −G "Visual Studio 15 2017" −A x64**
- **cmake −G "Visual Studio 15 2017" −A ARM**
- **cmake −G "Visual Studio 15 2017" −A ARM64**

For compatibility with CMake versions prior to 3.1, one may specify a target platform name optionally at the end of the generator name. This is supported only for:

Visual Studio 15 2017 Win64
Specify target platform **x64**.

Visual Studio 15 2017 ARM
Specify target platform **ARM**.

Toolset Selection
The **v141** toolset that comes with Visual Studio 15 2017 is selected by default. The**CMAKE_GENERATOR_TOOLSET** option may be set, perhaps via the **cmake(1) −T** option, to specify another toolset.

New in version 3.8: For each toolset that comes with this version of Visual Studio, there are variants that are themselves compiled for 32−bit (**x86**) and 64−bit (**x64**) hosts (independent of the architecture they target). By default this generator uses the 32−bit variant even on a 64−bit host. One may explicitly request use of either the 32−bit or 64−bit host tools by adding either **host=x86** or **host=x64** to the toolset specification. See the**CMAKE_GENERATOR_TOOLSET** variable for details.

New in version 3.14: Added support for **host=x86** option.

Visual Studio 16 2019
New in version 3.14.

Generates Visual Studio 16 (VS 2019) project files.

Project Types
Only Visual C++ and C# projects may be generated (and Fortran with Intel compiler integration). Other types of projects (JavaScript, Powershell, Python, etc.) are not supported.

Instance Selection
VS 2019 supports multiple installations on the same machine. The **CMAKE_GENERATOR_INSTANCE** variable may be set as a cache entry containing the absolute path to a Visual Studio instance. If the value is not specified explicitly by the user or a toolchain file, CMake queries the Visual Studio Installer to locate VS instances, chooses one, and sets the variable as a cache entry to hold the value persistently.

When CMake first chooses an instance, if the **VS160COMNTOOLS** environment variable is set and points to the **Common7/Tools** directory within one of the instances, that instance will be used. Otherwise, if more than one instance is installed we do not define which one is chosen by default.

Platform Selection
The default target platform name (architecture) is that of the host and is provided in the **CMAKE_VS_PLATFORM_NAME_DEFAULT** variable.

The **CMAKE_GENERATOR_PLATFORM** variable may be set, perhaps via the **cmake(1) −A** option, to specify a target platform name (architecture). For example:

- **cmake −G "Visual Studio 16 2019" −A Win32**
- **cmake −G "Visual Studio 16 2019" −A x64**
- **cmake −G "Visual Studio 16 2019" −A ARM**
- **cmake −G "Visual Studio 16 2019" −A ARM64**

Toolset Selection
The **v142** toolset that comes with Visual Studio 16 2019 is selected by default. The**CMAKE_GENERATOR_TOOLSET** option may be set, perhaps via the **cmake(1) −T** option, to specify another toolset.

For each toolset that comes with this version of Visual Studio, there are variants that are themselves compiled for 32−bit (**x86**) and 64−bit (**x64**) hosts (independent of the architecture they target). By default this

generator uses the 64−bit variant on x64 hosts and the 32−bit variant otherwise. One may explicitly request use of either the 32−bit or 64−bit host tools by adding either **host=x86** or **host=x64** to the toolset specification. See the**CMAKE_GENERATOR_TOOLSET** variable for details.

Visual Studio 17 2022

New in version 3.21.

Generates Visual Studio 17 (VS 2022) project files.

Project Types

Only Visual C++ and C# projects may be generated (and Fortran with Intel compiler integration). Other types of projects (JavaScript, Powershell, Python, etc.) are not supported.

Instance Selection

VS 2022 supports multiple installations on the same machine. The **CMAKE_GENERATOR_INSTANCE** variable may be set as a cache entry containing the absolute path to a Visual Studio instance. If the value is not specified explicitly by the user or a toolchain file, CMake queries the Visual Studio Installer to locate VS instances, chooses one, and sets the variable as a cache entry to hold the value persistently.

When CMake first chooses an instance, if the **VS170COMNTOOLS** environment variable is set and points to the **Common7/Tools** directory within one of the instances, that instance will be used. Otherwise, if more than one instance is installed we do not define which one is chosen by default.

Platform Selection

The default target platform name (architecture) is that of the host and is provided in the **CMAKE_VS_PLATFORM_NAME_DEFAULT** variable.

The **CMAKE_GENERATOR_PLATFORM** variable may be set, perhaps via the **cmake(1) −A** option, to specify a target platform name (architecture). For example:

- **cmake −G "Visual Studio 17 2022" −A Win32**
- **cmake −G "Visual Studio 17 2022" −A x64**
- **cmake −G "Visual Studio 17 2022" −A ARM**
- **cmake −G "Visual Studio 17 2022" −A ARM64**

Toolset Selection

The **v143** toolset that comes with VS 17 2022 is selected by default. The**CMAKE_GENERATOR_TOOLSET** option may be set, perhaps via the **cmake(1) −T** option, to specify another toolset.

For each toolset that comes with this version of Visual Studio, there are variants that are themselves compiled for 32−bit (**x86**) and 64−bit (**x64**) hosts (independent of the architecture they target). By default this generator uses the 64−bit variant on x64 hosts and the 32−bit variant otherwise. One may explicitly request use of either the 32−bit or 64−bit host tools by adding either **host=x86** or **host=x64** to the toolset specification. See the**CMAKE_GENERATOR_TOOLSET** variable for details.

Other Generators

Green Hills MULTI

New in version 3.3.

New in version 3.15: Linux support.

Generates Green Hills MULTI project files (experimental, work−in−progress).

Customizations are available through the following cache variables:

- **GHS_CUSTOMIZATION**
- **GHS_GPJ_MACROS**

New in version 3.14: The buildsystem has predetermined build−configuration settings that can be controlled via the **CMAKE_BUILD_TYPE** variable.

Toolset and Platform Selection
New in version 3.13.

Customizations that are used to pick toolset and target system:

- The **−A <arch>** can be supplied for setting the target architecture. **<ar ch>** usually is one of **arm**, **ppc**, **86**, etcetera. If the target architecture is not specified then the default architecture of **arm** will be used.
- The **−T <toolset>** option can be used to set the directory location of the toolset. Both absolute and relative paths are valid. Relative paths use **GHS_TOOLSET_ROOT** as the root. If the toolset is not specified then the latest toolset found in **GHS_TOOLSET_ROOT** will be used.

Cache variables that are used for toolset and target system customization:

- **GHS_TARGET_PLATFORM**
  Defaults to **integrity**.
  Usual values are **integrity**, **threadx**, **uvelosity**, **velosity**,
  vxworks, **standalone**.

- **GHS_PRIMARY_TARGET**
  Sets **primaryTarget** entry in project file.
  Defaults to **<arch>_<GHS_TARGET_PLATFORM>.tgt**.

- **GHS_TOOLSET_ROOT**
  Root path for **toolset** searches.
  Defaults to **C:/ghs** in Windows or **/usr/ghs** in Linux.

- **GHS_OS_ROOT**
  Root path for RTOS searches.
  Defaults to **C:/ghs** in Windows or **/usr/ghs** in Linux.

- **GHS_OS_DIR** and **GHS_OS_DIR_OPTION**
  Sets **−os_dir** entry in project file.
  Defaults to latest platform OS installation at **GHS_OS_ROOT**. Set this value if a specific RTOS is to be used.
  GHS_OS_DIR_OPTION default value is **−os_dir**.

  New in version 3.15: The **GHS_OS_DIR_OPTION** variable.

- **GHS_BSP_NAME**
  Sets **−bsp** entry in project file.
  Defaults to **sim<arch>** for **integrity** platforms.

Target Properties
New in version 3.14.

The following properties are available:

- **GHS_INTEGRITY_APP**
- **GHS_NO_SOURCE_GROUP_FILE**

NOTE:
This generator is deemed experimental as of CMake 3.22.1 and is still a work in progress. Future versions of CMake may make breaking changes as the generator matures.

Xcode
Generate Xcode project files.

Changed in version 3.15: This generator supports Xcode 5.0 and above.

Toolset and Build System Selection
By default Xcode is allowed to select its own default toolchain. The **CMAKE_GENERATOR_TOOLSET** option may be set, perhaps via the **cmake(1) −T** option, to specify another toolset.

New in version 3.19: This generator supports toolset specification using one of these forms:

- **toolset**
- **toolset[,key=value]***
- **key=value[,key=value]***

The **toolset** specifies the toolset name. The selected toolset name is provided in the **CMAKE_XCODE_PLATFORM_TOOLSET** variable.

The **key=value** pairs form a comma−separated list of options to specify generator−specific details of the toolset selection. Supported pairs are:

buildsystem=<variant>
Specify the buildsystem variant to use. See the **CMAKE_XCODE_BUILD_SYSTEM** variable for allowed values.

For example, to select the original build system under Xcode 12, run **cmake(1)** with the option **−T buildsystem=1**.

Swift Support
New in version 3.4.

When using the *Xcode* generator with Xcode 6.1 or higher, one may enable the **Swift** language with the **enable_language()** command or the **project()**.

EXTRA GENERATORS

Some of the *CMake Generators* listed in the **cmake(1)** command−line tool **−−help** output may have variants that specify an extra generator for an auxiliary IDE tool. Such generator names have the form **<extra−generator> − <main−generator>**. The following extra generators are known to CMake.

CodeBlocks
Generates CodeBlocks project files.

Project files for CodeBlocks will be created in the top directory and in every subdirectory which features a **CMakeLists.txt** file containing a **project()** call. Additionally a hierarchy of makefiles is generated into the build tree. The appropriate make program can build the project through the default **all** target. An**install** target is also provided.

New in version 3.10: The **CMAKE_CODEBLOCKS_EXCLUDE_EXTERNAL_FILES** variable may be set to **ON** to exclude any files which are located outside of the project root directory.

This "extra" generator may be specified as:

CodeBlocks − MinGW Makefiles
Generate with **MinGW Makefiles**.

CodeBlocks − NMake Makefiles
Generate with **NMake Makefiles**.

CodeBlocks − NMake Makefiles JOM
New in version 3.8: Generate with **NMake Makefiles JOM**.

CodeBlocks − Ninja
Generate with **Ninja**.

CodeBlocks − Unix Makefiles
Generate with **Unix Makefiles**.

CodeLite
Generates CodeLite project files.

Project files for CodeLite will be created in the top directory and in every subdirectory which features a CMakeLists.txt file containing a **project()** call. The appropriate make program can build the project through the default **all** target. An**install** tar get is also provided.

New in version 3.7: The **CMAKE_CODELITE_USE_TARGETS** variable may be set to **ON** to change the default behavior from projects to targets as the basis for project files.

This "extra" generator may be specified as:

CodeLite − MinGW Makefiles
Generate with **MinGW Makefiles**.

CodeLite − NMake Makefiles
Generate with **NMake Makefiles**.

CodeLite − Ninja
Generate with **Ninja**.

CodeLite − Unix Makefiles
Generate with **Unix Makefiles**.

Eclipse CDT4
Generates Eclipse CDT 4.0 project files.

Project files for Eclipse will be created in the top directory. In out of source builds, a linked resource to the top level source directory will be created. Additionally a hierarchy of makefiles is generated into the build tree. The appropriate make program can build the project through the default **all** target. An**install** tar get is also provided.

This "extra" generator may be specified as:

Eclipse CDT4 − MinGW Makefiles
Generate with **MinGW Makefiles**.

Eclipse CDT4 − NMake Makefiles
Generate with **NMake Makefiles**.

Eclipse CDT4 − Ninja
Generate with **Ninja**.

Eclipse CDT4 − Unix Makefiles
Generate with **Unix Makefiles**.

Kate
Generates Kate project files.

A project file for Kate will be created in the top directory in the top level build directory. To use it in Kate, the Project plugin must be enabled. The project file is loaded in Kate by opening the **ProjectName.kateproject** file in the editor. If the Kate Build−plugin is enabled, all targets generated by CMake are available for building.

This "extra" generator may be specified as:

Kate − MinGW Makefiles
Generate with **MinGW Makefiles**.

Kate − NMake Makefiles
Generate with **NMake Makefiles**.

Kate − Ninja
Generate with **Ninja**.

Kate − Unix Makefiles
Generate with **Unix Makefiles**.

Sublime Text 2
Generates Sublime Text 2 project files.

Project files for Sublime Text 2 will be created in the top directory and in every subdirectory which features a **CMakeLists.txt** file containing a **project()** call. Additionally **Makefiles** (or **build.ninja** files) are generated into the build tree. The appropriate make program can build the project through the default **all** target. An **install** target is also provided.

This "extra" generator may be specified as:

Sublime Text 2 − MinGW Makefiles
Generate with **MinGW Makefiles**.

Sublime Text 2 − NMake Makefiles
Generate with **NMake Makefiles**.

Sublime Text 2 − Ninja
Generate with **Ninja**.

Sublime Text 2 − Unix Makefiles
Generate with **Unix Makefiles**.

COPYRIGHT