NAME

cmake-properties − CMake Properties Reference

PROPERTIES OF GLOBAL SCOPE

ALLOW_DUPLICATE_CUSTOM_TARGETS

Allow duplicate custom targets to be created.

Normally CMake requires that all targets built in a project have globally unique logical names (see policy **CMP0002**). This is necessary to generate meaningful project file names in**Xcode** and V isual Studio Generators IDE generators. It also allows the target names to be referenced unambiguously.

Makefile generators are capable of supporting duplicate **add_custom_target()** names. For projects that care only about Makefile Generators and do not wish to support **Xcode** or Visual Studio Generators IDE generators, one may set this property to **True** to allow duplicate custom targets. The property allows multiple **add_custom_target()** command calls in different directories to specify the same target name. However, setting this property will cause non−Makefile generators to produce an error and refuse to generate the project.

AUTOGEN_SOURCE_GROUP

New in version 3.9.

Name of the **source_group()** for **AUTOMOC**, **AUTORCC** and **AUTOUIC** generated files.

Files generated by **AUTOMOC**, **AUTORCC** and **AUTOUIC** are not always known at configure time and therefore can't be passed to **source_group()**. *AUTOGEN_SOURCE_GROUP* can be used instead to generate or select a source group for **AUTOMOC**, **AUTORCC** and **AUTOUIC** generated files.

For **AUTOMOC**, **AUTORCC** and **AUTOUIC** specific overrides see **AUTOMOC_SOURCE_GROUP**, **AUTORCC_SOURCE_GROUP** and **AUTOUIC_SOURCE_GROUP** respectively.

AUTOGEN_TARGETS_FOLDER

Name of **FOLDER** for ***_autogen** targets that are added automatically by CMake for targets for which **AUTOMOC** is enabled.

If not set, CMake uses the **FOLDER** property of the parent target as a default value for this property. See also the documentation for the **FOLDER** target property and the **AUTOMOC** target property.

AUTOMOC_SOURCE_GROUP

New in version 3.9.

Name of the **source_group()** for **AUTOMOC** generated files.

When set this is used instead of **AUTOGEN_SOURCE_GROUP** for files generated by **AUTOMOC**.

AUTOMOC_TARGETS_FOLDER

Name of **FOLDER** for ***_autogen** targets that are added automatically by CMake for targets for which **AUTOMOC** is enabled.

This property is obsolete. Use **AUTOGEN_TARGETS_FOLDER** instead.

If not set, CMake uses the **FOLDER** property of the parent target as a default value for this property. See also the documentation for the **FOLDER** target property and the **AUTOMOC** target property.

AUTORCC_SOURCE_GROUP

New in version 3.9.

Name of the **source_group()** for **AUTORCC** generated files.

When set this is used instead of **AUTOGEN_SOURCE_GROUP** for files generated by **AUTORCC**.

AUTOUIC_SOURCE_GROUP
New in version 3.21.

Name of the **source_group()** for **AUTOUIC** generated files.

When set this is used instead of **AUTOGEN_SOURCE_GROUP** for files generated by **AUTOUIC**.

CMAKE_C_KNOWN_FEATURES
New in version 3.1.

List of C features known to this version of CMake.

The features listed in this global property may be known to be available to the C compiler. If the feature is available with the C compiler, it will be listed in the **CMAKE_C_COMPILE_FEATURES** variable.

The features listed here may be used with the **target_compile_features()** command. See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

The features known to this version of CMake are listed below.

High level meta features indicating C standard support
New in version 3.8.

c_std_90
Compiler mode is at least C 90.

c_std_99
Compiler mode is at least C 99.

c_std_11
Compiler mode is at least C 11.

c_std_17
New in version 3.21.

Compiler mode is at least C 17.

c_std_23
New in version 3.21.

Compiler mode is at least C 23.

Low level individual compile features

c_function_prototypes
Function prototypes, as defined in **ISO/IEC 9899:1990**.

c_restrict
restrict keyword, as defined in **ISO/IEC 9899:1999**.

c_static_assert
Static assert, as defined in **ISO/IEC 9899:2011**.

c_variadic_macros
Variadic macros, as defined in **ISO/IEC 9899:1999**.

CMAKE_CUDA_KNOWN_FEATURES
New in version 3.17.

List of CUDA features known to this version of CMake.

The features listed in this global property may be known to be available to the CUDA compiler. If the feature is available with the C++ compiler, it will be listed in the **CMAKE_CUDA_COMPILE_FEATURES** variable.

The features listed here may be used with the **target_compile_features()** command. See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

The features known to this version of CMake are:

cuda_std_03
Compiler mode is at least CUDA/C++ 03.

cuda_std_11
Compiler mode is at least CUDA/C++ 11.

cuda_std_14
Compiler mode is at least CUDA/C++ 14.

cuda_std_17
Compiler mode is at least CUDA/C++ 17.

cuda_std_20
Compiler mode is at least CUDA/C++ 20.

cuda_std_23
New in version 3.20.

Compiler mode is at least CUDA/C++ 23.

CMAKE_CXX_KNOWN_FEATURES
New in version 3.1.

List of C++ features known to this version of CMake.

The features listed in this global property may be known to be available to the C++ compiler. If the feature is available with the C++ compiler, it will be listed in the **CMAKE_CXX_COMPILE_FEATURES** variable.

The features listed here may be used with the **target_compile_features()** command. See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

The features known to this version of CMake are listed below.

High level meta features indicating C++ standard support
New in version 3.8.

The following meta features indicate general support for the associated language standard. It reflects the language support claimed by the compiler, but it does not necessarily imply complete conformance to that

standard.

cxx_std_98
Compiler mode is at least C++ 98.

cxx_std_11
Compiler mode is at least C++ 11.

cxx_std_14
Compiler mode is at least C++ 14.

cxx_std_17
Compiler mode is at least C++ 17.

cxx_std_20
New in version 3.12.

Compiler mode is at least C++ 20.

cxx_std_23
New in version 3.20.

Compiler mode is at least C++ 23.

Low level individual compile features
For C++ 11 and C++ 14, compilers were sometimes slow to implement certain language features. CMake provided some individual compile features to help projects determine whether specific features were available. These individual features are now less relevant and projects should generally prefer to use the high level meta features instead. Individual compile features are not provided for C++ 17 or later.

See the **cmake−compile−features(7)** manual for further discussion of the use of individual compile features.

Individual features from C++ 98
cxx_template_template_parameters
Template template parameters, as defined in **ISO/IEC 14882:1998**.

Individual features from C++ 11
cxx_alias_templates
Template aliases, as defined in *N2258*.

cxx_alignas
Alignment control **alignas**, as defined in *N2341*.

cxx_alignof
Alignment control **alignof**, as defined in *N2341*.

cxx_attributes
Generic attributes, as defined in *N2761*.

cxx_auto_type
Automatic type deduction, as defined in *N1984*.

cxx_constexpr
Constant expressions, as defined in *N2235*.

cxx_decltype_incomplete_return_types
Decltype on incomplete return types, as defined in *N3276*.

cxx_decltype
Decltype, as defined in *N2343*.

cxx_default_function_template_args
Default template arguments for function templates, as defined in *DR226*

cxx_defaulted_functions
Defaulted functions, as defined in *N2346*.

cxx_defaulted_move_initializers
Defaulted move initializers, as defined in *N3053*.

cxx_delegating_constructors
Delegating constructors, as defined in *N1986*.

cxx_deleted_functions
Deleted functions, as defined in *N2346*.

cxx_enum_forward_declarations
Enum forward declarations, as defined in *N2764*.

cxx_explicit_conversions
Explicit conversion operators, as defined in *N2437*.

cxx_extended_friend_declarations
Extended friend declarations, as defined in *N1791*.

cxx_extern_templates
Extern templates, as defined in *N1987*.

cxx_final
Override control **final** keyword, as defined in *N2928*, *N3206* and *N3272*.

cxx_func_identifier
Predefined **__func__** identifier, as defined in *N2340*.

cxx_generalized_initializers
Initializer lists, as defined in *N2672*.

cxx_inheriting_constructors
Inheriting constructors, as defined in *N2540*.

cxx_inline_namespaces
Inline namespaces, as defined in *N2535*.

cxx_lambdas
Lambda functions, as defined in *N2927*.

cxx_local_type_template_args
Local and unnamed types as template arguments, as defined in *N2657*.

cxx_long_long_type
long long type, as defined in *N1811*.

cxx_noexcept
Exception specifications, as defined in *N3050*.

cxx_nonstatic_member_init
Non−static data member initialization, as defined in *N2756*.

cxx_nullptr
Null pointer, as defined in *N2431*.

cxx_override
Override control **override** keyword, as defined in *N2928*, *N3206* and *N3272*.

cxx_range_for
Range−based for, as defined in *N2930*.

cxx_raw_string_literals
Raw string literals, as defined in *N2442*.

cxx_reference_qualified_functions
Reference qualified functions, as defined in *N2439*.

cxx_right_angle_brackets
Right angle bracket parsing, as defined in *N1757*.

cxx_rvalue_references
R−value references, as defined in *N2118*.

cxx_sizeof_member
Size of non−static data members, as defined in *N2253*.

cxx_static_assert
Static assert, as defined in *N1720*.

cxx_strong_enums
Strongly typed enums, as defined in *N2347*.

cxx_thread_local
Thread−local variables, as defined in *N2659*.

cxx_trailing_return_types
Automatic function return type, as defined in *N2541*.

cxx_unicode_literals
Unicode string literals, as defined in *N2442*.

cxx_uniform_initialization
Uniform initialization, as defined in *N2640*.

cxx_unrestricted_unions
Unrestricted unions, as defined in *N2544*.

cxx_user_literals
User−defined literals, as defined in *N2765*.

cxx_variadic_macros
Variadic macros, as defined in *N1653*.

cxx_variadic_templates
Variadic templates, as defined in *N2242*.

Individual features from C++ 14

cxx_aggregate_default_initializers
Aggregate default initializers, as defined in *N3605*.

cxx_attribute_deprecated
[[deprecated]] attribute, as defined in *N3760*.

cxx_binary_literals
Binary literals, as defined in *N3472*.

cxx_contextual_conversions
Contextual conversions, as defined in *N3323*.

cxx_decltype_auto
decltype(auto) semantics, as defined in *N3638*.

cxx_digit_separators
Digit separators, as defined in *N3781*.

cxx_generic_lambdas
Generic lambdas, as defined in *N3649*.

cxx_lambda_init_captures
Initialized lambda captures, as defined in *N3648*.

cxx_relaxed_constexpr
Relaxed constexpr, as defined in *N3652*.

cxx_return_type_deduction
Return type deduction on normal functions, as defined in *N3386*.

cxx_variable_templates
Variable templates, as defined in *N3651*.

CMAKE_ROLE
New in version 3.14.

Tells what mode the current running script is in. Could be one of several values:

PROJECT
Running in project mode (processing a **CMakeLists.txt** file).

SCRIPT
Running in **-P** script mode.

FIND_PACKAGE
Running in **--find-package** mode.

CTEST
Running in CTest script mode.

CPACK
Running in CPack.

DEBUG_CONFIGURATIONS
Specify which configurations are for debugging.

The value must be a semi-colon separated list of configuration names. Currently this property is used only by the **target_link_libraries()** command. Additional uses may be defined in the future.

This property must be set at the top level of the project and before the first **target_link_libraries()** command invocation. If any entry in the list does not match a valid configuration for the project the behavior is undefined.

DISABLED_FEATURES
List of features which are disabled during the CMake run.

List of features which are disabled during the CMake run. By default it contains the names of all packages which were not found. This is determined using the **<NAME>_FOUND** variables. Packages which are searched **QUIET** are not listed. A project can add its own features to this list. This property is used by the macros in **FeatureSummary.cmake**.

ECLIPSE_EXTRA_CPROJECT_CONTENTS
New in version 3.12.

Additional contents to be inserted into the generated Eclipse cproject file.

The cproject file defines the CDT specific information. Some third party IDE's are based on Eclipse with the addition of other information specific to that IDE. Through this property, it is possible to add this additional contents to the generated project. It is expected to contain valid XML.

Also see the **ECLIPSE_EXTRA_NATURES** property.

ECLIPSE_EXTRA_NATURES
List of natures to add to the generated Eclipse project file.

Eclipse projects specify language plugins by using natures. This property should be set to the unique identifier for a nature (which looks like a Java package name).

Also see the **ECLIPSE_EXTRA_CPROJECT_CONTENTS** property.

ENABLED_FEATURES
List of features which are enabled during the CMake run.

List of features which are enabled during the CMake run. By default it contains the names of all packages which were found. This is determined using the **<NAME>_FOUND** variables. Packages which are searched **QUIET** are not listed. A project can add its own features to this list. This property is used by the macros in **FeatureSummary.cmake**.

ENABLED_LANGUAGES
Read−only property that contains the list of currently enabled languages

Set to list of currently enabled languages.

FIND_LIBRARY_USE_LIB32_PATHS
New in version 3.7.

Whether the **find_library()** command should automatically search **lib32** directories.

FIND_LIBRARY_USE_LIB32_PATHS is a boolean specifying whether the **find_library()** command should automatically search the **lib32** variant of directories called **lib** in the search path when building 32−bit binaries.

See also the **CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX** variable.

FIND_LIBRARY_USE_LIB64_PATHS
Whether **find_library()** should automatically search lib64 directories.

FIND_LIBRARY_USE_LIB64_PATHS is a boolean specifying whether the **find_library()** command should automatically search the lib64 variant of directories called lib in the search path when building 64−bit binaries.

See also the **CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX** variable.

FIND_LIBRARY_USE_LIBX32_PATHS
New in version 3.9.

Whether the **find_library()** command should automatically search **libx32** directories.

FIND_LIBRARY_USE_LIBX32_PATHS is a boolean specifying whether the **find_library()** command should automatically search the **libx32** variant of directories called **lib** in the search path when building x32−abi binaries.

See also the **CMAKE_FIND_LIBRARY_CUSTOM_LIB_SUFFIX** variable.

FIND_LIBRARY_USE_OPENBSD_VERSIONING
Whether **find_library()** should find OpenBSD−style shared libraries.

This property is a boolean specifying whether the **find_library()** command should find shared libraries

with OpenBSD−style versioned extension: ".so.<major>.<minor>". The property is set to true on OpenBSD and false on other platforms.

GENERATOR_IS_MULTI_CONFIG
New in version 3.9.

Read−only property that is true on multi−configuration generators.

True when using a multi−configuration generator such as:

- **Ninja Multi−Config**
- Visual Studio Generators
- **Xcode**

Multi−config generators use **CMAKE_CONFIGURATION_TYPES** as the set of configurations and ignore **CMAKE_BUILD_TYPE**.

GLOBAL_DEPENDS_DEBUG_MODE
Enable global target dependency graph debug mode.

CMake automatically analyzes the global inter−target dependency graph at the beginning of native build system generation. This property causes it to display details of its analysis to stderr.

GLOBAL_DEPENDS_NO_CYCLES
Disallow global target dependency graph cycles.

CMake automatically analyzes the global inter−target dependency graph at the beginning of native build system generation. It reports an error if the dependency graph contains a cycle that does not consist of all STATIC library targets. This property tells CMake to disallow all cycles completely, even among static libraries.

IN_TRY_COMPILE
Read−only property that is true during a try−compile configuration.

True when building a project inside a **try_compile()** or **try_run()** command.

JOB_POOLS
Ninja only: List of available pools.

A pool is a named integer property and defines the maximum number of concurrent jobs which can be started by a rule assigned to the pool. The *JOB_POOLS* property is a semicolon−separated list of pairs using the syntax NAME=integer (without a space after the equality sign).

For instance:

```
set_property(GLOBAL PROPERTY JOB_POOLS two_jobs=2 ten_jobs=10)
```

Defined pools could be used globally by setting **CMAKE_JOB_POOL_COMPILE** and **CMAKE_JOB_POOL_LINK** or per target by setting the target properties **JOB_POOL_COMPILE** and **JOB_POOL_LINK**. **Custom commands** and **custom targets** can specify pools using the option **JOB_POOL**. Using a pool that is not defined by*JOB_POOLS* causes an error by ninja at b uild time.

If not set, this property uses the value of the **CMAKE_JOB_POOLS** variable.

Build targets provided by CMake that are meant for individual interactive use, such as **install**, are placed in the **console** pool automatically.

PACKAGES_FOUND
List of packages which were found during the CMake run.

List of packages which were found during the CMake run. Whether a package has been found is determined using the <NAME>_FOUND variables.

PACKAGES_NOT_FOUND
List of packages which were not found during the CMake run.

List of packages which were not found during the CMake run. Whether a package has been found is determined using the <NAME>_FOUND variables.

PREDEFINED_TARGETS_FOLDER
Name of FOLDER for targets that are added automatically by CMake.

If not set, CMake uses "CMakePredefinedTargets" as a default value for this property. Targets such as INSTALL, PACKAGE and RUN_TESTS will be organized into this FOLDER. See also the documentation for the **FOLDER** target property.

REPORT_UNDEFINED_PROPERTIES
If set, report any undefined properties to this file.

If this property is set to a filename then when CMake runs it will report any properties or variables that were accessed but not defined into the filename specified in this property.

RULE_LAUNCH_COMPILE
Specify a launcher for compile rules.

Makefile Generators and the **Ninja** generator prefix compiler commands with the given launcher command line. This is intended to allow launchers to intercept build problems with high granularity. Other generators ignore this property because their underlying build systems provide no hook to wrap individual commands with a launcher.

RULE_LAUNCH_CUSTOM
Specify a launcher for custom rules.

Makefile Generators and the **Ninja** generator prefix custom commands with the given launcher command line. This is intended to allow launchers to intercept build problems with high granularity. Other generators ignore this property because their underlying build systems provide no hook to wrap individual commands with a launcher.

RULE_LAUNCH_LINK
Specify a launcher for link rules.

Makefile Generators and the **Ninja** generator prefix link and archive commands with the given launcher command line. This is intended to allow launchers to intercept build problems with high granularity. Other generators ignore this property because their underlying build systems provide no hook to wrap individual commands with a launcher.

RULE_MESSAGES
Specify whether to report a message for each make rule.

This property specifies whether Makefile generators should add a progress message describing what each build rule does. If the property is not set the default is ON. Set the property to OFF to disable granular messages and report only as each target completes. This is intended to allow scripted builds to avoid the build time cost of detailed reports. If a **CMAKE_RULE_MESSAGES** cache entry exists its value initializes the value of this property. Non−Makefile generators currently ignore this property.

TARGET_ARCHIVES_MAY_BE_SHARED_LIBS
Set if shared libraries may be named like archives.

On AIX shared libraries may be named "lib<name>.a". This property is set to true on such platforms.

TARGET_MESSAGES
New in version 3.4.

Specify whether to report the completion of each target.

This property specifies whether Makefile Generators should add a progress message describing that each target has been completed. If the property is not set the default is **ON**. Set the property to**OFF** to disable target completion messages.

This option is intended to reduce build output when little or no work needs to be done to bring the build tree up to date.

If a **CMAKE_TARGET_MESSAGES** cache entry exists its value initializes the value of this property.

Non−Makefile generators currently ignore this property.

See the counterpart property **RULE_MESSAGES** to disable everything except for target completion messages.

TARGET_SUPPORTS_SHARED_LIBS
Does the target platform support shared libraries.

TARGET_SUPPORTS_SHARED_LIBS is a boolean specifying whether the target platform supports shared libraries. Basically all current general general purpose OS do so, the exception are usually embedded systems with no or special OSs.

USE_FOLDERS
Use the **FOLDER** target property to organize targets into folders.

If not set, CMake treats this property as **OFF** by default. CMake generators that are capable of organizing into a hierarchy of folders use the values of the **FOLDER** target property to name those folders. See also the documentation for the **FOLDER** target property.

XCODE_EMIT_EFFECTIVE_PLATFORM_NAME
New in version 3.8.

Control emission of **EFFECTIVE_PLATFORM_NAME** by the **Xcode** generator.

It is required for building the same target with multiple SDKs. A common use case is the parallel use of **iphoneos** and **iphonesimulator** SDKs.

Three different states possible that control when the **Xcode** generator emits the **EFFECTIVE_PLATFORM_NAME** variable:

- If set to **ON** it will always be emitted
- If set to **OFF** it will never be emitted
- If unset (the default) it will only be emitted when the project was configured for an embedded Xcode SDK like iOS, tvOS, watchOS or any of the simulators.

NOTE:

When this behavior is enable for generated Xcode projects, the **EFFECTIVE_PLATFORM_NAME** variable will leak into **Generator expressions** like **TARGET_FILE** and will render those mostly unusable.

PROPERTIES ON DIRECTORIES

ADDITIONAL_CLEAN_FILES

New in version 3.15.

A ;−list of files or directories that will be removed as a part of the global **clean** target. It is useful for specifying generated files or directories that are used by multiple targets or by CMake itself, or that are generated in ways which cannot be captured as outputs or byproducts of custom commands.

If an additional clean file is specific to a single target only, then the **ADDITIONAL_CLEAN_FILES** target property would usually be a better choice than this directory property.

Relative paths are allowed and are interpreted relative to the current binary directory.

Contents of **ADDITIONAL_CLEAN_FILES** may use **generator expressions**.

This property only works for the **Ninja** and the Makefile generators. It is ignored by other generators.

BINARY_DIR

New in version 3.7.

This read−only directory property reports absolute path to the binary directory corresponding to the source on which it is read.

BUILDSYSTEM_TARGETS

New in version 3.7.

This read−only directory property contains a semicolon−separated list of buildsystem targets added in the directory by calls to the **add_library()**, **add_executable()**, and **add_custom_target()** commands. The list does not include any Imported Targets or Alias Targets, but does include Interface Libraries. Each entry in the list is the logical name of a target, suitable to pass to the **get_property()** command **TARGET** option.

See also the **IMPORTED_TARGETS** directory property.

CACHE_VARIABLES

List of cache variables available in the current directory.

This read−only property specifies the list of CMake cache variables currently defined. It is intended for debugging purposes.

CLEAN_NO_CUSTOM

Set to true to tell Makefile Generators not to remove the outputs of custom commands for this directory during the **make clean** operation. This is ignored on other generators because it is not possible to implement.

CMAKE_CONFIGURE_DEPENDS

Tell CMake about additional input files to the configuration process. If any named file is modified the build system will re−run CMake to re−configure the file and generate the build system again.

Specify files as a semicolon−separated list of paths. Relative paths are interpreted as relative to the current source directory.

COMPILE_DEFINITIONS
Preprocessor definitions for compiling a directory's sources.

This property specifies the list of options given so far to the **add_compile_definitions()** (or **add_definitions()**) command.

The **COMPILE_DEFINITIONS** property may be set to a semicolon−separated list of preprocessor definitions using the syntax **VAR** or **VAR=value**. Function−style definitions are not supported. CMake will automatically escape the value correctly for the native build system (note that CMake language syntax may require escapes to specify some values).

This property will be initialized in each directory by its value in the directory's parent.

CMake will automatically drop some definitions that are not supported by the native build tool.

Disclaimer: Most native build tools have poor support for escaping certain values. CMake has work−arounds for many cases but some values may just not be possible to pass correctly. If a value does not seem to be escaped correctly, do not attempt to work−around the problem by adding escape sequences to the value. Your work−around may break in a future version of CMake that has improved escape support. Instead consider defining the macro in a (configured) header file. Then report the limitation. Known limitations include:

```
#           − broken almost everywhere
;           − broken in VS IDE 7.0 and Borland Makefiles
,           − broken in VS IDE
%           − broken in some cases in NMake
& |         − broken in some cases on MinGW
^ < > \"    − broken in most Make tools on Windows
```

CMake does not reject these values outright because they do work in some cases. Use with caution.

Contents of **COMPILE_DEFINITIONS** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. See the**cmak e−buildsystem(7)** manual for more on defining buildsystem properties.

The corresponding **COMPILE_DEFINITIONS_<CONFIG>** property may be set to specify per−configuration definitions. Generator expressions should be preferred instead of setting the alternative property.

COMPILE_OPTIONS
List of options to pass to the compiler.

This property holds a semicolon−separated list of options given so far to the **add_compile_options()** command.

This property is used to initialize the **COMPILE_OPTIONS** target property when a target is created, which is used by the generators to set the options for the compiler.

Contents of **COMPILE_OPTIONS** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. See the**cmak e−buildsystem(7)** manual for more on defining buildsystem properties.

DEFINITIONS
For CMake 2.4 compatibility only. Use**COMPILE_DEFINITIONS** instead.

This read−only property specifies the list of flags given so far to the **add_definitions()** command. It is intended for debugging purposes. Use the **COMPILE_DEFINITIONS** directory property instead.

This built−in read−only property does not exist if policy **CMP0059** is set to **NEW**.

EXCLUDE_FROM_ALL
Set this directory property to a true value on a subdirectory to exclude its targets from the "all" target of its ancestors. If excluded, running e.g. **make** in the parent directory will not build targets the subdirectory by default. This does not affect the "all" target of the subdirectory itself. Running e.g. **make** inside the subdirectory will still build its targets.

If the **EXCLUDE_FROM_ALL** target property is set on a target then its value determines whether the target is included in the "all" target of this directory and its ancestors.

IMPLICIT_DEPENDS_INCLUDE_TRANSFORM
Specify **#include** line transforms for dependencies in a directory.

This property specifies rules to transform macro−like **#include** lines during implicit dependency scanning of C and C++ source files. The list of rules must be semicolon−separated with each entry of the form **A_MACRO(%)=value−with−%** (the **%** must be literal). During dependency scanning occurrences of **A_MACRO(...)** on **#include** lines will be replaced by the value given with the macro argument substituted for **%**. For example, the entry

```
MYDIR(%)=<mydir/%>
```

will convert lines of the form

```
#include MYDIR(myheader.h)
```

to

```
#include <mydir/myheader.h>
```

allowing the dependency to be followed.

This property applies to sources in all targets within a directory. The property value is initialized in each directory by its value in the directory's parent.

IMPORTED_TARGETS
New in version 3.21.

This read−only directory property contains a semicolon−separated list of Imported Targets added in the directory by calls to the **add_library()** and **add_executable()** commands. Each entry in the list is the logical name of a target, suitable to pass to the **get_property()** command **TARGET** option when called in the same directory.

See also the **BUILDSYSTEM_TARGETS** directory property.

INCLUDE_DIRECTORIES
List of preprocessor include file search directories.

This property specifies the list of directories given so far to the **include_directories()** command.

This property is used to populate the **INCLUDE_DIRECTORIES** target property, which is used by the generators to set the include directories for the compiler.

In addition to accepting values from that command, values may be set directly on any directory using the **set_property()** command, and can be set on the current directory using the **set_directory_properties()** command. A directory gets its initial value from its parent directory if it has one. The initial value of the

INCLUDE_DIRECTORIES target property comes from the value of this property. Both directory and target property values are adjusted by calls to the **include_directories()** command. Calls to **set_property()** or **set_directory_properties()**, however, will update the directory property value without updating target property values. Therefore direct property updates must be made before calls to**add_executable()** or **add_library()** for targets they are meant to affect.

The target property values are used by the generators to set the include paths for the compiler.

Contents of **INCLUDE_DIRECTORIES** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. See the**cmak e−buildsystem(7)** manual for more on defining buildsystem properties.

INCLUDE_REGULAR_EXPRESSION
Include file scanning regular expression.

This property specifies the regular expression used during dependency scanning to match include files that should be followed. See the**include_r egular_expression()** command for a high−level interface to set this property.

INTERPROCEDURAL_OPTIMIZATION
Enable interprocedural optimization for targets in a directory.

If set to true, enables interprocedural optimizations if they are known to be supported by the compiler.

INTERPROCEDURAL_OPTIMIZATION_<CONFIG>
Per−configuration interprocedural optimization for a directory.

This is a per−configuration version of **INTERPROCEDURAL_OPTIMIZATION**. If set, this property overrides the generic property for the named configuration.

LABELS
New in version 3.10.

Specify a list of text labels associated with a directory and all of its subdirectories. This is equivalent to setting the **LABELS** target property and the **LABELS** test property on all targets and tests in the current directory and subdirectories. Note: Launchers must enabled to propagate labels to targets.

The **CMAKE_DIRECTORY_LABELS** variable can be used to initialize this property.

The list is reported in dashboard submissions.

LINK_DIRECTORIES
List of linker search directories.

This property holds a semicolon−separated list of directories and is typically populated using the **link_directories()** command. It gets its initial value from its parent directory, if it has one.

The directory property is used to initialize the **LINK_DIRECTORIES** target property when a target is created. That target property is used by the generators to set the library search directories for the linker.

Contents of **LINK_DIRECTORIES** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. See the**cmak e−buildsystem(7)** manual for more on defining buildsystem properties.

LINK_OPTIONS
New in version 3.13.

List of options to use for the link step of shared library, module and executable targets as well as the device link step.

This property holds a semicolon−separated list of options given so far to the **add_link_options()** command.

This property is used to initialize the **LINK_OPTIONS** target property when a target is created, which is used by the generators to set the options for the compiler.

Contents of **LINK_OPTIONS** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. See the**cmak e−buildsystem(7)** manual for more on defining buildsystem properties.

LISTFILE_STACK
The current stack of listfiles being processed.

This property is mainly useful when trying to debug errors in your CMake scripts. It returns a list of what list files are currently being processed, in order. So if one listfile does an**include()** command then that is effectively pushing the included listfile onto the stack.

MACROS
List of macro commands available in the current directory.

This read−only property specifies the list of CMake macros currently defined. It is intended for debugging purposes. See the**macr o()** command.

PARENT_DIRECTORY
Source directory that added current subdirectory.

This read−only property specifies the source directory that added the current source directory as a subdirectory of the build. In the top−level directory the value is the empty−string.

RULE_LAUNCH_COMPILE
Specify a launcher for compile rules.

See the global property of the same name for details. This overrides the global property for a directory.

RULE_LAUNCH_CUSTOM
Specify a launcher for custom rules.

See the global property of the same name for details. This overrides the global property for a directory.

RULE_LAUNCH_LINK
Specify a launcher for link rules.

See the global property of the same name for details. This overrides the global property for a directory.

SOURCE_DIR
New in version 3.7.

This read−only directory property reports absolute path to the source directory on which it is read.

SUBDIRECTORIES
New in version 3.7.

This read−only directory property contains a semicolon−separated list of subdirectories processed so far by the **add_subdirectory()** or **subdirs()** commands. Each entry is the absolute path to the source directory (containing the **CMakeLists.txt** file). This is suitable to pass to the **get_property()** command

DIRECTORY option.

NOTE:
The **subdirs()** command does not process its arguments until after the calling directory is fully processed. Therefore looking up this property in the current directory will not see them.

TESTS
New in version 3.12.

List of tests.

This read−only property holds a semicolon−separated list of tests defined so far, in the current directory, by the **add_test()** command.

TEST_INCLUDE_FILES
New in version 3.10.

A list of cmake files that will be included when ctest is run.

If you specify **TEST_INCLUDE_FILES**, those files will be included and processed when ctest is run on the directory.

VARIABLES
List of variables defined in the current directory.

This read−only property specifies the list of CMake variables currently defined. It is intended for debugging purposes.

VS_GLOBAL_SECTION_POST_<section>
Specify a postSolution global section in Visual Studio.

Setting a property like this generates an entry of the following form in the solution file:

```
GlobalSection(<section>) = postSolution
  <contents based on property value>
EndGlobalSection
```

The property must be set to a semicolon−separated list of **key=value** pairs. Each such pair will be transformed into an entry in the solution global section. Whitespace around key and value is ignored. List elements which do not contain an equal sign are skipped.

This property only works for Visual Studio 9 and above; it is ignored on other generators. The property only applies when set on a directory whose **CMakeLists.txt** contains a **project()** command.

Note that CMake generates postSolution sections **ExtensibilityGlobals** and **ExtensibilityAddIns** by default. If you set the corresponding property, it will override the default section. For example, setting **VS_GLOBAL_SECTION_POST_ExtensibilityGlobals** will override the default contents of the **ExtensibilityGlobals** section, while keeping ExtensibilityAddIns on its default. However, CMake will always add a **SolutionGuid** to the **ExtensibilityGlobals** section if it is not specified explicitly.

VS_GLOBAL_SECTION_PRE_<section>
Specify a preSolution global section in Visual Studio.

Setting a property like this generates an entry of the following form in the solution file:

```
GlobalSection(<section>) = preSolution
  <contents based on property value>
EndGlobalSection
```

The property must be set to a semicolon−separated list of **key=value** pairs. Each such pair will be transformed into an entry in the solution global section. Whitespace around key and value is ignored. List elements which do not contain an equal sign are skipped.

This property only works for Visual Studio 9 and above; it is ignored on other generators. The property only applies when set on a directory whose **CMakeLists.txt** contains a **project()** command.

VS_STARTUP_PROJECT
New in version 3.6.

Specify the default startup project in a Visual Studio solution.

The Visual Studio Generators create a **.sln** file for each directory whose **CMakeLists.txt** file calls the **project()** command. Set this property in the same directory as a **project()** command call (e.g. in the top−level **CMakeLists.txt** file) to specify the default startup project for the corresponding solution file.

The property must be set to the name of an existing target. This will cause that project to be listed first in the generated solution file causing Visual Studio to make it the startup project if the solution has never been opened before.

If this property is not specified, then the **ALL_BUILD** project will be the default.

PROPERTIES ON TARGETS

ADDITIONAL_CLEAN_FILES
New in version 3.15.

A ;−list of files or directories that will be removed as a part of the global **clean** target. It can be used to specify files and directories that are generated as part of building the target or that are directly associated with the target in some way (e.g. created as a result of running the target).

For custom targets, if such files can be captured as outputs or byproducts instead, then that should be preferred over adding them to this property. If an additional clean file is used by multiple targets or isn't target−specific, then the **ADDITIONAL_CLEAN_FILES** directory property may be the more appropriate property to use.

Relative paths are allowed and are interpreted relative to the current binary directory.

Contents of **ADDITIONAL_CLEAN_FILES** may use **generator expressions**.

This property only works for the **Ninja** and the Makefile generators. It is ignored by other generators.

AIX_EXPORT_ALL_SYMBOLS
New in version 3.17.

On AIX, CMake automatically exports all symbols from shared libraries, and from executables with the **ENABLE_EXPORTS** target property set. Explicitly disable this boolean property to suppress the behavior and export no symbols by default. In this case it is expected that the project will use other means to export some symbols.

This property is initialized by the value of the **CMAKE_AIX_EXPORT_ALL_SYMBOLS** variable if it is set when a target is created.

ALIAS_GLOBAL
New in version 3.18.

Read−only property indicating of whether an ALIAS target is globally visible.

The boolean value of this property is **TRUE** for aliases to IMPORTED targets created with the **GLOBAL** options to **add_executable()** or **add_library()**, **FALSE** otherwise. It is undefined for targets built within the project.

NOTE:
Promoting an IMPORTED target from **LOCAL** to **GLOBAL** scope by changing the value or **IMPORTED_GLOBAL** target property do not change the scope of local aliases.

ALIASED_TARGET
Name of target aliased by this target.

If this is an Alias Target, this property contains the name of the target aliased.

ANDROID_ANT_ADDITIONAL_OPTIONS
New in version 3.4.

Set the additional options for Android Ant build system. This is a string value containing all command line options for the Ant build. This property is initialized by the value of the **CMAKE_ANDROID_ANT_ADDITIONAL_OPTIONS** variable if it is set when a target is created.

ANDROID_API
New in version 3.1.

When Cross Compiling for Android with NVIDIA Nsight Tegra Visual Studio Edition, this property sets the Android target API version (e.g. **15**). The version number must be a positive decimal integer. This property is initialized by the value of the **CMAKE_ANDROID_API** variable if it is set when a target is created.

ANDROID_API_MIN
New in version 3.2.

Set the Android MIN API version (e.g. **9**). The version number must be a positive decimal integer. This property is initialized by the value of the **CMAKE_ANDROID_API_MIN** variable if it is set when a target is created. Native code builds using this API version.

ANDROID_ARCH
New in version 3.4.

When Cross Compiling for Android with NVIDIA Nsight Tegra Visual Studio Edition, this property sets the Android target architecture.

This is a string property that could be set to the one of the following values:

- **armv7−a**: "ARMv7−A (armv7−a)"

- **armv7−a−hard**: "ARMv7−A, hard−float ABI (armv7−a)"
- **arm64−v8a**: "ARMv8−A, 64bit (arm64−v8a)"
- **x86**: "x86 (x86)"
- **x86_64**: "x86_64 (x86_64)"

This property is initialized by the value of the **CMAKE_ANDROID_ARCH** variable if it is set when a target is created.

ANDROID_ASSETS_DIRECTORIES
New in version 3.4.

Set the Android assets directories to copy into the main assets folder before build. This a string property that contains the directory paths separated by semicolon. This property is initialized by the value of the **CMAKE_ANDROID_ASSETS_DIRECTORIES** variable if it is set when a target is created.

ANDROID_GUI
New in version 3.1.

When Cross Compiling for Android with NVIDIA Nsight Tegra Visual Studio Edition, this property specifies whether to build an executable as an application package on Android.

When this property is set to true the executable when built for Android will be created as an application package. This property is initialized by the value of the **CMAKE_ANDROID_GUI** variable if it is set when a target is created.

Add the **AndroidManifest.xml** source file explicitly to the target **add_executable()** command invocation to specify the root directory of the application package source.

ANDROID_JAR_DEPENDENCIES
New in version 3.4.

Set the Android property that specifies JAR dependencies. This is a string value property. This property is initialized by the value of the **CMAKE_ANDROID_JAR_DEPENDENCIES** variable if it is set when a target is created.

ANDROID_JAR_DIRECTORIES
New in version 3.4.

Set the Android property that specifies directories to search for the JAR libraries.

This a string property that contains the directory paths separated by semicolons. This property is initialized by the value of the **CMAKE_ANDROID_JAR_DIRECTORIES** variable if it is set when a target is created.

Contents of **ANDROID_JAR_DIRECTORIES** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions.

ANDROID_JAVA_SOURCE_DIR
New in version 3.4.

Set the Android property that defines the Java source code root directories. This a string property that

contains the directory paths separated by semicolon. This property is initialized by the value of the **CMAKE_ANDROID_JAVA_SOURCE_DIR** variable if it is set when a target is created.

ANDROID_NATIVE_LIB_DEPENDENCIES
New in version 3.4.

Set the Android property that specifies the .so dependencies. This is a string property.

This property is initialized by the value of the **CMAKE_ANDROID_NATIVE_LIB_DEPENDENCIES** variable if it is set when a target is created.

Contents of **ANDROID_NATIVE_LIB_DEPENDENCIES** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions.

ANDROID_NATIVE_LIB_DIRECTORIES
New in version 3.4.

Set the Android property that specifies directories to search for the **.so** libraries.

This a string property that contains the directory paths separated by semicolons.

This property is initialized by the value of the **CMAKE_ANDROID_NATIVE_LIB_DIRECTORIES** variable if it is set when a target is created.

Contents of **ANDROID_NATIVE_LIB_DIRECTORIES** may use "generator expressions" with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions.

ANDROID_PROCESS_MAX
New in version 3.4.

Set the Android property that defines the maximum number of a parallel Android NDK compiler processes (e.g. **4**). This property is initialized by the value of the **CMAKE_ANDROID_PROCESS_MAX** variable if it is set when a target is created.

ANDROID_PROGUARD
New in version 3.4.

When this property is set to true that enables the ProGuard tool to shrink, optimize, and obfuscate the code by removing unused code and renaming classes, fields, and methods with semantically obscure names. This property is initialized by the value of the **CMAKE_ANDROID_PROGUARD** variable if it is set when a target is created.

ANDROID_PROGUARD_CONFIG_PATH
New in version 3.4.

Set the Android property that specifies the location of the ProGuard config file. Leave empty to use the default one. This a string property that contains the path to ProGuard config file. This property is initialized by the value of the **CMAKE_ANDROID_PROGUARD_CONFIG_PATH** variable if it is set when a target is created.

ANDROID_SECURE_PROPS_PATH
New in version 3.4.

Set the Android property that states the location of the secure properties file. This is a string property that contains the file path. This property is initialized by the value of the **CMAKE_ANDROID_SECURE_PROPS_PATH** variable if it is set when a target is created.

ANDROID_SKIP_ANT_STEP
New in version 3.4.

Set the Android property that defines whether or not to skip the Ant build step. This is a boolean property initialized by the value of the **CMAKE_ANDROID_SKIP_ANT_STEP** variable if it is set when a target is created.

ANDROID_STL_TYPE
New in version 3.4.

When Cross Compiling for Android with NVIDIA Nsight Tegra Visual Studio Edition, this property specifies the type of STL support for the project. This is a string property that could set to the one of the following values:

none No C++ Support

system Minimal C++ without STL

gabi++_static
GAbi++ Static

gabi++_shared
GAbi++ Shared

gnustl_static
GNU libstdc++ Static

gnustl_shared
GNU libstdc++ Shared

stlport_static
STLport Static

stlport_shared
STLport Shared

This property is initialized by the value of the **CMAKE_ANDROID_STL_TYPE** variable if it is set when a target is created.

ARCHIVE_OUTPUT_DIRECTORY
Output directory in which to build ARCHIVE target files.

This property specifies the directory into which archive target files should be built. The property value may use **generator expressions**. Multi−configuration generators (Visual Studio, **Xcode**, **Ninja Multi−Config**) append a per−configuration subdirectory to the specified directory unless a generator expression is used.

This property is initialized by the value of the **CMAKE_ARCHIVE_OUTPUT_DIRECTORY** variable if it is set when a target is created.

See also the **ARCHIVE_OUTPUT_DIRECTORY_<CONFIG>** target property.

ARCHIVE_OUTPUT_DIRECTORY_<CONFIG>
Per−configuration output directory for ARCHIVE target files.

This is a per−configuration version of the **ARCHIVE_OUTPUT_DIRECTORY** target property, but multi−configuration generators (VS, Xcode) do NOT append a per−configuration subdirectory to the

specified directory. This property is initialized by the value of the **CMAKE_ARCHIVE_OUTPUT_DIRECTORY_<CONFIG>** variable if it is set when a target is created.

Contents of **ARCHIVE_OUTPUT_DIRECTORY_<CONFIG>** may use **generator expressions**.

ARCHIVE_OUTPUT_NAME
Output name for ARCHIVE target files.

This property specifies the base name for archive target files. It overrides **OUTPUT_NAME** and **OUTPUT_NAME_<CONFIG>** properties.

See also the **ARCHIVE_OUTPUT_NAME_<CONFIG>** target property.

ARCHIVE_OUTPUT_NAME_<CONFIG>
Per−configuration output name for ARCHIVE target files.

This is the configuration−specific version of the **ARCHIVE_OUTPUT_NAME** target property.

AUTOGEN_BUILD_DIR
New in version 3.9.

Directory where **AUTOMOC**, **AUTOUIC** and **AUTORCC** generate files for the target.

The directory is created on demand and automatically added to the **ADDITIONAL_CLEAN_FILES** target property.

When unset or empty the directory **<dir>/<target−name>_autogen** is used where **<dir>** is **CMAKE_CURRENT_BINARY_DIR** and **<target−name>** is **NAME**.

By default *AUTOGEN_BUILD_DIR* is unset.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

AUTOGEN_ORIGIN_DEPENDS
New in version 3.14.

Switch for forwarding origin target dependencies to the corresponding **_autogen** target.

Targets which have their **AUTOMOC** or **AUTOUIC** property **ON** have a corresponding **_autogen** target which generates **moc** and **uic** files. As this **_autogen** target is created at generate−time, it is not possible to define dependencies of it using e.g. **add_dependencies()**. Instead the*A UTOGEN_ORIGIN_DEPENDS* target property decides whether the origin target dependencies should be forwarded to the **_autogen** target or not.

By default *AUTOGEN_ORIGIN_DEPENDS* is initialized from **CMAKE_AUTOGEN_ORIGIN_DEPENDS** which is **ON** by default.

In total the dependencies of the **_autogen** target are composed from

- forwarded origin target dependencies (enabled by default via *AUTOGEN_ORIGIN_DEPENDS*)
- additional user defined dependencies from **AUTOGEN_TARGET_DEPENDS**

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

Note

Disabling *AUTOGEN_ORIGIN_DEPENDS* is useful to avoid building of origin target dependencies when building the **_autogen** target only. This is especially interesting when a**global autogen tar get** is enabled.

When the **_autogen** target doesn't require all the origin target's dependencies, and *AUTOGEN_ORIGIN_DEPENDS* is disabled, it might be necessary to extend **AUTOGEN_TARGET_DEPENDS** to add missing dependencies.

AUTOGEN_PARALLEL

New in version 3.11.

Number of parallel **moc** or **uic** processes to start when using **AUTOMOC** and **AUTOUIC**.

The custom **<origin>_autogen** target starts a number of threads of which each one parses a source file and on demand starts a **moc** or **uic** process. **AUTOGEN_PARALLEL** controls how many parallel threads (and therefore **moc** or **uic** processes) are started.

- An empty (or unset) value or the string **AUTO** sets the number of threads/processes to the number of physical CPUs on the host system.
- A positive non zero integer value sets the exact thread/process count.
- Otherwise a single thread/process is started.

By default **AUTOGEN_PARALLEL** is initialized from **CMAKE_AUTOGEN_PARALLEL**.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

AUTOGEN_TARGET_DEPENDS

Additional target dependencies of the corresponding **_autogen** target.

Targets which have their **AUTOMOC** or **AUTOUIC** property **ON** have a corresponding **_autogen** target which generates **moc** and **uic** files. As this **_autogen** target is created at generate−time, it is not possible to define dependencies of it using e.g. **add_dependencies()**. Instead the*A UTOGEN_TARGET_DEPENDS* target property can be set to a ;−list of additional dependencies for the **_autogen** target. Dependencies can be target names or file names.

In total the dependencies of the **_autogen** target are composed from

- forwarded origin target dependencies (enabled by default via **AUTOGEN_ORIGIN_DEPENDS**)
- additional user defined dependencies from *AUTOGEN_TARGET_DEPENDS*

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

Use cases

If **AUTOMOC** or **AUTOUIC** depends on a file that is either

- a **GENERATED** non C++ file (e.g. a **GENERATED .json** or **.ui** file) or
- a **GENERATED** C++ file that isn't recognized by **AUTOMOC** and **AUTOUIC** because it's skipped by **SKIP_AUTOMOC**, **SKIP_AUTOUIC**, **SKIP_AUTOGEN** or **CMP0071** or
- a file that isn't in the origin target's sources

it must be added to *AUTOGEN_TARGET_DEPENDS*.

AUTOMOC

Should the target be processed with auto−moc (for Qt projects).

AUTOMOC is a boolean specifying whether CMake will handle the Qt **moc** preprocessor automatically, i.e.

without having to use commands like **QT4_WRAP_CPP()**, **QT5_WRAP_CPP()**, etc. Currently, Qt versions 4 to 6 are supported.

This property is initialized by the value of the **CMAKE_AUTOMOC** variable if it is set when a target is created.

When this property is set **ON**, CMake will scan the header and source files at build time and invoke **moc** accordingly.

Header file processing

At configuration time, a list of header files that should be scanned by *AUTOMOC* is computed from the target's sources.

- All header files in the target's sources are added to the scan list.
- For all C++ source files **<source_base>.<source_extension>** in the target's sources, CMake searches for
  - a regular header with the same base name (**<source_base>.<header_extention>**) and
  - a private header with the same base name and a **_p** suffix (**<source_base>_p.<header_extention>**)

  and adds these to the scan list.

At build time, CMake scans each unknown or modified header file from the list and searches for

- a Qt macro from **AUTOMOC_MACRO_NAMES**,
- additional file dependencies from the **FILE** argument of a **Q_PLUGIN_METADATA** macro and
- additional file dependencies detected by filters defined in **AUTOMOC_DEPEND_FILTERS**.

If a Qt macro is found, then the header will be compiled by the **moc** to the output file **moc_<base_name>.cpp**. The complete output file path is described in the section*Output file location*.

The header will be **moc** compiled again if a file from the additional file dependencies changes.

Header **moc** output files **moc_<base_name>.cpp** can be included in source files. In the section *Including header moc files in sources* there is more information on that topic.

Source file processing

At build time, CMake scans each unknown or modified C++ source file from the target's sources for

- a Qt macro from **AUTOMOC_MACRO_NAMES**,
- includes of header **moc** files (see *Including header moc files in sources*),
- additional file dependencies from the **FILE** argument of a **Q_PLUGIN_METADATA** macro and
- additional file dependencies detected by filters defined in **AUTOMOC_DEPEND_FILTERS**.

If a Qt macro is found, then the C++ source file **<base>.<source_extension>** is expected to as well contain an include statement

```
#include <<base>.moc> // or
#include "<base>.moc"
```

The source file then will be compiled by the **moc** to the output file **<base>.moc**. A description of the complete output file path is in section *Output file location*.

The source will be **moc** compiled again if a file from the additional file dependencies changes.

Including header moc files in sources

A source file can include the **moc** output file of a header **<header_base>.<header_extension>** by using an include statement of the form

```
#include <moc_<header_base>.cpp> // or
#include "moc_<header_base>.cpp"
```

If the **moc** output file of a header is included by a source, it will be generated in a different location than if it was not included. This is described in the section *Output file location*.

Output file location

Included moc output files

moc output files that are included by a source file will be generated in

- **<AUTOGEN_BUILD_DIR>/include** for single configuration generators or in
- **<AUTOGEN_BUILD_DIR>/include_<CONFIG>** for **multi configuration** generators.

Where **<AUTOGEN_BUILD_DIR>** is the value of the target property **AUTOGEN_BUILD_DIR**.

The include directory is automatically added to the target's **INCLUDE_DIRECTORIES**.

Not included moc output files

moc output files that are not included in a source file will be generated in

- **<AUTOGEN_BUILD_DIR>/<SOURCE_DIR_CHECKSUM>** for single configuration generators or in,
- **<AUTOGEN_BUILD_DIR>/include_<CONFIG>/<SOURCE_DIR_CHECKSUM>** for **multi configuration** generators.

Where **<SOURCE_DIR_CHECKSUM>** is a checksum computed from the relative parent directory path of the **moc** input file. This scheme allows to have **moc** input files with the same name in different directories.

All not included **moc** output files will be included automatically by the CMake generated file

- **<AUTOGEN_BUILD_DIR>/mocs_compilation.cpp**, or
- **<AUTOGEN_BUILD_DIR>/mocs_compilation_$<CONFIG>.cpp**,

which is added to the target's sources.

Qt version detection

AUTOMOC enabled targets need to know the Qt major and minor version they're working with. The major version usually is provided by the **INTERFACE_QT_MAJOR_VERSION** property of the **Qt[456]Core** library, that the target links to. To find the minor version, CMake builds a list of available Qt versions from

- **Qt6Core_VERSION_MAJOR** and **Qt6Core_VERSION_MINOR** variables (usually set by **find_package(Qt6...)**)
- **Qt6Core_VERSION_MAJOR** and **Qt6Core_VERSION_MINOR** directory properties
- **Qt5Core_VERSION_MAJOR** and **Qt5Core_VERSION_MINOR** variables (usually set by **find_package(Qt5...)**)
- **Qt5Core_VERSION_MAJOR** and **Qt5Core_VERSION_MINOR** directory properties
- **QT_VERSION_MAJOR** and **QT_VERSION_MINOR** variables (usually set by **find_package(Qt4...)**)
- **QT_VERSION_MAJOR** and **QT_VERSION_MINOR** directory properties

in the context of the **add_executable()** or **add_library()** call.

Assumed **INTERFACE_QT_MAJOR_VERSION** is a valid number, the first entry in the list with a matching major version is taken. If no matching major version was found, an error is generated. If **INTERFACE_QT_MAJOR_VERSION** is not a valid number, the first entry in the list is taken.

A **find_package(Qt[456]...)** call sets the **QT/Qt[56]Core_VERSION_MAJOR/MINOR** variables. If the call is in a different context than the **add_executable()** or **add_library()** call, e.g. in a function, then the version variables might not be available to the *AUTOMOC* enabled target. In that case the version variables can be forwarded from the **find_package(Qt[456]...)** calling context to the **add_executable()** or **add_library()** calling context as directory properties. The following Qt5 example demonstrates the procedure.

```
function (add_qt5_client)
  find_package(Qt5 REQUIRED QUIET COMPONENTS Core Widgets)
  ...
  set_property(DIRECTORY "${CMAKE_CURRENT_SOURCE_DIR}"
    PROPERTY Qt5Core_VERSION_MAJOR "${Qt5Core_VERSION_MAJOR}")
  set_property(DIRECTORY "${CMAKE_CURRENT_SOURCE_DIR}"
    PROPERTY Qt5Core_VERSION_MINOR "${Qt5Core_VERSION_MAJOR}")
  ...
endfunction ()
...
add_qt5_client()
add_executable(myTarget main.cpp)
target_link_libraries(myTarget Qt5::QtWidgets)
set_property(TARGET myTarget PROPERTY AUTOMOC ON)
```

Modifiers

AUTOMOC_EXECUTABLE: The **moc** executable will be detected automatically, but can be forced to a certain binary using this target property.

AUTOMOC_MOC_OPTIONS: Additional command line options for **moc** can be set in this target property.

AUTOMOC_MACRO_NAMES: This list of Qt macro names can be extended to search for additional macros in headers and sources.

AUTOMOC_DEPEND_FILTERS: **moc** dependency file names can be extracted from headers or sources by defining file name filters in this target property.

AUTOMOC_COMPILER_PREDEFINES: Compiler pre definitions for **moc** are written to the **moc_predefs.h** file. The generation of this file can be enabled or disabled in this target property.

SKIP_AUTOMOC: Sources and headers can be excluded from *AUTOMOC* processing by setting this source file property.

SKIP_AUTOGEN: Source files can be excluded from *AUTOMOC*, **AUTOUIC** and **AUTORCC** processing by setting this source file property.

AUTOGEN_SOURCE_GROUP: This global property can be used to group files generated by *AUTOMOC* or **AUTORCC** together in an IDE, e.g. in MSVS.

AUTOGEN_TARGETS_FOLDER: This global property can be used to group *AUTOMOC*, **AUTOUIC** and **AUTORCC** targets together in an IDE, e.g. in MSVS.

CMAKE_GLOBAL_AUTOGEN_TARGET: A global **autogen** target, that depends on all *AUTOMOC* or **AUTOUIC** generated **<ORIGIN>_autogen** targets in the project, will be generated when this variable is **ON**.

AUTOGEN_PARALLEL: This target property controls the number of **moc** or **uic** processes to start in parallel during builds.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

AUTOMOC_COMPILER_PREDEFINES
New in version 3.10.

Boolean value used by **AUTOMOC** to determine if the compiler pre definitions file **moc_predefs.h** should be generated.

CMake generates a **moc_predefs.h** file with compiler pre definitions from the output of the command defined in **CMAKE_CXX_COMPILER_PREDEFINES_COMMAND** when

- **AUTOMOC** is enabled,
- *AUTOMOC_COMPILER_PREDEFINES* is enabled,
- **CMAKE_CXX_COMPILER_PREDEFINES_COMMAND** isn't empty and
- the Qt version is greater or equal 5.8.

The **moc_predefs.h** file, which is generated in **AUTOGEN_BUILD_DIR**, is passed to **moc** as the argument to the **−−include** option.

By default *AUTOMOC_COMPILER_PREDEFINES* is initialized from **CMAKE_AUTOMOC_COMPILER_PREDEFINES**, which is ON by default.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

AUTOMOC_DEPEND_FILTERS
New in version 3.9.

Filter definitions used by **AUTOMOC** to extract file names from a source file that are registered as additional dependencies for the **moc** file of the source file.

Filters are defined as **KEYWORD;REGULAR_EXPRESSION** pairs. First the file content is searched for **KEYWORD**. If it is found at least once, then file names are extracted by successively searching for **REGULAR_EXPRESSION** and taking the first match group.

The file name found in the first match group is searched for

- first in the vicinity of the source file
- and afterwards in the target's **INCLUDE_DIRECTORIES**.

If any of the extracted files changes, then the **moc** file for the source file gets rebuilt even when the source file itself doesn't change.

If any of the extracted files is **GENERATED** or if it is not in the target's sources, then it might be necessary to add it to the **_autogen** target dependencies. See**A UTOGEN_TARGET_DEPENDS** for reference.

By default *AUTOMOC_DEPEND_FILTERS* is initialized from

CMAKE_AUTOMOC_DEPEND_FILTERS, which is empty by default.

From Qt 5.15.0 on this variable is ignored as moc is able to output the correct dependencies.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

Example 1

A header file **my_class.hpp** uses a custom macro **JSON_FILE_MACRO** which is defined in an other header **macros.hpp**. We want the **moc** file of **my_class.hpp** to depend on the file name argument of **JSON_FILE_MACRO**:

```
// my_class.hpp
class My_Class : public QObject
{
  Q_OBJECT
  JSON_FILE_MACRO ( "info.json" )
...
};
```

In **CMakeLists.txt** we add a filter to **CMAKE_AUTOMOC_DEPEND_FILTERS** like this:

```
list( APPEND CMAKE_AUTOMOC_DEPEND_FILTERS
  "JSON_FILE_MACRO"
  "[\n][ \t]*JSON_FILE_MACRO[ \t]*\\([ \t]*\"([^\"]+)\""
)
```

We assume **info.json** is a plain (not **GENERATED**) file that is listed in the target's source. Therefore we do not need to add it to **AUTOGEN_TARGET_DEPENDS**.

Example 2

In the target **my_target** a header file **complex_class.hpp** uses a custom macro **JSON_BASED_CLASS** which is defined in an other header **macros.hpp**:

```
// macros.hpp
...
#define JSON_BASED_CLASS(name, json) \
class name : public QObject \
{ \
  Q_OBJECT \
  Q_PLUGIN_METADATA(IID "demo" FILE json) \
  name() {} \
};
...

// complex_class.hpp
#pragma once
JSON_BASED_CLASS(Complex_Class, "meta.json")
// end of file
```

Since **complex_class.hpp** doesn't contain a **Q_OBJECT** macro it would be ignored by **AUTOMOC**. We change this by adding **JSON_BASED_CLASS** to **CMAKE_AUTOMOC_MACRO_NAMES**:

```
list(APPEND CMAKE_AUTOMOC_MACRO_NAMES "JSON_BASED_CLASS")
```

We want the **moc** file of **complex_class.hpp** to depend on **meta.json**. So we add a filter to**CMAKE_A U-TOMOC_DEPEND_FILTERS**:

```
list(APPEND CMAKE_AUTOMOC_DEPEND_FILTERS
  "JSON_BASED_CLASS"
  "[\n^][ \t]*JSON_BASED_CLASS[ \t]*\\([^,]*,[ \t]*\"([^\"]+)\""
)
```

Additionally we assume **meta.json** is **GENERATED** which is why we have to add it to **AUTOGEN_TARGET_DEPENDS**:

```
set_property(TARGET my_target APPEND PROPERTY AUTOGEN_TARGET_DEPENDS "meta.jso
```

AUTOMOC_EXECUTABLE
New in version 3.14.

AUTOMOC_EXECUTABLE is file path pointing to the **moc** executable to use for **AUTOMOC** enabled files. Setting this property will make CMake skip the automatic detection of the **moc** binary as well as the sanity−tests normally run to ensure that the binary is available and working as expected.

Usually this property does not need to be set. Only consider this property if auto−detection of **moc** can not work −− e.g. because you are building the **moc** binary as part of your project.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

AUTOMOC_MACRO_NAMES
New in version 3.10.

A semicolon−separated list list of macro names used by **AUTOMOC** to determine if a C++ file needs to be processed by **moc**.

This property is only used if the **AUTOMOC** property is **ON** for this target.

When running **AUTOMOC**, CMake searches for the strings listed in *AUTOMOC_MACRO_NAMES* in C++ source and header files. If any of the strings is found

- as the first non space string on a new line or
- as the first non space string after a **{** on a new line,

then the file will be processed by **moc**.

By default *AUTOMOC_MACRO_NAMES* is initialized from **CMAKE_AUTOMOC_MACRO_NAMES**.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

Example
In this case the **Q_OBJECT** macro is hidden inside another macro called **CUSTOM_MACRO**. To let CMake know that source files that contain **CUSTOM_MACRO** need to be **moc** processed, we call:

```
set_property(TARGET tgt APPEND PROPERTY AUTOMOC_MACRO_NAMES "CUSTOM_MACRO")
```

AUTOMOC_MOC_OPTIONS
Additional options for **moc** when using **AUTOMOC**

This property is only used if the **AUTOMOC** property is **ON** for this target. In this case, it holds additional command line options which will be used when **moc** is executed during the build, i.e. it is equivalent to the optional **OPTIONS** argument of the **qt4_wrap_cpp()** macro.

This property is initialized by the value of the **CMAKE_AUTOMOC_MOC_OPTIONS** variable if it is set when a target is created, or an empty string otherwise.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

AUTOMOC_PATH_PREFIX
New in version 3.16.

When this property is **ON**, CMake will generate the **−p** path prefix option for **moc** on **AUTOMOC** enabled Qt targets.

To generate the path prefix, CMake tests if the header compiled by **moc** is in any of the target **include directories**. If so, CMake will compute the relative path accordingly. If the header is not in the**include directories**, CMake will omit the **−p** path prefix option. **moc** usually generates a relative include path in that case.

AUTOMOC_PATH_PREFIX is initialized from the variable **CMAKE_AUTOMOC_PATH_PREFIX**, which is **OFF** by default.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

Reproducible builds
For reproducible builds it is recommended to keep headers that are **moc** compiled in one of the target **include directories** and set *AUTOMOC_PATH_PREFIX* to **ON**. This ensures that:

- **moc** output files are identical on different build setups,
- **moc** output files will compile correctly when the source and/or build directory is a symbolic link.

AUTORCC
Should the target be processed with auto−rcc (for Qt projects).

AUTORCC is a boolean specifying whether CMake will handle the Qt **rcc** code generator automatically, i.e. without having to use commands like **QT4_ADD_RESOURCES()**, **QT5_ADD_RESOURCES()**, etc. Currently, Qt versions 4 to 6 are supported.

When this property is **ON**, CMake will handle **.qrc** files added as target sources at build time and invoke **rcc** accordingly. This property is initialized by the value of the **CMAKE_AUTORCC** variable if it is set when a target is created.

By default *AUTORCC* is processed by a **custom command**. If the**.qr c** file is **GENERATED**, a **custom target** is used instead.

When there are multiple **.qrc** files with the same name, CMake will generate unspecified unique output file names for **rcc**. Therefore, if**Q_INIT_RESOURCE()** or **Q_CLEANUP_RESOURCE()** need to be used, the **.qrc** file name must be unique.

Modifiers
AUTORCC_EXECUTABLE: The **rcc** executable will be detected automatically, but can be forced to a certain binary by setting this target property.

AUTORCC_OPTIONS: Additional command line options for **rcc** can be set via this target property. The corresponding **AUTORCC_OPTIONS** source file property can be used to specify options to be applied only to a specific **.qrc** file.

SKIP_AUTORCC: **.qrc** files can be excluded from *AUTORCC* processing by setting this source file property.

SKIP_AUTOGEN: Source files can be excluded from **AUTOMOC**, **AUTOUIC** and *AUTORCC* processing by setting this source file property.

AUTOGEN_SOURCE_GROUP: This global property can be used to group files generated by **AUTOMOC** or *AUTORCC* together in an IDE, e.g. in MSVS.

AUTOGEN_TARGETS_FOLDER: This global property can be used to group **AUTOMOC**, **AUTOUIC** and *AUTORCC* targets together in an IDE, e.g. in MSVS.

CMAKE_GLOBAL_AUTORCC_TARGET: A global **autorcc** target that depends on all *AUTORCC* targets in the project will be generated when this variable is **ON**.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

AUTORCC_EXECUTABLE
New in version 3.14.

AUTORCC_EXECUTABLE is file path pointing to the **rcc** executable to use for **AUTORCC** enabled files. Setting this property will make CMake skip the automatic detection of the **rcc** binary as well as the sanity−tests normally run to ensure that the binary is available and working as expected.

Usually this property does not need to be set. Only consider this property if auto−detection of **rcc** can not work −− e.g. because you are building the **rcc** binary as part of your project.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

AUTORCC_OPTIONS
Additional options for **rcc** when using **AUTORCC**

This property holds additional command line options which will be used when **rcc** is executed during the build via **AUTORCC**, i.e. it is equivalent to the optional **OPTIONS** argument of the **qt4_add_resources()** macro.

This property is initialized by the value of the **CMAKE_AUTORCC_OPTIONS** variable if it is set when a target is created, or an empty string otherwise.

The options set on the target may be overridden by **AUTORCC_OPTIONS** set on the **.qrc** source file.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

EXAMPLE

```
# ...
set_property(TARGET tgt PROPERTY AUTORCC_OPTIONS "−−compress;9")
# ...
```

AUTOUIC
Should the target be processed with auto−uic (for Qt projects).

AUTOUIC is a boolean specifying whether CMake will handle the Qt **uic** code generator automatically, i.e. without having to use commands like **QT4_WRAP_UI()**, **QT5_WRAP_UI()**, etc. Currently, Qt versions 4 to 6 are supported.

This property is initialized by the value of the **CMAKE_AUTOUIC** variable if it is set when a target is created.

When this property is **ON**, CMake will scan the header and source files at build time and invoke **uic**

accordingly.

Header and source file processing

At build time, CMake scans each header and source file from the target's sources for include statements of the form

```
#include "ui_<ui_base>.h"
```

Once such an include statement is found in a file, CMake searches for the **uic** input file **<ui_base>.ui**

- in the vicinity of the file and
- in the **AUTOUIC_SEARCH_PATHS** of the target.

If the **<ui_base>.ui** file was found, **uic** is called on it to generate **ui_<ui_base>.h** in the directory

- **<AUTOGEN_BUILD_DIR>/include** for single configuration generators or in
- **<AUTOGEN_BUILD_DIR>/include_<CONFIG>** for **multi configuration** generators.

Where **<AUTOGEN_BUILD_DIR>** is the value of the target property **AUTOGEN_BUILD_DIR**.

The include directory is automatically added to the target's **INCLUDE_DIRECTORIES**.

Modifiers

AUTOUIC_EXECUTABLE: The **uic** executable will be detected automatically, but can be forced to a certain binary using this target property.

AUTOUIC_OPTIONS: Additional command line options for **uic** can be set via this target property. The corresponding **AUTOUIC_OPTIONS** source file property can be used to specify options to be applied only to a specific **<base_name>.ui** file.

SKIP_AUTOUIC: Source files can be excluded from *AUTOUIC* processing by setting this source file property.

SKIP_AUTOGEN: Source files can be excluded from **AUTOMOC**, *AUTOUIC* and **AUTORCC** processing by setting this source file property.

AUTOGEN_TARGETS_FOLDER: This global property can be used to group **AUTOMOC**, *AUTOUIC* and **AUTORCC** targets together in an IDE, e.g. in MSVS.

CMAKE_GLOBAL_AUTOGEN_TARGET: A global **autogen** target, that depends on all **AUTOMOC** or *AUTOUIC* generated **<ORIGIN>_autogen** targets in the project, will be generated when this variable is **ON**.

AUTOGEN_PARALLEL: This target property controls the number of **moc** or **uic** processes to start in parallel during builds.

See the **cmake-qt(7)** manual for more information on using CMake with Qt.

AUTOUIC_EXECUTABLE

New in version 3.14.

AUTOUIC_EXECUTABLE is file path pointing to the **uic** executable to use for **AUTOUIC** enabled files. Setting this property will make CMake skip the automatic detection of the **uic** binary as well as the sanity-tests normally run to ensure that the binary is available and working as expected.

Usually this property does not need to be set. Only consider this property if auto-detection of **uic** can not

work −− e.g. because you are building the **uic** binary as part of your project.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

AUTOUIC_OPTIONS
Additional options for **uic** when using **AUTOUIC**

This property holds additional command line options which will be used when **uic** is executed during the build via **AUTOUIC**, i.e. it is equivalent to the optional **OPTIONS** argument of the **qt4_wrap_ui()** macro.

This property is initialized by the value of the **CMAKE_AUTOUIC_OPTIONS** variable if it is set when a target is created, or an empty string otherwise.

The options set on the target may be overridden by **AUTOUIC_OPTIONS** set on the **.ui** source file.

This property may use "generator expressions" with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

EXAMPLE

```
# ...
set_property(TARGET tgt PROPERTY AUTOUIC_OPTIONS "−−no−protection")
# ...
```

AUTOUIC_SEARCH_PATHS
New in version 3.9.

Search path list used by **AUTOUIC** to find included **.ui** files.

This property is initialized by the value of the **CMAKE_AUTOUIC_SEARCH_PATHS** variable if it is set when a target is created. Otherwise it is empty.

See the **cmake−qt(7)** manual for more information on using CMake with Qt.

BINARY_DIR
New in version 3.4.

This read−only property reports the value of the **CMAKE_CURRENT_BINARY_DIR** variable in the directory in which the target was defined.

BUILD_RPATH
New in version 3.8.

A semicolon−separated list specifying runtime path (**RPATH**) entries to add to binaries linked in the build tree (for platforms that support it). The entries will *not* be used for binaries in the install tree. See also the **INSTALL_RPATH** target property.

This property is initialized by the value of the variable **CMAKE_BUILD_RPATH** if it is set when a target is created.

This property supports **generator expressions**.

BUILD_RPATH_USE_ORIGIN
New in version 3.14.

Whether to use relative paths for the build **RPATH**.

This property is initialized by the value of the variable **CMAKE_BUILD_RPATH_USE_ORIGIN**.

On platforms that support runtime paths (**RPATH**) with the **$ORIGIN** token, setting this property to **TRUE** enables relative paths in the build **RPATH** for executables and shared libraries that point to shared libraries in the same build tree.

Normally the build **RPATH** of a binary contains absolute paths to the directory of each shared library it links to. The **RPATH** entries for directories contained within the build tree can be made relative to enable relocatable builds and to help achieve reproducible builds by omitting the build directory from the build environment.

This property has no effect on platforms that do not support the **$ORIGIN** token in **RPATH**, or when the **CMAKE_SKIP_RPATH** variable is set. The runtime path set through the **BUILD_RPATH** target property is also unaffected by this property.

BUILD_WITH_INSTALL_NAME_DIR
New in version 3.9.

BUILD_WITH_INSTALL_NAME_DIR is a boolean specifying whether the macOS **install_name** of a target in the build tree uses the directory given by **INSTALL_NAME_DIR**. This setting only applies to targets on macOS.

This property is initialized by the value of the variable **CMAKE_BUILD_WITH_INSTALL_NAME_DIR** if it is set when a target is created.

If this property is not set and policy **CMP0068** is not **NEW**, the value of **BUILD_WITH_INSTALL_RPATH** is used in its place.

BUILD_WITH_INSTALL_RPATH
BUILD_WITH_INSTALL_RPATH is a boolean specifying whether to link the target in the build tree with the **INSTALL_RPATH**. This takes precedence over **SKIP_BUILD_RPATH** and avoids the need for relinking before installation.

This property is initialized by the value of the **CMAKE_BUILD_WITH_INSTALL_RPATH** variable if it is set when a target is created.

If policy **CMP0068** is not **NEW**, this property also controls use of **INSTALL_NAME_DIR** in the build tree on macOS. Either way, the **BUILD_WITH_INSTALL_NAME_DIR** target property takes precedence.

BUNDLE
This target is a **CFBundle** on the macOS.

If a module library target has this property set to true it will be built as a **CFBundle** when built on the mac. It will have the directory structure required for a **CFBundle** and will be suitable to be used for creating Browser Plugins or other application resources.

BUNDLE_EXTENSION
The file extension used to name a **BUNDLE**, a **FRAMEWORK**, or a **MACOSX_BUNDLE** target on the macOS and iOS.

The default value is **bundle**, **framework**, or **app** for the respective target types.

C_EXTENSIONS
New in version 3.1.

Boolean specifying whether compiler specific extensions are requested.

This property specifies whether compiler specific extensions should be used. For some compilers, this results in adding a flag such as **−std=gnu11** instead of **−std=c11** to the compile line. This property is **ON** by default. The basic C standard level is controlled by the **C_STANDARD** target property.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_C_EXTENSIONS** variable if set when a target is created and otherwise by the value of **CMAKE_C_EXTENSIONS_DEFAULT** (see **CMP0128**).

C_STANDARD
New in version 3.1.

The C standard whose features are requested to build this target.

This property specifies the C standard whose features are requested to build this target. For some compilers, this results in adding a flag such as **−std=gnu11** to the compile line. For compilers that have no notion of a C standard level, such as Microsoft Visual C++ before VS 16.7, this property has no effect.

Supported values are:

90 C89/C90

99 C99

11 C11

17 New in version 3.21.

C17

23 New in version 3.21.

C23

If the value requested does not result in a compile flag being added for the compiler in use, a previous standard flag will be added instead. This means that using:

```
set_property(TARGET tgt PROPERTY C_STANDARD 11)
```

with a compiler which does not support **−std=gnu11** or an equivalent flag will not result in an error or warning, but will instead add the **−std=gnu99** or **−std=gnu90** flag if supported. This "decay" behavior may be controlled with the **C_STANDARD_REQUIRED** target property. Additionally, the **C_EXTENSIONS** target property may be used to control whether compiler−specific extensions are enabled on a per−target basis.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_C_STANDARD** variable if it is set when a target is created.

C_STANDARD_REQUIRED
New in version 3.1.

Boolean describing whether the value of **C_STANDARD** is a requirement.

If this property is set to **ON**, then the value of the **C_STANDARD** target property is treated as a requirement. If this property is**OFF** or unset, the **C_ST ANDARD** target property is treated as optional and may "decay" to a previous standard if the requested is not available. For compilers that have no notion of a C standard level, such as Microsoft Visual C++ before VS 16.7, this property has no effect.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_C_STANDARD_REQUIRED** variable if it is set when a target is created.

COMMON_LANGUAGE_RUNTIME
New in version 3.12.

By setting this target property, the target is configured to build with **C++/CLI** support.

The Visual Studio generator defines the **clr** parameter depending on the value of **COMMON_LANGUAGE_RUNTIME**:

- property not set: native C++ (i.e. default)
- property set but empty: mixed unmanaged/managed C++
- property set to any non empty value: managed C++

Supported values: **""**, **"pure"**, **"safe"**

This property is only evaluated Visual Studio Generators for VS 2010 and above.

To be able to build managed C++ targets with VS 2017 and above the component **C++/CLI support** must be installed, which may not be done by default.

See also **IMPORTED_COMMON_LANGUAGE_RUNTIME**

COMPATIBLE_INTERFACE_BOOL
Properties which must be compatible with their link interface

The **COMPATIBLE_INTERFACE_BOOL** property may contain a list of properties for this target which must be consistent when evaluated as a boolean with the **INTERFACE** variant of the property in all linked dependees. For example, if a property **FOO** appears in the list, then for each dependee, the **INTERFACE_FOO** property content in all of its dependencies must be consistent with each other, and with the **FOO** property in the depender.

Consistency in this sense has the meaning that if the property is set, then it must have the same boolean value as all others, and if the property is not set, then it is ignored.

Note that for each dependee, the set of properties specified in this property must not intersect with the set specified in any of the other Compatible Interface Properties.

COMPATIBLE_INTERFACE_NUMBER_MAX
Properties whose maximum value from the link interface will be used.

The **COMPATIBLE_INTERFACE_NUMBER_MAX** property may contain a list of properties for this target whose maximum value may be read at generate time when evaluated in the **INTERFACE** variant of the property in all linked dependees. For example, if a property **FOO** appears in the list, then for each dependee, the **INTERFACE_FOO** property content in all of its dependencies will be compared with each other and with the **FOO** property in the depender. When reading the**FOO** property at generate time, the maximum value will be returned. If the property is not set, then it is ignored.

Note that for each dependee, the set of properties specified in this property must not intersect with the set specified in any of the other Compatible Interface Properties.

COMPATIBLE_INTERFACE_NUMBER_MIN
Properties whose maximum value from the link interface will be used.

The **COMPATIBLE_INTERFACE_NUMBER_MIN** property may contain a list of properties for this target whose minimum value may be read at generate time when evaluated in the **INTERFACE** variant of the property of all linked dependees. For example, if a property **FOO** appears in the list, then for each dependee, the **INTERFACE_FOO** property content in all of its dependencies will be compared with each other and with the **FOO** property in the depender. When reading the**FOO** property at generate time, the minimum value will be returned. If the property is not set, then it is ignored.

Note that for each dependee, the set of properties specified in this property must not intersect with the set specified in any of the other Compatible Interface Properties.

COMPATIBLE_INTERFACE_STRING
Properties which must be string−compatible with their link interface

The **COMPATIBLE_INTERFACE_STRING** property may contain a list of properties for this target which must be the same when evaluated as a string in the **INTERFACE** variant of the property all linked dependees. For example, if a property **FOO** appears in the list, then for each dependee, the **INTERFACE_FOO** property content in all of its dependencies must be equal with each other, and with the **FOO** property in the depender. If the property is not set, then it is ignored.

Note that for each dependee, the set of properties specified in this property must not intersect with the set specified in any of the other Compatible Interface Properties.

COMPILE_DEFINITIONS
Preprocessor definitions for compiling a target's sources.

The **COMPILE_DEFINITIONS** property may be set to a semicolon−separated list of preprocessor definitions using the syntax **VAR** or **VAR=value**. Function−style definitions are not supported. CMake will automatically escape the value correctly for the native build system (note that CMake language syntax may require escapes to specify some values).

CMake will automatically drop some definitions that are not supported by the native build tool.

Disclaimer: Most native build tools have poor support for escaping certain values. CMake has work−arounds for many cases but some values may just not be possible to pass correctly. If a value does not seem to be escaped correctly, do not attempt to work−around the problem by adding escape sequences to the value. Your work−around may break in a future version of CMake that has improved escape support. Instead consider defining the macro in a (configured) header file. Then report the limitation. Known limitations include:

```
#          − broken almost everywhere
```

```
;          - broken in VS IDE 7.0 and Borland Makefiles
,          - broken in VS IDE
%          - broken in some cases in NMake
& |        - broken in some cases on MinGW
^ < > \"   - broken in most Make tools on Windows
```

CMake does not reject these values outright because they do work in some cases. Use with caution.

Contents of **COMPILE_DEFINITIONS** may use "generator expressions" with the syntax **$<...>**. See the **cmake-generator-expressions(7)** manual for available expressions. See the**cmak e-buildsystem(7)** manual for more on defining buildsystem properties.

The corresponding **COMPILE_DEFINITIONS_<CONFIG>** property may be set to specify per-configuration definitions. Generator expressions should be preferred instead of setting the alternative property.

COMPILE_FEATURES
New in version 3.1.

Compiler features enabled for this target.

The list of features in this property are a subset of the features listed in the **CMAKE_C_COMPILE_FEATURES**, **CMAKE_CUDA_COMPILE_FEATURES**, and **CMAKE_CXX_COMPILE_FEATURES** variables.

Contents of **COMPILE_FEATURES** may use "generator expressions" with the syntax **$<...>**. See the **cmake-generator-expressions(7)** manual for available expressions. See the**cmak e-compile-features(7)** manual for information on compile features and a list of supported compilers.

COMPILE_FLAGS
Additional flags to use when compiling this target's sources.

The **COMPILE_FLAGS** property sets additional compiler flags used to build sources within the target. Use **COMPILE_DEFINITIONS** to pass additional preprocessor definitions.

This property is deprecated. Use the **COMPILE_OPTIONS** property or the **target_compile_options()** command instead.

COMPILE_OPTIONS
List of options to pass to the compiler.

This property holds a semicolon-separated list of options specified so far for its target. Use the**tar - get_compile_options()** command to append more options. The options will be added after after flags in the **CMAKE_<LANG>_FLAGS** and **CMAKE_<LANG>_FLAGS_<CONFIG>** variables, but before those propagated from dependencies by the **INTERFACE_COMPILE_OPTIONS** property.

This property is initialized by the **COMPILE_OPTIONS** directory property when a target is created, and is used by the generators to set the options for the compiler.

Contents of **COMPILE_OPTIONS** may use "generator expressions" with the syntax **$<...>**. See the **cmake-generator-expressions(7)** manual for available expressions. See the**cmak e-buildsystem(7)** manual for more on defining buildsystem properties.

Option De-duplication
The final set of options used for a target is constructed by accumulating options from the current target and the usage requirements of its dependencies. The set of options is de-duplicated to avoid repetition.

New in version 3.12: While beneficial for individual options, the de−duplication step can break up option groups. For example, **−option A −option B** becomes **−option A B**. One may specify a group of options using shell−like quoting along with a **SHELL:** prefix. The **SHELL:** prefix is dropped, and the rest of the option string is parsed using the **separate_arguments() UNIX_COMMAND** mode. For example, **"SHELL:−option A" "SHELL:−option B"** becomes **−option A −option B**.

COMPILE_PDB_NAME
New in version 3.1.

Output name for the MS debug symbol **.pdb** file generated by the compiler while building source files.

This property specifies the base name for the debug symbols file. If not set, the default is unspecified.

NOTE:
The compiler−generated program database files are specified by the **/Fd** compiler flag and are not the same as linker−generated program database files specified by the **/pdb** linker flag. Use the **PDB_NAME** property to specify the latter.

COMPILE_PDB_NAME_<CONFIG>
New in version 3.1.

Per−configuration output name for the MS debug symbol **.pdb** file generated by the compiler while building source files.

This is the configuration−specific version of **COMPILE_PDB_NAME**.

NOTE:
The compiler−generated program database files are specified by the **/Fd** compiler flag and are not the same as linker−generated program database files specified by the **/pdb** linker flag. Use the **PDB_NAME_<CONFIG>** property to specify the latter.

COMPILE_PDB_OUTPUT_DIRECTORY
New in version 3.1.

Output directory for the MS debug symbol **.pdb** file generated by the compiler while building source files.

This property specifies the directory into which the MS debug symbols will be placed by the compiler. This property is initialized by the value of the **CMAKE_COMPILE_PDB_OUTPUT_DIRECTORY** variable if it is set when a target is created.

NOTE:
The compiler−generated program database files are specified by the **/Fd** compiler flag and are not the same as linker−generated program database files specified by the **/pdb** linker flag. Use the **PDB_OUTPUT_DIRECTORY** property to specify the latter.

COMPILE_PDB_OUTPUT_DIRECTORY_<CONFIG>
New in version 3.1.

Per−configuration output directory for the MS debug symbol **.pdb** file generated by the compiler while building source files.

This is a per−configuration version of **COMPILE_PDB_OUTPUT_DIRECTORY**, but

multi−configuration generators (Visual Studio, Xcode) do NOT append a per−configuration subdirectory to the specified directory. This property is initialized by the value of the **CMAKE_COMPILE_PDB_OUTPUT_DIRECTORY_<CONFIG>** variable if it is set when a target is created.

NOTE:
The compiler−generated program database files are specified by the **/Fd** compiler flag and are not the same as linker−generated program database files specified by the **/pdb** linker flag. Use the **PDB_OUTPUT_DIRECTORY_<CONFIG>** property to specify the latter.

<CONFIG>_OUTPUT_NAME
Old per−configuration target file base name. Use **OUTPUT_NAME_<CONFIG>** instead.

This is a configuration−specific version of the **OUTPUT_NAME** target property.

<CONFIG>_POSTFIX
Postfix to append to the target file name for configuration <CONFIG>.

When building with configuration <CONFIG> the value of this property is appended to the target file name built on disk. For non−executable targets, this property is initialized by the value of the variable CMAKE_<CONFIG>_POSTFIX if it is set when a target is created. This property is ignored on the Mac for Frameworks and App Bundles.

For macOS see also the **FRAMEWORK_MULTI_CONFIG_POSTFIX_<CONFIG>** target property.

CROSSCOMPILING_EMULATOR
New in version 3.3.

Use the given emulator to run executables created when crosscompiling. This command will be added as a prefix to **add_test()**, **add_custom_command()**, and **add_custom_target()** commands for built target system executables.

If this property contains a semicolon−separated list, then the first value is the command and remaining values are its arguments.

This property is initialized by the value of the **CMAKE_CROSSCOMPILING_EMULATOR** variable if it is set when a target is created.

CUDA_ARCHITECTURES
New in version 3.18.

List of architectures to generate device code for.

An architecture can be suffixed by either **−real** or **−virtual** to specify the kind of architecture to generate code for. If no suffix is given then code is generated for both real and virtual architectures.

A non−empty false value (e.g. **OFF**) disables adding architectures. This is intended to support packagers and rare cases where full control over the passed flags is required.

This property is initialized by the value of the **CMAKE_CUDA_ARCHITECTURES** variable if it is set when a target is created.

The **CUDA_ARCHITECTURES** target property must be set to a non−empty value on targets that compile CUDA sources, or it is an error. See policy **CMP0104**.

Examples

```
set_target_properties(tgt PROPERTIES CUDA_ARCHITECTURES "35;50;72")
```

Generates code for real and virtual architectures **30**, **50** and **72**.

```
set_property(TARGET tgt PROPERTY CUDA_ARCHITECTURES 70-real 72-virtual)
```

Generates code for real architecture **70** and virtual architecture **72**.

```
set_property(TARGET tgt PROPERTY CUDA_ARCHITECTURES OFF)
```

CMake will not pass any architecture flags to the compiler.

CUDA_EXTENSIONS

New in version 3.8.

Boolean specifying whether compiler specific extensions are requested.

This property specifies whether compiler specific extensions should be used. For some compilers, this results in adding a flag such as **-std=gnu++11** instead of **-std=c++11** to the compile line. This property is **ON** by default. The basic CUDA/C++ standard level is controlled by the **CUDA_STANDARD** target property.

See the **cmake-compile-features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_CUDA_EXTENSIONS** variable if set when a target is created and otherwise by the value of **CMAKE_CUDA_EXTENSIONS_DEFAULT** (see **CMP0128**).

CUDA_PTX_COMPILATION

New in version 3.9.

Compile CUDA sources to **.ptx** files instead of **.obj** files within Object Libraries.

For example:

```
add_library(myptx OBJECT a.cu b.cu)
set_property(TARGET myptx PROPERTY CUDA_PTX_COMPILATION ON)
```

CUDA_RESOLVE_DEVICE_SYMBOLS

New in version 3.9.

CUDA only: Enables device linking for the specific library target where required.

If set, this will tell the required compilers to enable device linking on the library target. Device linking is an additional link step required by some CUDA compilers when **CUDA_SEPARABLE_COMPILATION** is enabled. Normally device linking is deferred until a shared library or executable is generated, allowing for multiple static libraries to resolve device symbols at the same time when they are used by a shared library or executable.

By default static library targets have this property is disabled, while shared, module, and executable targets have this property enabled.

Note that device linking is not supported for Object Libraries.

For instance:

```
set_property(TARGET mystaticlib PROPERTY CUDA_RESOLVE_DEVICE_SYMBOLS ON)
```

CUDA_RUNTIME_LIBRARY
New in version 3.17.

Select the CUDA runtime library for use by compilers targeting the CUDA language.

The allowed case insensitive values are:

None Link with **−cudart=none** or equivalent flag(s) to use no CUDA runtime library.

Shared Link with **−cudart=shared** or equivalent flag(s) to use a dynamically−linked CUDA runtime library.

Static Link with **−cudart=static** or equivalent flag(s) to use a statically−linked CUDA runtime library.

Contents of **CUDA_RUNTIME_LIBRARY** may use **generator expressions**.

If that property is not set then CMake uses an appropriate default value based on the compiler to select the CUDA runtime library.

NOTE:
This property has effect only when the **CUDA** language is enabled. To control the CUDA runtime linking when only using the CUDA SDK with the **C** or **C++** language we recommend using the **FindCUDAToolkit** module.

CUDA_SEPARABLE_COMPILATION
New in version 3.8.

CUDA only: Enables separate compilation of device code

If set this will enable separable compilation for all CUDA files for the given target.

For instance:

```
set_property(TARGET myexe PROPERTY CUDA_SEPARABLE_COMPILATION ON)
```

This property is initialized by the value of the **CMAKE_CUDA_SEPARABLE_COMPILATION** variable if it is set when a target is created.

CUDA_STANDARD
New in version 3.8.

The CUDA/C++ standard whose features are requested to build this target.

This property specifies the CUDA/C++ standard whose features are requested to build this target. For some compilers, this results in adding a flag such as **−std=gnu++11** to the compile line.

Supported values are:

98 CUDA C++98. Note that this maps to the same as **03** internally.

03 CUDA C++03

11 CUDA C++11

14 CUDA C++14. While CMake 3.8 and later *recognize* **14** as a valid value, CMake 3.9 was the first version to include support for any compiler.

17 CUDA C++17. While CMake 3.8 and later *recognize* **17** as a valid value, CMake 3.18 was the first version to include support for any compiler.

20 New in version 3.12.

CUDA C++20. While CMake 3.12 and later *recognize* **20** as a valid value, CMake 3.18 was the first version to include support for any compiler.

23 New in version 3.20.

CUDA C++23

If the value requested does not result in a compile flag being added for the compiler in use, a previous standard flag will be added instead. This means that using:

```
set_property(TARGET tgt PROPERTY CUDA_STANDARD 11)
```

with a compiler which does not support **−std=gnu++11** or an equivalent flag will not result in an error or warning, but will instead add the **−std=gnu++03** flag if supported. This "decay" behavior may be controlled with the **CUDA_STANDARD_REQUIRED** target property. Additionally, the **CUDA_EXTENSIONS** target property may be used to control whether compiler−specific extensions are enabled on a per−target basis.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_CUDA_STANDARD** variable if it is set when a target is created.

CUDA_STANDARD_REQUIRED
New in version 3.8.

Boolean describing whether the value of **CUDA_STANDARD** is a requirement.

If this property is set to **ON**, then the value of the **CUDA_STANDARD** target property is treated as a requirement. If this property is**OFF** or unset, the **CUD A_STANDARD** target property is treated as optional and may "decay" to a previous standard if the requested is not available. For compilers that have no notion of a standard level, such as MSVC 1800 (Visual Studio 2013) and lower, this has no effect.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_CUDA_STANDARD_REQUIRED** variable if it is set when a target is created.

CXX_EXTENSIONS
New in version 3.1.

Boolean specifying whether compiler specific extensions are requested.

This property specifies whether compiler specific extensions should be used. For some compilers, this results in adding a flag such as **−std=gnu++11** instead of **−std=c++11** to the compile line. This property is **ON** by default. The basic C++ standard level is controlled by the **CXX_STANDARD** target property.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_CXX_EXTENSIONS** variable if set when a target is created and otherwise by the value of **CMAKE_CXX_EXTENSIONS_DEFAULT** (see **CMP0128**).

CXX_STANDARD
New in version 3.1.

The C++ standard whose features are requested to build this target.

This property specifies the C++ standard whose features are requested to build this target. For some compilers, this results in adding a flag such as **−std=gnu++11** to the compile line. For compilers that have no notion of a standard level, such as Microsoft Visual C++ before 2015 Update 3, this has no effect.

Supported values are:

98 C++98

11 C++11

14 C++14

17 New in version 3.8.

C++17

20 New in version 3.12.

C++20

23 New in version 3.20.

C++23

If the value requested does not result in a compile flag being added for the compiler in use, a previous standard flag will be added instead. This means that using:

```
set_property(TARGET tgt PROPERTY CXX_STANDARD 11)
```

with a compiler which does not support **−std=gnu++11** or an equivalent flag will not result in an error or warning, but will instead add the **−std=gnu++98** flag if supported. This "decay" behavior may be controlled with the **CXX_STANDARD_REQUIRED** target property. Additionally, the **CXX_EXTENSIONS** target property may be used to control whether compiler−specific extensions are enabled on a per−target basis.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_CXX_STANDARD** variable if it is set when a target is created.

CXX_STANDARD_REQUIRED
New in version 3.1.

Boolean describing whether the value of **CXX_STANDARD** is a requirement.

If this property is set to **ON**, then the value of the **CXX_STANDARD** target property is treated as a requirement. If this property is**OFF** or unset, the **CXX_ST ANDARD** target property is treated as optional and may "decay" to a previous standard if the requested is not available. For compilers that have no notion of a standard level, such as MSVC 1800 (Visual Studio 2013) and lower, this has no effect.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_CXX_STANDARD_REQUIRED** variable if it is set when a target is created.

DEBUG_POSTFIX
See target property **<CONFIG>_POSTFIX**.

This property is a special case of the more−general **<CONFIG>_POSTFIX** property for the **DEBUG** configuration.

DEFINE_SYMBOL
Define a symbol when compiling this target's sources.

DEFINE_SYMBOL sets the name of the preprocessor symbol defined when compiling sources in a shared library. If not set here then it is set to**tar get_EXPORTS** by default (with some substitutions if the target is not a valid C identifier). This is useful for headers to know whether they are being included from inside their library or outside to properly setup dllexport/dllimport decorations.

DEPLOYMENT_ADDITIONAL_FILES
New in version 3.13.

Set the WinCE project **AdditionalFiles** in **DeploymentTool** in **.vcproj** files generated by the **Visual Studio 9 2008** generator. This is useful when you want to debug on remote WinCE device. Specify additional files that will be copied to the device. For example:

```
set_property(TARGET ${TARGET} PROPERTY
  DEPLOYMENT_ADDITIONAL_FILES "english.lng|local_folder|remote_folder|0"
  "german.lng|local_folder|remote_folder|0")
```

produces:

```
<DeploymentTool AdditionalFiles="english.lng|local_folder|remote_folder|0;germa
```

DEPLOYMENT_REMOTE_DIRECTORY
New in version 3.6.

Set the WinCE project **RemoteDirectory** in **DeploymentTool** and **RemoteExecutable** in **DebuggerTool** in **.vcproj** files generated by the **Visual Studio 9 2008** generator. This is useful when you want to debug on remote WinCE device. For example:

```
set_property(TARGET ${TARGET} PROPERTY
  DEPLOYMENT_REMOTE_DIRECTORY "\\FlashStorage")
```

produces:

```
<DeploymentTool RemoteDirectory="\FlashStorage" ... />
<DebuggerTool RemoteExecutable="\FlashStorage\target_file" ... />
```

DEPRECATION
New in version 3.17.

Deprecation message from imported target's developer.

DEPRECATION is the message regarding a deprecation status to be displayed to downstream users of a target.

DISABLE_PRECOMPILE_HEADERS
New in version 3.16.

Disables the precompilation of header files specified by **PRECOMPILE_HEADERS** property.

If the property is not set, CMake will use the value provided by **CMAKE_DISABLE_PRECOMPILE_HEADERS**.

DOTNET_TARGET_FRAMEWORK
New in version 3.17.

Specify the .NET target framework.

Used to specify the .NET target framework for C++/CLI and C#. For example: **netcoreapp2.1**.

This property is only evaluated for Visual Studio Generators VS 2010 and above.

Can be initialized for all targets using the variable **CMAKE_DOTNET_TARGET_FRAMEWORK**.

DOTNET_TARGET_FRAMEWORK_VERSION
New in version 3.12.

Specify the .NET target framework version.

Used to specify the .NET target framework version for C++/CLI and C#. For example: **v4.5**.

This property is only evaluated for Visual Studio Generators VS 2010 and above.

To initialize this variable for all targets set **CMAKE_DOTNET_TARGET_FRAMEWORK** or **CMAKE_DOTNET_TARGET_FRAMEWORK_VERSION**. If both are set, the latter is ignored.

EchoString
A message to be displayed when the target is built.

A message to display on some generators (such as Makefile Generators) when the target is built.

ENABLE_EXPORTS
Specify whether an executable exports symbols for loadable modules.

Normally an executable does not export any symbols because it is the final program. It is possible for an executable to export symbols to be used by loadable modules. When this property is set to true CMake will allow other targets to "link" to the executable with the **target_link_libraries()** command. On all platforms a target−level dependency on the executable is created for targets that link to it. Handling of the executable on the link lines of the loadable modules varies by platform:

- On Windows−based systems (including Cygwin) an "import library" is created along with the executable to list the exported symbols. Loadable modules link to the import library to get the symbols.
- On macOS, loadable modules link to the executable itself using the **−bundle_loader** flag.
- On AIX, a linker "import file" is created along with the executable to list the exported symbols for import when linking other targets. Loadable modules link to the import file to get the symbols.
- On other platforms, loadable modules are simply linked without referencing the executable since the dynamic loader will automatically bind symbols when the module is loaded.

This property is initialized by the value of the variable **CMAKE_ENABLE_EXPORTS** if it is set when a target is created.

EXCLUDE_FROM_ALL
Set this target property to a true (or false) value to exclude (or include) the target from the "all" target of the containing directory and its ancestors. If excluded, running e.g. **make** in the containing directory or its ancestors will not build the target by default.

If this target property is not set then the target will be included in the "all" target of the containing directory. Furthermore, it will be included in the "all" target of its ancestor directories unless the **EXCLUDE_FROM_ALL** directory property is set.

With **EXCLUDE_FROM_ALL** set to false or not set at all, the target will be brought up to date as part of doing a **make install** or its equivalent for the CMake generator being used.

If a target has **EXCLUDE_FROM_ALL** set to true, it may still be listed in an **install(TARGETS)** command, but the user is responsible for ensuring that the target's build artifacts are not missing or outdated when an install is performed.

This property may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions.

Only the "Ninja Multi−Config" generator supports a property value that varies by configuration. For all other generators the value of this property must be the same for all configurations.

EXCLUDE_FROM_DEFAULT_BUILD
Exclude target from **Build Solution**.

This property is only used by Visual Studio generators. When set to **TRUE**, the target will not be built when you press **Build Solution**.

EXCLUDE_FROM_DEFAULT_BUILD_<CONFIG>
Per−configuration version of target exclusion from **Build Solution**.

This is the configuration−specific version of **EXCLUDE_FROM_DEFAULT_BUILD**. If the generic**EXCLUDE_FROM_DEFAULT_BUILD** is also set on a target, **EXCLUDE_FROM_DEFAULT_BUILD_<CONFIG>** takes precedence in configurations for which it has a value.

EXPORT_COMPILE_COMMANDS
New in version 3.20.

Enable/Disable output of compile commands during generation for a target.

This property is initialized by the value of the variable **CMAKE_EXPORT_COMPILE_COMMANDS** if it is set when a target is created.

EXPORT_NAME
Exported name for target files.

This sets the name for the **IMPORTED** target generated by the **install(EXPORT)** and **export()** commands. If not set, the logical target name is used by default.

EXPORT_PROPERTIES
New in version 3.12.

List additional properties to export for a target.

This property contains a list of property names that should be exported by the **install(EXPORT)** and **export()** commands. By default only a limited number of properties are exported. This property can be used to additionally export other properties as well.

Properties starting with **INTERFACE_** or **IMPORTED_** are not allowed as they are reserved for internal CMake use.

Properties containing generator expressions are also not allowed.

NOTE:
Since CMake 3.19, Interface Libraries may have arbitrary target properties. If a project exports an interface library with custom properties, the resulting package may not work with dependents configured by older versions of CMake that reject the custom properties.

FOLDER
Set the folder name. Use to organize targets in an IDE.

Targets with no **FOLDER** property will appear as top level entities in IDEs like Visual Studio. Targets with the same **FOLDER** property value will appear next to each other in a folder of that name. To nest folders, use **FOLDER** values such as 'GUI/Dialogs' with '/' characters separating folder levels.

This property is initialized by the value of the variable **CMAKE_FOLDER** if it is set when a target is created.

Fortran_BUILDING_INSTRINSIC_MODULES
New in version 3.22.

Instructs the CMake Fortran preprocessor that the target is building Fortran intrinsics for building a Fortran compiler.

This property is off by default and should be turned only on projects that build a Fortran compiler. It should not be turned on for projects that use a Fortran compiler.

Turning this property on will correctly add dependencies for building Fortran intrinsic modules whereas turning the property off will ignore Fortran intrinsic modules in the dependency graph as they are supplied

by the compiler itself.

Fortran_FORMAT
Set to **FIXED** or **FREE** to indicate the Fortran source layout.

This property tells CMake whether the Fortran source files in a target use fixed−format or free−format. CMake will pass the corresponding format flag to the compiler. Use the source−specific**F ortran_FORMAT** property to change the format of a specific source file. If the variable **CMAKE_Fortran_FORMAT** is set when a target is created its value is used to initialize this property.

Fortran_MODULE_DIRECTORY
Specify output directory for Fortran modules provided by the target.

If the target contains Fortran source files that provide modules and the compiler supports a module output directory this specifies the directory in which the modules will be placed. When this property is not set the modules will be placed in the build directory corresponding to the target's source directory. If the variable **CMAKE_Fortran_MODULE_DIRECTORY** is set when a target is created its value is used to initialize this property.

When using one of the Visual Studio Generators with the Intel Fortran plugin installed in Visual Studio, a subdirectory named after the configuration will be appended to the path where modules are created. For example, if **Fortran_MODULE_DIRECTORY** is set to **C:/some/path**, modules will end up in **C:/some/path/Debug** (or **C:/some/path/Release** etc.) when an Intel Fortran **.vfproj** file is generated, and in **C:/some/path** when any other generator is used.

Note that some compilers will automatically search the module output directory for modules USEd during compilation but others will not. If your sources USE modules their location must be specified by **INCLUDE_DIRECTORIES** regardless of this property.

Fortran_PREPROCESS
New in version 3.18.

Control whether the Fortran source file should be unconditionally preprocessed.

If unset or empty, rely on the compiler to determine whether the file should be preprocessed. If explicitly set to **OFF** then the file does not need to be preprocessed. If explicitly set to **ON**, then the file does need to be preprocessed as part of the compilation step.

When using the **Ninja** generator, all source files are first preprocessed in order to generate module dependency information. Setting this property to **OFF** will make **Ninja** skip this step.

Use the source−specific **Fortran_PREPROCESS** property if a single file needs to be preprocessed. If the variable **CMAKE_Fortran_PREPROCESS** is set when a target is created its value is used to initialize this property.

NOTE:
For some compilers, **NAG**, **PGI** and **Solaris Studio**, setting this to **OFF** will have no effect.

FRAMEWORK
Build **SHARED** or **STATIC** library as Framework Bundle on the macOS and iOS.

If such a library target has this property set to **TRUE** it will be built as a framework when built on the macOS and iOS. It will have the directory structure required for a framework and will be suitable to be used with the **−framework** option. This property is initialized by the value of the **CMAKE_FRAMEWORK** variable if it is set when a target is created.

To customize **Info.plist** file in the framework, use **MACOSX_FRAMEWORK_INFO_PLIST** target property.

For macOS see also the **FRAMEWORK_VERSION** target property.

Example of creation **dynamicFramework**:

```
add_library(dynamicFramework SHARED
            dynamicFramework.c
            dynamicFramework.h
)
set_target_properties(dynamicFramework PROPERTIES
  FRAMEWORK TRUE
  FRAMEWORK_VERSION C
  MACOSX_FRAMEWORK_IDENTIFIER com.cmake.dynamicFramework
  MACOSX_FRAMEWORK_INFO_PLIST Info.plist
  # "current version" in semantic format in Mach-O binary file
  VERSION 16.4.0
  # "compatibility version" in semantic format in Mach-O binary file
  SOVERSION 1.0.0
  PUBLIC_HEADER dynamicFramework.h
  XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY "iPhone Developer"
)
```

FRAMEWORK_MULTI_CONFIG_POSTFIX_<CONFIG>
New in version 3.18.

Postfix to append to the framework file name for configuration **<CONFIG>**, when using a multi-config generator (like Xcode and Ninja Multi-Config).

When building with configuration **<CONFIG>** the value of this property is appended to the framework file name built on disk.

For example, given a framework called **my_fw**, a value of **_debug** for the **FRAMEWORK_MULTI_CONFIG_POSTFIX_DEBUG** property, and **Debug;Release** in **CMAKE_CONFIGURATION_TYPES**, the following relevant files would be created for the **Debug** and **Release** configurations:

- **Release/my_fw.framework/my_fw**
- **Release/my_fw.framework/Versions/A/my_fw**
- **Debug/my_fw.framework/my_fw_debug**
- **Debug/my_fw.framework/Versions/A/my_fw_debug**

For framework targets, this property is initialized by the value of the **CMAKE_FRAMEWORK_MULTI_CONFIG_POSTFIX_<CONFIG>** variable if it is set when a target is created.

This property is ignored for non-framework targets, and when using single config generators.

FRAMEWORK_VERSION
New in version 3.4.

Version of a framework created using the **FRAMEWORK** target property (e.g. **A**).

This property only affects macOS, as iOS doesn't have versioned directory structure.

GENERATOR_FILE_NAME
Generator's file for this target.

An internal property used by some generators to record the name of the project or dsp file associated with this target. Note that at configure time, this property is only set for targets created by **include_external_msproject()**.

GHS_INTEGRITY_APP
New in version 3.14.

ON / **OFF** boolean to determine if an executable target should be treated as an *Integrity Application*.

If no value is set and if a **.int** file is added as a source file to the executable target it will be treated as an *Integrity Application*.

Supported on **Green Hills MULTI**.

GHS_NO_SOURCE_GROUP_FILE
New in version 3.14.

ON / **OFF** boolean to control if the project file for a target should be one single file or multiple files.

The default behavior or when the property is **OFF** is to generate a project file for the target and then a sub−project file for each source group.

When this property is **ON** or if **CMAKE_GHS_NO_SOURCE_GROUP_FILE** is **ON** then only a single project file is generated for the target.

Supported on **Green Hills MULTI**.

GNUtoMS
Convert GNU import library (**.dll.a**) to MS format (**.lib**).

When linking a shared library or executable that exports symbols using GNU tools on Windows (MinGW/MSYS) with Visual Studio installed convert the import library (**.dll.a**) from GNU to MS format (**.lib**). Both import libraries will be installed by**install(T ARGETS)** and exported by **install(EXPORT)** and **export()** to be linked by applications with either GNU− or MS−compatible tools.

If the variable **CMAKE_GNUtoMS** is set when a target is created its value is used to initialize this property. The variable must be set prior to the first command that enables a language such as **project()** or **enable_language()**. CMake provides the variable as an option to the user automatically when configuring on Windows with GNU tools.

HAS_CXX
Link the target using the C++ linker tool (obsolete).

This is equivalent to setting the **LINKER_LANGUAGE** property to **CXX**.

HIP_ARCHITECTURES
New in version 3.21.

List of AMD GPU architectures to generate device code for.

A non−empty false value (e.g. **OFF**) disables adding architectures. This is intended to support packagers

and rare cases where full control over the passed flags is required.

This property is initialized by the value of the **CMAKE_HIP_ARCHITECTURES** variable if it is set when a target is created.

The HIP compilation model has two modes: whole and separable. Whole compilation generates device code at compile time. Separable compilation generates device code at link time. Therefore the **HIP_ARCHITECTURES** target property should be set on targets that compile or link with any HIP sources.

Examples

```
set_property(TARGET tgt PROPERTY HIP_ARCHITECTURES gfx801 gfx900)
```

Generates code for both **gfx801** and **gfx900**.

HIP_EXTENSIONS

New in version 3.21.

Boolean specifying whether compiler specific extensions are requested.

This property specifies whether compiler specific extensions should be used. For some compilers, this results in adding a flag such as **−std=gnu++11** instead of **−std=c++11** to the compile line. This property is **ON** by default. The basic HIP/C++ standard level is controlled by the **HIP_STANDARD** target property.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_HIP_EXTENSIONS** variable if set when a target is created and otherwise by the value of **CMAKE_HIP_EXTENSIONS_DEFAULT** (see **CMP0128**).

HIP_STANDARD

New in version 3.21.

The HIP/C++ standard requested to build this target.

Supported values are:

98 HIP C++98

11 HIP C++11

14 HIP C++14

17 HIP C++17

20 HIP C++20

23 HIP C++23

If the value requested does not result in a compile flag being added for the compiler in use, a previous standard flag will be added instead. This means that using:

```
set_property(TARGET tgt PROPERTY HIP_STANDARD 11)
```

with a compiler which does not support **−std=gnu++11** or an equivalent flag will not result in an error or warning, but will instead add the **−std=gnu++98** flag if supported. This "decay" behavior may be controlled with the **HIP_STANDARD_REQUIRED** target property. Additionally, the **HIP_EXTENSIONS** target property may be used to control whether compiler−specific extensions are enabled on a per−target basis.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_HIP_STANDARD** variable if it is set when a target is created.

HIP_STANDARD_REQUIRED
New in version 3.21.

Boolean describing whether the value of **HIP_STANDARD** is a requirement.

If this property is set to **ON**, then the value of the **HIP_STANDARD** target property is treated as a requirement. If this property is**OFF** or unset, the **HIP_ST ANDARD** target property is treated as optional and may "decay" to a previous standard if the requested is not available.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_HIP_STANDARD_REQUIRED** variable if it is set when a target is created.

IMPLICIT_DEPENDS_INCLUDE_TRANSFORM
Specify **#include** line transforms for dependencies in a target.

This property specifies rules to transform macro−like **#include** lines during implicit dependency scanning of C and C++ source files. The list of rules must be semicolon−separated with each entry of the form **A_MACRO(%)=value−with−%** (the **%** must be literal). During dependency scanning occurrences of **A_MACRO(...)** on **#include** lines will be replaced by the value given with the macro argument substituted for **%**. For example, the entry

```
MYDIR(%)=<mydir/%>
```

will convert lines of the form

```
#include MYDIR(myheader.h)
```

to

```
#include <mydir/myheader.h>
```

allowing the dependency to be followed.

This property applies to sources in the target on which it is set.

IMPORTED
Read−only indication of whether a target is **IMPORTED**.

The boolean value of this property is **True** for targets created with the **IMPORTED** option to **add_executable()** or **add_library()**. It is**F alse** for targets built within the project.

IMPORTED_COMMON_LANGUAGE_RUNTIME
New in version 3.12.

Property to define if the target uses **C++/CLI**.

Ignored for non−imported targets.

See also the **COMMON_LANGUAGE_RUNTIME** target property.

IMPORTED_CONFIGURATIONS
Configurations provided for an IMPORTED target.

Set this to the list of configuration names available for an IMPORTED target. The names correspond to configurations defined in the project from which the target is imported. If the importing project uses a different set of configurations the names may be mapped using the MAP_IMPORTED_CONFIG_<CONFIG> property. Ignored for non−imported targets.

IMPORTED_GLOBAL
New in version 3.11.

Indication of whether an IMPORTED target is globally visible.

The boolean value of this property is True for targets created with the **IMPORTED GLOBAL** options to **add_executable()** or **add_library()**. It is always False for targets built within the project.

For targets created with the **IMPORTED** option to **add_executable()** or **add_library()** but without the additional option **GLOBAL** this is False, too. However, setting this property for such a locally **IMPORTED** target to True promotes that target to global scope. This promotion can only be done in the same directory where that **IMPORTED** target was created in the first place.

NOTE:
Once an imported target has been made global, it cannot be changed back to non−global. Therefore, if a project sets this property, it may only provide a value of True. CMake will issue an error if the project tries to set the property to a non−True value, even if the value was already False.

NOTE:
Local ALIAS targets created before promoting an IMPORTED target from **LOCAL** to **GLOBAL**, keep their initial scope (see **ALIAS_GLOBAL** target property).

IMPORTED_IMPLIB
Full path to the import library for an **IMPORTED** target.

Set this to the location of the **.lib** part of a Windows DLL, or on AIX set it to an import file created for executables that export symbols (see the **ENABLE_EXPORTS** target property). Ignored for non−imported targets.

IMPORTED_IMPLIB_<CONFIG>
<CONFIG>−specific version of **IMPORTED_IMPLIB** property.

Configuration names correspond to those provided by the project from which the target is imported.

IMPORTED_LIBNAME
New in version 3.8.

Specify the link library name for an imported Interface Library.

An interface library builds no library file itself but does specify usage requirements for its consumers. The **IMPORTED_LIBNAME** property may be set to specify a single library name to be placed on the link line in place of the interface library target name as a requirement for using the interface.

This property is intended for use in naming libraries provided by a platform SDK for which the full path to a library file may not be known. The value may be a plain library name such as **foo** but may *not* be a path (e.g. **/usr/lib/libfoo.so**) or a flag (e.g. **−Wl,...**). The name is never treated as a library target name even if it happens to name one.

The **IMPORTED_LIBNAME** property is allowed only on imported Interface Libraries and is rejected on targets of other types (for which the **IMPORTED_LOCATION** target property may be used).

IMPORTED_LIBNAME_<CONFIG>
New in version 3.8.

<CONFIG>−specific version of **IMPORTED_LIBNAME** property.

Configuration names correspond to those provided by the project from which the target is imported.

IMPORTED_LINK_DEPENDENT_LIBRARIES
Dependent shared libraries of an imported shared library.

Shared libraries may be linked to other shared libraries as part of their implementation. On some platforms the linker searches for the dependent libraries of shared libraries they are including in the link. Set this property to the list of dependent shared libraries of an imported library. The list should be disjoint from the list of interface libraries in the **INTERFACE_LINK_LIBRARIES** property. On platforms requiring dependent shared libraries to be found at link time CMake uses this list to add appropriate files or paths to the link command line. Ignored for non−imported targets.

IMPORTED_LINK_DEPENDENT_LIBRARIES_<CONFIG>
<CONFIG>−specific version of **IMPORTED_LINK_DEPENDENT_LIBRARIES**.

Configuration names correspond to those provided by the project from which the target is imported. If set, this property completely overrides the generic property for the named configuration.

IMPORTED_LINK_INTERFACE_LANGUAGES
Languages compiled into an **IMPORTED** static library.

Set this to the list of languages of source files compiled to produce a **STATIC IMPORTED** library (such as **C** or **CXX**). CMake accounts for these languages when computing how to link a target to the imported library. For example, when a C executable links to an imported C++ static library CMake chooses the C++ linker to satisfy language runtime dependencies of the static library.

This property is ignored for targets that are not **STATIC** libraries. This property is ignored for non−imported targets.

IMPORTED_LINK_INTERFACE_LANGUAGES_<CONFIG>
<CONFIG>−specific version of **IMPORTED_LINK_INTERFACE_LANGUAGES**.

Configuration names correspond to those provided by the project from which the target is imported. If set, this property completely overrides the generic property for the named configuration.

IMPORTED_LINK_INTERFACE_LIBRARIES
Transitive link interface of an **IMPORTED** target.

Set this to the list of libraries whose interface is included when an **IMPORTED** library target is linked to another target. The libraries will be included on the link line for the target. Unlike the **LINK_INTERFACE_LIBRARIES** property, this property applies to all imported target types, including **STATIC** libraries. This property is ignored for non−imported targets.

This property is ignored if the target also has a non−empty **INTERFACE_LINK_LIBRARIES** property.

This property is deprecated. Use **INTERFACE_LINK_LIBRARIES** instead.

IMPORTED_LINK_INTERFACE_LIBRARIES_<CONFIG>
<CONFIG>−specific version of **IMPORTED_LINK_INTERFACE_LIBRARIES**.

Configuration names correspond to those provided by the project from which the target is imported. If set, this property completely overrides the generic property for the named configuration.

This property is ignored if the target also has a non−empty **INTERFACE_LINK_LIBRARIES** property.

This property is deprecated. Use **INTERFACE_LINK_LIBRARIES** instead.

IMPORTED_LINK_INTERFACE_MULTIPLICITY
Repetition count for cycles of **IMPORTED** static libraries.

This is **LINK_INTERFACE_MULTIPLICITY** for **IMPORTED** targets.

IMPORTED_LINK_INTERFACE_MULTIPLICITY_<CONFIG>
<CONFIG>−specific version of **IMPORTED_LINK_INTERFACE_MULTIPLICITY**.

If set, this property completely overrides the generic property for the named configuration.

IMPORTED_LOCATION
Full path to the main file on disk for an **IMPORTED** target.

Set this to the location of an **IMPORTED** target file on disk. For executables this is the location of the executable file. For **STATIC** libraries and modules this is the location of the library or module. For **SHARED** libraries on non−DLL platforms this is the location of the shared library. For application bundles on macOS this is the location of the executable file inside **Contents/MacOS** within the bundle folder. For frameworks on macOS this is the location of the library file symlink just inside the framework folder. For DLLs this is the location of the **.dll** part of the library. For **UNKNOWN** libraries this is the location of the file to be linked. Ignored for non−imported targets.

The **IMPORTED_LOCATION** target property may be overridden for a given configuration **<CONFIG>** by the configuration−specific **IMPORTED_LOCATION_<CONFIG>** target property. Furthermore, the **MAP_IMPORTED_CONFIG_<CONFIG>** target property may be used to map between a project's configurations and those of an imported target. If none of these is set then the name of any other configuration listed in the **IMPORTED_CONFIGURATIONS** target property may be selected and its **IMPORTED_LOCATION_<CONFIG>** value used.

To get the location of an imported target read one of the **LOCATION** or **LOCATION_<CONFIG>** properties.

For platforms with import libraries (e.g. Windows) see also **IMPORTED_IMPLIB**.

IMPORTED_LOCATION_<CONFIG>
<CONFIG>−specific version of **IMPORTED_LOCATION** property.

Configuration names correspond to those provided by the project from which the target is imported.

IMPORTED_NO_SONAME
Specifies that an **IMPORTED** shared library target has no **soname**.

Set this property to true for an imported shared library file that has no **soname** field. CMake may adjust generated link commands for some platforms to prevent the linker from using the path to the library in place of its missing **soname**. Ignored for non−imported targets.

IMPORTED_NO_SONAME_<CONFIG>
<CONFIG>−specific version of **IMPORTED_NO_SONAME** property.

Configuration names correspond to those provided by the project from which the target is imported.

IMPORTED_OBJECTS
New in version 3.9.

A semicolon−separated list of absolute paths to the object files on disk for an imported object library.

Ignored for non−imported targets.

Projects may skip **IMPORTED_OBJECTS** if the configuration−specific property **IMPORTED_OBJECTS_<CONFIG>** is set instead, except in situations as noted in the section below.

Xcode Generator Considerations
New in version 3.20.

For Apple platforms, a project may be built for more than one architecture. This is controlled by the **CMAKE_OSX_ARCHITECTURES** variable. For all but the **Xcode** generator, CMake invokes compilers once per source file and passes multiple **−arch** flags, leading to a single object file which will be a universal binary. Such object files work well when listed in the **IMPORTED_OBJECTS** of a separate CMake build, even for the **Xcode** generator. But producing such object files with the**Xcode** generator is more difficult, since it invokes the compiler once per architecture for each source file. Unlike the other generators, it does not generate universal object file binaries.

A further complication with the **Xcode** generator is that when targeting device platforms (iOS, tvOS or watchOS), the **Xcode** generator has the ability to use either the device or simulator SDK without needing CMake to be re−run. The SDK can be selected at build time. But since some architectures can be supported by both the device and the simulator SDKs (e.g. **arm64** with Xcode 12 or later), not all combinations can be represented in a single universal binary. The only solution in this case is to have multiple object files.

IMPORTED_OBJECTS doesn't support generator expressions, so every file it lists needs to be valid for every architecture and SDK. If incorporating object files that are not universal binaries, the path and/or file name of each object file has to somehow encapsulate the different architectures and SDKs. With the **Xcode** generator, Xcode variables of the form **$(...)** can be used to represent these aspects and Xcode will substitute the appropriate values at build time. CMake doesn't interpret these variables and embeds them unchanged in the Xcode project file. **$(CURRENT_ARCH)** can be used to represent the architecture, while **$(EFFECTIVE_PLATFORM_NAME)** can be used to differentiate between SDKs.

The following shows one example of how these two variables can be used to refer to an object file whose location depends on both the SDK and the architecture:

```
add_library(someObjs OBJECT IMPORTED)

set_property(TARGET someObjs PROPERTY IMPORTED_OBJECTS
  # Quotes are required because of the ()
  "/path/to/somewhere/objects$(EFFECTIVE_PLATFORM_NAME)/$(CURRENT_ARCH)/func.o"
)

# Example paths:
#   /path/to/somewhere/objects−iphoneos/arm64/func.o
#   /path/to/somewhere/objects−iphonesimulator/x86_64/func.o
```

In some cases, you may want to have configuration−specific object files as well. The **$(CONFIGURATION)** Xcode variable is often used for this and can be used in conjunction with the others mentioned above:

```
add_library(someObjs OBJECT IMPORTED)
set_property(TARGET someObjs PROPERTY IMPORTED_OBJECTS
  "/path/to/somewhere/$(CONFIGURATION)$(EFFECTIVE_PLATFORM_NAME)/$(CURRENT_ARCH
)

# Example paths:
#   /path/to/somewhere/Release−iphoneos/arm64/func.o
#   /path/to/somewhere/Debug−iphonesimulator/x86_64/func.o
```

When any Xcode variable is used, CMake is not able to fully evaluate the path(s) at configure time. One consequence of this is that the configuration−specific **IMPORTED_OBJECTS_<CONFIG>** properties cannot be used, since CMake cannot determine whether an object file exists at a particular **<CONFIG>** location. The**IMPOR TED_OBJECTS** property must be used for these situations and the configuration−specific aspects of the path should be handled by the **$(CONFIGURATION)** Xcode variable.

IMPORTED_OBJECTS_<CONFIG>
New in version 3.9.

<CONFIG>−specific version of **IMPORTED_OBJECTS** property.

Configuration names correspond to those provided by the project from which the target is imported.

Xcode Generator Considerations
Do not use this **<CONFIG>**−specific property if you need to use Xcode variables like **$(CURRENT_ARCH)** or **$(EFFECTIVE_PLATFORM_NAME)** in the value. The**<CONFIG>**−specific properties will be ignored in such cases because CMake cannot determine whether a file exists at the configuration−specific path at configuration time. For such cases, use **IMPORTED_OBJECTS** instead.

IMPORTED_SONAME
The **soname** of an **IMPORTED** target of shared library type.

Set this to the **soname** embedded in an imported shared library. This is meaningful only on platforms supporting the feature. Ignored for non−imported targets.

IMPORTED_SONAME_<CONFIG>
<CONFIG>−specific version of **IMPORTED_SONAME** property.

Configuration names correspond to those provided by the project from which the target is imported.

IMPORT_PREFIX
What comes before the import library name.

Similar to the target property **PREFIX**, but used for import libraries (typically corresponding to a **DLL**) instead of regular libraries. A target property that can be set to override the prefix (such as **lib**) on an import library name.

IMPORT_SUFFIX
What comes after the import library name.

Similar to the target property **SUFFIX**, but used for import libraries (typically corresponding to a **DLL**) instead of regular libraries. A target property that can be set to override the suffix (such as **.lib**) on an import library name.

INCLUDE_DIRECTORIES
List of preprocessor include file search directories.

This property specifies the list of directories given so far to the **target_include_directories()** command. In addition to accepting values from that command, values may be set directly on any target using the **set_property()** command. A target gets its initial value for this property from the value of the **INCLUDE_DIRECTORIES** directory property. Both directory and target property values are adjusted by calls to the **include_directories()** command.

The value of this property is used by the generators to set the include paths for the compiler.

Relative paths should not be added to this property directly. Use one of the commands above instead to handle relative paths.

Contents of **INCLUDE_DIRECTORIES** may use **cmake−generator−expressions(7)** with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions. See the **cmake−buildsystem(7)** manual for more on defining buildsystem properties.

INSTALL_NAME_DIR
Directory name for installed targets on Apple platforms.

INSTALL_NAME_DIR is a string specifying the directory portion of the "install_name" field of shared libraries on Apple platforms for installed targets. When not set, the default directory used is determined by **MACOSX_RPATH**. Policies**CMP0068** and **CMP0042** are also rele vant.

This property is initialized by the value of the variable **CMAKE_INSTALL_NAME_DIR** if it is set when a target is created.

This property supports **generator expressions**. In particular, the **$<INSTALL_PREFIX>** generator expression can be used to set the directory relative to the install−time prefix.

INSTALL_REMOVE_ENVIRONMENT_RPATH
New in version 3.16.

Controls whether toolchain−defined rpaths should be removed during installation.

When a target is being installed, CMake may need to rewrite its rpath information. This occurs when the install rpath (as specified by the **INSTALL_RPATH** target property) has different contents to the rpath that the target was built with. Some toolchains insert their own rpath contents into the binary as part of the build. By default, CMake will preserve those extra inserted contents in the install rpath. For those scenarios where such toolchain−inserted entries need to be discarded during install, set the **INSTALL_REMOVE_ENVIRONMENT_RPATH** target property to true.

This property is initialized by the value of **CMAKE_INSTALL_REMOVE_ENVIRONMENT_RPATH** when the target is created.

INSTALL_RPATH
The rpath to use for installed targets.

A semicolon−separated list specifying the rpath to use in installed targets (for platforms that support it). This property is initialized by the value of the variable **CMAKE_INSTALL_RPATH** if it is set when a target is created.

Because the rpath may contain **${ORIGIN}**, which coincides with CMake syntax, the contents of **INSTALL_RPATH** are properly escaped in the **cmake_install.cmake** script (see policy **CMP0095**.)

This property supports **generator expressions**.

INSTALL_RPATH_USE_LINK_PATH
Add paths to linker search and installed rpath.

INSTALL_RPATH_USE_LINK_PATH is a boolean that if set to **True** will append to the runtime search path (rpath) of installed binaries any directories outside the project that are in the linker search path or contain linked library files. The directories are appended after the value of the **INSTALL_RPATH** target property.

This property is initialized by the value of the variable **CMAKE_INSTALL_RPATH_USE_LINK_PATH** if it is set when a target is created.

INTERFACE_AUTOUIC_OPTIONS
List of interface options to pass to uic.

Targets may populate this property to publish the options required to use when invoking **uic**. Consuming targets can add entries to their own **AUTOUIC_OPTIONS** property such as **$<TARGET_PROPERTY:foo,INTERFACE_AUTOUIC_OPTIONS>** to use the uic options specified in the interface of **foo**. This is done automatically by the **target_link_libraries()** command.

This property supports generator expressions. See the**cmak e−generator−expressions(7)** manual for available expressions.

INTERFACE_COMPILE_DEFINITIONS
List of public compile definitions requirements for a library.

Targets may populate this property to publish the compile definitions required to compile against the headers for the target. The**tar get_compile_definitions()** command populates this property with values given to the **PUBLIC** and **INTERFACE** keywords. Projects may also get and set the property directly.

When target dependencies are specified using **target_link_libraries()**, CMake will read this property from all target dependencies to determine the build properties of the consumer.

Contents of **INTERFACE_COMPILE_DEFINITIONS** may use "generator expressions" with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions. See the **cmake−buildsystem(7)** −manual for more on defining buildsystem properties.

INTERFACE_COMPILE_FEATURES
New in version 3.1.

List of public compile features requirements for a library.

Targets may populate this property to publish the compile features required to compile against the headers for the target. The**tar get_compile_features()** command populates this property with values given to the **PUBLIC** and **INTERFACE** keywords. Projects may also get and set the property directly.

When target dependencies are specified using **target_link_libraries()**, CMake will read this property from all target dependencies to determine the build properties of the consumer.

Contents of **INTERFACE_COMPILE_FEATURES** may use "generator expressions" with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions. See the **cmake−buildsystem(7)** −manual for more on defining buildsystem properties.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

INTERFACE_COMPILE_OPTIONS
List of public compile options requirements for a library.

Targets may populate this property to publish the compile options required to compile against the headers for the target. The**tar get_compile_options()** command populates this property with values given to the **PUBLIC** and **INTERFACE** keywords. Projects may also get and set the property directly.

When target dependencies are specified using **target_link_libraries()**, CMake will read this property from all target dependencies to determine the build properties of the consumer.

Contents of **INTERFACE_COMPILE_OPTIONS** may use "generator expressions" with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions. See the **cmake−buildsystem(7)** −manual for more on defining buildsystem properties.

INTERFACE_INCLUDE_DIRECTORIES
List of public include directories requirements for a library.

Targets may populate this property to publish the include directories required to compile against the headers for the target. The**tar get_include_directories()** command populates this property with values given to the **PUBLIC** and **INTERFACE** keywords. Projects may also get and set the property directly.

When target dependencies are specified using **target_link_libraries()**, CMake will read this property from all target dependencies to determine the build properties of the consumer.

Contents of **INTERFACE_INCLUDE_DIRECTORIES** may use "generator expressions" with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions. See the **cmake−buildsystem(7)** −manual for more on defining buildsystem properties.

Include directories usage requirements commonly differ between the build−tree and the install−tree. The **BUILD_INTERFACE** and **INSTALL_INTERFACE** generator expressions can be used to describe separate usage requirements based on the usage location. Relative paths are allowed within the **INSTALL_INTERFACE** expression and are interpreted relative to the installation prefix. For example:

```
target_include_directories(mylib INTERFACE
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include/mylib>
  $<INSTALL_INTERFACE:include/mylib>  # <prefix>/include/mylib
)
```

Creating Relocatable Packages
Note that it is not advisable to populate the **INSTALL_INTERFACE** of the **INTERFACE_INCLUDE_DIRECTORIES** of a target with absolute paths to the include directories of dependencies. That would hard−code into installed packages the include directory paths for dependencies **as found on the machine the package was made on**.

The **INSTALL_INTERFACE** of the **INTERFACE_INCLUDE_DIRECTORIES** is only suitable for specifying the required include directories for headers provided with the target itself, not those provided by the transitive dependencies listed in its **INTERFACE_LINK_LIBRARIES** target property. Those dependencies should themselves be targets that specify their own header locations in **INTERFACE_INCLUDE_DIRECTORIES**.

See the Creating Relocatable Packages section of the **cmake−packages(7)** manual for discussion of additional care that must be taken when specifying usage requirements while creating packages for redistribution.

INTERFACE_LINK_DEPENDS
New in version 3.13.

Additional public interface files on which a target binary depends for linking.

This property is supported only by **Ninja** and Makefile Generators. It is intended to specify dependencies on "linker scripts" for custom Makefile link rules.

When target dependencies are specified using **target_link_libraries()**, CMake will read this property from all target dependencies to determine the build properties of the consumer.

Contents of **INTERFACE_LINK_DEPENDS** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. See the**cmak e−buildsystem(7)** −manual for more on defining buildsystem properties.

Link dependency files usage requirements commonly differ between the build−tree and the install−tree. The **BUILD_INTERFACE** and **INSTALL_INTERFACE** generator expressions can be used to describe separate usage requirements based on the usage location. Relative paths are allowed within the **INSTALL_INTERFACE** expression and are interpreted relative to the installation prefix. For example:

```
set_property(TARGET mylib PROPERTY INTERFACE_LINK_DEPENDS
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/mylinkscript>
  $<INSTALL_INTERFACE:mylinkscript>  # <prefix>/mylinkscript
)
```

INTERFACE_LINK_DIRECTORIES
New in version 3.13.

List of public link directories requirements for a library.

Targets may populate this property to publish the link directories required to compile against the headers for the target. The**tar get_link_directories()** command populates this property with values given to the **PUBLIC** and **INTERFACE** keywords. Projects may also get and set the property directly.

When target dependencies are specified using **target_link_libraries()**, CMake will read this property from all target dependencies to determine the build properties of the consumer.

Contents of **INTERFACE_LINK_DIRECTORIES** may use "generator expressions" with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions. See the **cmake−buildsystem(7)** −manual for more on defining buildsystem properties.

INTERFACE_LINK_LIBRARIES
List public interface libraries for a library.

This property contains the list of transitive link dependencies. When the target is linked into another target using the **target_link_libraries()** command, the libraries listed (and recursively their link interface libraries) will be provided to the other target also. This property is overridden by the **LINK_INTERFACE_LIBRARIES** or **LINK_INTERFACE_LIBRARIES_<CONFIG>** property if policy **CMP0022** is **OLD** or unset.

Contents of **INTERFACE_LINK_LIBRARIES** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. See the**cmak e−buildsystem(7)** manual for more on defining buildsystem properties.

NOTE:

A call to **target_link_libraries(<target> ...)** may update this property on **<target>**. If**<tar get>** was not created in the same directory as the call then **target_link_libraries()** will wrap each entry with the form **::@(directory−id);...;::@**, where the **::@** is literal and the **(directory−id)** is unspecified. This tells the generators that the named libraries must be looked up in the scope of the caller rather than in the scope in which the **<target>** was created. Valid directory ids are stripped on export by the **install(EXPORT)** and **export()** commands.

Creating Relocatable Packages

Note that it is not advisable to populate the **INTERFACE_LINK_LIBRARIES** of a target with absolute paths to dependencies. That would hard−code into installed packages the library file paths for dependencies **as found on the machine the package was made on**.

See the Creating Relocatable Packages section of the **cmake−packages(7)** manual for discussion of additional care that must be taken when specifying usage requirements while creating packages for redistribution.

INTERFACE_LINK_OPTIONS

New in version 3.13.

List of public link options requirements for a library.

Targets may populate this property to publish the link options required to compile against the headers for the target. The**tar get_link_options()** command populates this property with values given to the **PUBLIC** and **INTERFACE** keywords. Projects may also get and set the property directly.

When target dependencies are specified using **target_link_libraries()**, CMake will read this property from all target dependencies to determine the build properties of the consumer.

Contents of **INTERFACE_LINK_OPTIONS** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. See the**cmak e−buildsystem(7)** −manual for more on defining buildsystem properties.

INTERFACE_POSITION_INDEPENDENT_CODE

Whether consumers need to create a position−independent target

The **INTERFACE_POSITION_INDEPENDENT_CODE** property informs consumers of this target whether they must set their **POSITION_INDEPENDENT_CODE** property to **ON**. If this property is set to **ON**, then the **POSITION_INDEPENDENT_CODE** property on all consumers will be set to **ON**. Similarly, if this property is set to **OFF**, then the **POSITION_INDEPENDENT_CODE** property on all consumers will be set to **OFF**. If this property is undefined, then consumers will determine their**POSITION_INDEPENDENT_CODE** property by other means. Consumers must ensure that the targets that they link to have a consistent requirement for their **INTERFACE_POSITION_INDEPENDENT_CODE** property.

Contents of **INTERFACE_POSITION_INDEPENDENT_CODE** may use "generator expressions" with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions. See the **cmake−buildsystem(7)** manual for more on defining buildsystem properties.

INTERFACE_PRECOMPILE_HEADERS

New in version 3.16.

List of interface header files to precompile into consuming targets.

Targets may populate this property to publish the header files for consuming targets to precompile. The

target_precompile_headers() command populates this property with values given to the **PUBLIC** and **INTERFACE** keywords. Projects may also get and set the property directly. See the discussion in**tar - get_precompile_headers()** for guidance on appropriate use of this property for installed or exported targets.

Contents of **INTERFACE_PRECOMPILE_HEADERS** may use "generator expressions" with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions. See the **cmake−buildsystem(7)** manual for more on defining buildsystem properties.

INTERFACE_SOURCES
New in version 3.1.

List of interface sources to compile into consuming targets.

Targets may populate this property to publish the sources for consuming targets to compile. The **target_sources()** command populates this property with values given to the **PUBLIC** and **INTERFACE** keywords. Projects may also get and set the property directly.

When target dependencies are specified using **target_link_libraries()**, CMake will read this property from all target dependencies to determine the sources of the consumer.

Contents of **INTERFACE_SOURCES** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. See the**cmak e−buildsystem(7)** manual for more on defining buildsystem properties.

INTERFACE_SYSTEM_INCLUDE_DIRECTORIES
List of public system include directories for a library.

Targets may populate this property to publish the include directories which contain system headers, and therefore should not result in compiler warnings. The**tar get_include_directories(SYSTEM)** command signature populates this property with values given to the **PUBLIC** and **INTERFACE** keywords.

Projects may also get and set the property directly, but must be aware that adding directories to this property does not make those directories used during compilation. Adding directories to this property marks directories as **SYSTEM** which otherwise would be used in a non−**SYSTEM** manner. This can appear similar to 'duplication', so prefer the high−level **target_include_directories(SYSTEM)** command and avoid setting the property by low−level means.

When target dependencies are specified using **target_link_libraries()**, CMake will read this property from all target dependencies to mark the same include directories as containing system headers.

Contents of **INTERFACE_SYSTEM_INCLUDE_DIRECTORIES** may use "generator expressions" with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions. See the **cmake−buildsystem(7)** manual for more on defining buildsystem properties.

INTERPROCEDURAL_OPTIMIZATION
Enable interprocedural optimization for a target.

If set to true, enables interprocedural optimizations if they are known **to be supported** by the compiler. Depending on value of policy **CMP0069**, the error will be reported or ignored, if interprocedural optimization is enabled but not supported.

This property is initialized by the **CMAKE_INTERPROCEDURAL_OPTIMIZATION** variable if it is set when a target is created.

INTERPROCEDURAL_OPTIMIZATION_<CONFIG>
Per−configuration interprocedural optimization for a target.

This is a per−configuration version of **INTERPROCEDURAL_OPTIMIZATION**. If set, this property overrides the generic property for the named configuration.

This property is initialized by the **CMAKE_INTERPROCEDURAL_OPTIMIZATION_<CONFIG>** variable if it is set when a target is created.

IOS_INSTALL_COMBINED
New in version 3.5.

Build a combined (device and simulator) target when installing.

When this property is set to set to false (which is the default) then it will either be built with the device SDK or the simulator SDK depending on the SDK set. But if this property is set to true then the target will at install time also be built for the corresponding SDK and combined into one library.

NOTE:
If a selected architecture is available for both: device SDK and simulator SDK it will be built for the SDK selected by **CMAKE_OSX_SYSROOT** and removed from the corresponding SDK.

This feature requires at least Xcode version 6.

ISPC_HEADER_DIRECTORY
New in version 3.19.

Specify relative output directory for ISPC headers provided by the target.

If the target contains ISPC source files, this specifies the directory in which the generated headers will be placed. Relative paths are treated with respect to the value of **CMAKE_CURRENT_BINARY_DIR**. When this property is not set, the headers will be placed a generator defined build directory. If the variable **CMAKE_ISPC_HEADER_DIRECTORY** is set when a target is created its value is used to initialize this property.

ISPC_HEADER_SUFFIX
New in version 3.19.2.

Specify output suffix to be used for ISPC generated headers provided by the target.

This property is initialized by the value of the **CMAKE_ISPC_HEADER_SUFFIX** variable if it is set when a target is created.

If the target contains ISPC source files, this specifies the header suffix to be used for the generated headers.

The default value is **_ispc.h**.

ISPC_INSTRUCTION_SETS
New in version 3.19.

List of instruction set architectures to generate code for.

This property is initialized by the value of the **CMAKE_ISPC_INSTRUCTION_SETS** variable if it is set

when a target is created.

The **ISPC_INSTRUCTION_SETS** target property must be used when generating for multiple instruction sets so that CMake can track what object files will be generated.

Examples

```
set_property(TARGET tgt PROPERTY ISPC_INSTRUCTION_SETS avx2-i32x4 avx512skx-i3
```

Generates code for avx2 and avx512skx target architectures.

JOB_POOL_COMPILE

Ninja only: Pool used for compiling.

The number of parallel compile processes could be limited by defining pools with the global **JOB_POOLS** property and then specifying here the pool name.

For instance:

```
set_property(TARGET myexe PROPERTY JOB_POOL_COMPILE ten_jobs)
```

This property is initialized by the value of **CMAKE_JOB_POOL_COMPILE**.

JOB_POOL_LINK

Ninja only: Pool used for linking.

The number of parallel link processes could be limited by defining pools with the global **JOB_POOLS** property and then specifying here the pool name.

For instance:

```
set_property(TARGET myexe PROPERTY JOB_POOL_LINK two_jobs)
```

This property is initialized by the value of **CMAKE_JOB_POOL_LINK**.

JOB_POOL_PRECOMPILE_HEADER

New in version 3.17.

Ninja only: Pool used for generating pre−compiled headers.

The number of parallel compile processes could be limited by defining pools with the global **JOB_POOLS** property and then specifying here the pool name.

For instance:

```
set_property(TARGET myexe PROPERTY JOB_POOL_PRECOMPILE_HEADER two_jobs)
```

This property is initialized by the value of **CMAKE_JOB_POOL_PRECOMPILE_HEADER**.

If neither *JOB_POOL_PRECOMPILE_HEADER* nor **CMAKE_JOB_POOL_PRECOMPILE_HEADER** are set then **JOB_POOL_COMPILE** will be used for this task.

LABELS

Specify a list of text labels associated with a target.

Target label semantics are currently unspecified.

<LANG>_CLANG_TIDY
New in version 3.6.

This property is implemented only when **<LANG>** is **C**, **CXX**, **OBJC** or **OBJCXX**.

Specify a semicolon−separated list containing a command line for the **clang−tidy** tool. The Makefile Generators and the **Ninja** generator will run this tool along with the compiler and report a warning if the tool reports any problems.

This property is initialized by the value of the **CMAKE_<LANG>_CLANG_TIDY** variable if it is set when a target is created.

<LANG>_COMPILER_LAUNCHER
New in version 3.4.

This property is implemented only when **<LANG>** is **C**, **CXX**, **Fortran**, **HIP**, **ISPC**, **OBJC**, **OBJCXX**, or **CUDA**.

Specify a semicolon−separated list containing a command line for a compiler launching tool. The Makefile Generators and the **Ninja** generator will run this tool and pass the compiler and its arguments to the tool. Some example tools are distcc and ccache.

This property is initialized by the value of the **CMAKE_<LANG>_COMPILER_LAUNCHER** variable if it is set when a target is created.

<LANG>_CPPCHECK
New in version 3.10.

This property is supported only when **<LANG>** is **C** or **CXX**.

Specify a semicolon−separated list containing a command line for the **cppcheck** static analysis tool. The Makefile Generators and the **Ninja** generator will run **cppcheck** along with the compiler and report any problems. If the command−line specifies the exit code options to **cppcheck** then the build will fail if the tool returns non−zero.

This property is initialized by the value of the **CMAKE_<LANG>_CPPCHECK** variable if it is set when a target is created.

<LANG>_CPPLINT
New in version 3.8.

This property is supported only when **<LANG>** is **C** or **CXX**.

Specify a semicolon−separated list containing a command line for the **cpplint** style checker. The Makefile Generators and the **Ninja** generator will run **cpplint** along with the compiler and report any problems.

This property is initialized by the value of the **CMAKE_<LANG>_CPPLINT** variable if it is set when a target is created.

<LANG>_EXTENSIONS
The variations are:

- **C_EXTENSIONS**
- **CXX_EXTENSIONS**
- **CUDA_EXTENSIONS**
- **HIP_EXTENSIONS**
- **OBJC_EXTENSIONS**
- **OBJCXX_EXTENSIONS**

These properties specify whether compiler−specific extensions are requested.

These properties are initialized by the value of the **CMAKE_<LANG>_EXTENSIONS** variable if it is set when a target is created and otherwise by the value of **CMAKE_<LANG>_EXTENSIONS_DEFAULT** (see **CMP0128**).

For supported CMake versions see the respective pages. To control language standard versions see **<LANG>_STANDARD**.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

<LANG>_INCLUDE_WHAT_YOU_USE
New in version 3.3.

This property is implemented only when **<LANG>** is **C** or **CXX**.

Specify a semicolon−separated list containing a command line for the **include−what−you−use** tool. The Makefile Generators and the **Ninja** generator will run this tool along with the compiler and report a warning if the tool reports any problems.

This property is initialized by the value of the **CMAKE_<LANG>_INCLUDE_WHAT_YOU_USE** variable if it is set when a target is created.

<LANG>_LINKER_LAUNCHER
New in version 3.21.

This property is implemented only when **<LANG>** is **C**, **CXX**, **OBJC**, or **OBJCXX**

Specify a semicolon−separated list containing a command line for a linker launching tool. The Makefile Generators and the **Ninja** generator will run this tool and pass the linker and its arguments to the tool. This is useful for tools such as static analyzers.

This property is initialized by the value of the **CMAKE_<LANG>_LINKER_LAUNCHER** variable if it is set when a target is created.

<LANG>_STANDARD
The variations are:

- **C_STANDARD**
- **CXX_STANDARD**
- **CUDA_STANDARD**
- **HIP_STANDARD**

- **OBJC_STANDARD**
- **OBJCXX_STANDARD**

These properties specify language standard versions which are requested. When a newer standard is specified than is supported by the compiler, then it will fallback to the latest supported standard. This "decay" behavior may be controlled with the **<LANG>_STANDARD_REQUIRED** target property.

These properties are initialized by the value of the **CMAKE_<LANG>_STANDARD** variable if it is set when a target is created.

For supported values and CMake versions see the respective pages. To control compiler−specific extensions see **<LANG>_EXTENSIONS**.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

<LANG>_STANDARD_REQUIRED
The variations are:

- **C_STANDARD_REQUIRED**
- **CXX_STANDARD_REQUIRED**
- **CUDA_STANDARD_REQUIRED**
- **HIP_STANDARD_REQUIRED**
- **OBJC_STANDARD_REQUIRED**
- **OBJCXX_STANDARD_REQUIRED**

These properties specify whether the value of **<LANG>_STANDARD** is a requirement. When **OFF** or unset, the **<LANG>_STANDARD** target property is treated as optional and may "decay" to a previous standard if the requested is not available.

These properties are initialized by the value of the **CMAKE_<LANG>_STANDARD_REQUIRED** variable if it is set when a target is created.

For supported CMake versions see the respective pages. To control language standard versions see **<LANG>_STANDARD**.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

<LANG>_VISIBILITY_PRESET
Value for symbol visibility compile flags

The **<LANG>_VISIBILITY_PRESET** property determines the value passed in a visibility related compile option, such as **−fvisibility=** for **<LANG>**. This property affects compilation in sources of all types of targets (subject to policy **CMP0063**).

This property is initialized by the value of the **CMAKE_<LANG>_VISIBILITY_PRESET** variable if it is set when a target is created.

LIBRARY_OUTPUT_DIRECTORY
Output directory in which to build LIBRARY target files.

This property specifies the directory into which library target files should be built. The property value may use **generator expressions**. Multi−configuration generators (Visual Studio, **Xcode**, **Ninja Multi−Config**)

append a per−configuration subdirectory to the specified directory unless a generator expression is used.

This property is initialized by the value of the **CMAKE_LIBRARY_OUTPUT_DIRECTORY** variable if it is set when a target is created.

See also the **LIBRARY_OUTPUT_DIRECTORY_<CONFIG>** target property.

LIBRARY_OUTPUT_DIRECTORY_<CONFIG>
Per−configuration output directory for LIBRARY target files.

This is a per−configuration version of the **LIBRARY_OUTPUT_DIRECTORY** target property, but multi−configuration generators (Visual Studio Generators, **Xcode**) do NOT append a per−configuration subdirectory to the specified directory. This property is initialized by the value of the **CMAKE_LIBRARY_OUTPUT_DIRECTORY_<CONFIG>** variable if it is set when a target is created.

Contents of **LIBRARY_OUTPUT_DIRECTORY_<CONFIG>** may use **generator expressions**.

LIBRARY_OUTPUT_NAME
Output name for LIBRARY target files.

This property specifies the base name for library target files. It overrides **OUTPUT_NAME** and **OUTPUT_NAME_<CONFIG>** properties.

See also the **LIBRARY_OUTPUT_NAME_<CONFIG>** target property.

LIBRARY_OUTPUT_NAME_<CONFIG>
Per−configuration output name for LIBRARY target files.

This is the configuration−specific version of the **LIBRARY_OUTPUT_NAME** target property.

LINK_DEPENDS
Additional files on which a target binary depends for linking.

Specifies a semicolon−separated list of full−paths to files on which the link rule for this target depends. The target binary will be linked if any of the named files is newer than it.

This property is supported only by **Ninja** and Makefile Generators. It is intended to specify dependencies on "linker scripts" for custom Makefile link rules.

Contents of **LINK_DEPENDS** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. See the**cmak e−buildsystem(7)** manual for more on defining buildsystem properties.

LINK_DEPENDS_NO_SHARED
Do not depend on linked shared library files.

Set this property to true to tell CMake generators not to add file−level dependencies on the shared library files linked by this target. Modification to the shared libraries will not be sufficient to re−link this target. Logical target−level dependencies will not be affected so the linked shared libraries will still be brought up to date before this target is built.

This property is initialized by the value of the **CMAKE_LINK_DEPENDS_NO_SHARED** variable if it is set when a target is created.

LINK_DIRECTORIES
New in version 3.13.

List of directories to use for the link step of shared library, module and executable targets.

This property holds a semicolon−separated list of directories specified so far for its target. Use the**tar - get_link_directories()** command to append more search directories.

This property is initialized by the **LINK_DIRECTORIES** directory property when a target is created, and is used by the generators to set the search directories for the linker.

Contents of **LINK_DIRECTORIES** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. See the**cmak e−buildsystem(7)** manual for more on defining buildsystem properties.

LINK_FLAGS
Additional flags to use when linking this target if it is a shared library, module library, or an executable. Static libraries need to use **STATIC_LIBRARY_OPTIONS** or **STATIC_LIBRARY_FLAGS** properties.

The **LINK_FLAGS** property, managed as a string, can be used to add extra flags to the link step of a target. **LINK_FLAGS_<CONFIG>** will add to the configuration **<CONFIG>**, for example, **DEBUG**, **RELEASE**, **MINSIZEREL**, **RELWITHDEBINFO**, ...

NOTE:
This property has been superseded by **LINK_OPTIONS** property.

LINK_FLAGS_<CONFIG>
Per−configuration linker flags for a **SHARED** library, **MODULE** or **EXECUTABLE** target.

This is the configuration−specific version of **LINK_FLAGS**.

NOTE:
This property has been superseded by **LINK_OPTIONS** property.

LINK_INTERFACE_LIBRARIES
List public interface libraries for a shared library or executable.

By default linking to a shared library target transitively links to targets with which the library itself was linked. For an executable with exports (see the **ENABLE_EXPORTS** target property) no default transitive link dependencies are used. This property replaces the default transitive link dependencies with an explicit list. When the target is linked into another target using the **target_link_libraries()** command, the libraries listed (and recursively their link interface libraries) will be provided to the other target also. If the list is empty then no transitive link dependencies will be incorporated when this target is linked into another target even if the default set is non−empty. This property is initialized by the value of the **CMAKE_LINK_INTERFACE_LIBRARIES** variable if it is set when a target is created. This property is ignored for **STATIC** libraries.

This property is overridden by the **INTERFACE_LINK_LIBRARIES** property if policy **CMP0022** is **NEW**.

This property is deprecated. Use **INTERFACE_LINK_LIBRARIES** instead.

Creating Relocatable Packages
Note that it is not advisable to populate the **LINK_INTERFACE_LIBRARIES** of a target with absolute paths to dependencies. That would hard−code into installed packages the library file paths for dependencies **as found on the machine the package was made on**.

See the Creating Relocatable Packages section of the **cmake−packages(7)** manual for discussion of additional care that must be taken when specifying usage requirements while creating packages for redistribution.

LINK_INTERFACE_LIBRARIES_<CONFIG>
Per−configuration list of public interface libraries for a target.

This is the configuration−specific version of **LINK_INTERFACE_LIBRARIES**. If set, this property completely overrides the generic property for the named configuration.

This property is overridden by the **INTERFACE_LINK_LIBRARIES** property if policy **CMP0022** is **NEW**.

This property is deprecated. Use **INTERFACE_LINK_LIBRARIES** instead.

Creating Relocatable Packages
Note that it is not advisable to populate the **LINK_INTERFACE_LIBRARIES_<CONFIG>** of a target with absolute paths to dependencies. That would hard−code into installed packages the library file paths for dependencies **as found on the machine the package was made on**.

See the Creating Relocatable Packages section of the **cmake−packages(7)** manual for discussion of additional care that must be taken when specifying usage requirements while creating packages for redistribution.

LINK_INTERFACE_MULTIPLICITY
Repetition count for **STATIC** libraries with cyclic dependencies.

When linking to a **STATIC** library target with cyclic dependencies the linker may need to scan more than once through the archives in the strongly connected component of the dependency graph. CMake by default constructs the link line so that the linker will scan through the component at least twice. This property specifies the minimum number of scans if it is larger than the default. CMake uses the largest value specified by any target in a component.

LINK_INTERFACE_MULTIPLICITY_<CONFIG>
Per−configuration repetition count for cycles of **STATIC** libraries.

This is the configuration−specific version of **LINK_INTERFACE_MULTIPLICITY**. If set, this property completely overrides the generic property for the named configuration.

LINK_LIBRARIES
List of direct link dependencies.

This property specifies the list of libraries or targets which will be used for linking. In addition to accepting values from the **target_link_libraries()** command, values may be set directly on any target using the **set_property()** command.

The value of this property is used by the generators to set the link libraries for the compiler.

Contents of **LINK_LIBRARIES** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. See the**cmak e−buildsystem(7)** manual for more on defining buildsystem properties.

NOTE:
A call to **target_link_libraries(<target> ...)** may update this property on **<target>**. If**<tar get>** was not created in the same directory as the call then **target_link_libraries()** will wrap each entry with the form **::@(directory−id);...;::@**, where the **::@** is literal and the **(directory−id)** is unspecified. This tells the generators that the named libraries must be looked up in the scope of the caller rather than in the scope in which the **<target>** was created. Valid directory ids are stripped on export by the **install(EXPORT)** and **export()** commands.

LINK_OPTIONS
New in version 3.13.

List of options to use for the link step of shared library, module and executable targets as well as the device link step. Targets that are static libraries need to use the **STATIC_LIBRARY_OPTIONS** target property.

These options are used for both normal linking and device linking (see policy **CMP0105**). To control link options for normal and device link steps, **$<HOST_LINK>** and **$<DEVICE_LINK>** **generator expressions** can be used.

This property holds a semicolon−separated list of options specified so far for its target. Use the**tar - get_link_options()** command to append more options.

This property is initialized by the **LINK_OPTIONS** directory property when a target is created, and is used by the generators to set the options for the compiler.

Contents of **LINK_OPTIONS** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. See the**cmak e−buildsystem(7)** manual for more on defining buildsystem properties.

NOTE:
This property must be used in preference to **LINK_FLAGS** property.

Host And Device Specific Link Options
New in version 3.18: When a device link step is involved, which is controlled by **CUDA_SEPARABLE_COMPILATION** and **CUDA_RESOLVE_DEVICE_SYMBOLS** properties and policy **CMP0105**, the raw options will be delivered to the host and device link steps (wrapped in **−Xcompiler** or equivalent for device link). Options wrapped with **$<DEVICE_LINK:...>** **generator expression** will be used only for the device link step. Options wrapped with **$<HOST_LINK:...>** **generator expression** will be used only for the host link step.

Option De−duplication
The final set of options used for a target is constructed by accumulating options from the current target and the usage requirements of its dependencies. The set of options is de−duplicated to avoid repetition.

New in version 3.12: While beneficial for individual options, the de−duplication step can break up option groups. For example, **−option A −option B** becomes **−option A B**. One may specify a group of options using shell−like quoting along with a **SHELL:** prefix. The **SHELL:** prefix is dropped, and the rest of the option string is parsed using the **separate_arguments() UNIX_COMMAND** mode. For example, **"SHELL:−option A" "SHELL:−option B"** becomes **−option A −option B**.

Handling Compiler Driver Differences
To pass options to the linker tool, each compiler driver has its own syntax. The **LINKER:** prefix and **,** separator can be used to specify, in a portable way, options to pass to the linker tool. **LINKER:** is replaced by the appropriate driver option and **,** by the appropriate driver separator. The driver prefix and driver separator are given by the values of the **CMAKE_<LANG>_LINKER_WRAPPER_FLAG** and **CMAKE_<LANG>_LINKER_WRAPPER_FLAG_SEP** variables.

For example, **"LINKER:−z,defs"** becomes **−Xlinker −z −Xlinker defs** for **Clang** and **−Wl,−z,defs** for **GNU GCC**.

The **LINKER:** prefix can be specified as part of a **SHELL:** prefix expression.

The **LINKER:** prefix supports, as an alternative syntax, specification of arguments using the **SHELL:** prefix and space as separator. The previous example then becomes **"LINKER:SHELL:−z defs"**.

NOTE:
Specifying the **SHELL:** prefix anywhere other than at the beginning of the **LINKER:** prefix is not supported.

LINK_SEARCH_END_STATIC
End a link line such that static system libraries are used.

Some linkers support switches such as **−Bstatic** and **−Bdynamic** to determine whether to use static or shared libraries for **−lXXX** options. CMake uses these options to set the link type for libraries whose full paths are not known or (in some cases) are in implicit link directories for the platform. By default CMake adds an option at the end of the library list (if necessary) to set the linker search type back to its starting type. This property switches the final linker search type to **−Bstatic** regardless of how it started.

This property is initialized by the value of the variable **CMAKE_LINK_SEARCH_END_STATIC** if it is set when a target is created.

See also **LINK_SEARCH_START_STATIC**.

LINK_SEARCH_START_STATIC
Assume the linker looks for static libraries by default.

Some linkers support switches such as **−Bstatic** and **−Bdynamic** to determine whether to use static or shared libraries for **−lXXX** options. CMake uses these options to set the link type for libraries whose full paths are not known or (in some cases) are in implicit link directories for the platform. By default the linker search type is assumed to be **−Bdynamic** at the beginning of the library list. This property switches the assumption to **−Bstatic**. It is intended for use when linking an executable statically (e.g. with the GNU **−static** option).

This property is initialized by the value of the variable
CMAKE_LINK_SEARCH_START_STATIC if it is set when a target is created.

See also **LINK_SEARCH_END_STATIC**.

LINK_WHAT_YOU_USE
New in version 3.7.

This is a boolean option that, when set to **TRUE**, will automatically run contents of variable **CMAKE_LINK_WHAT_YOU_USE_CHECK** on the target after it is linked. In addition, the linker flag specified by variable **CMAKE_<LANG>_LINK_WHAT_YOU_USE_FLAG** will be passed to the target with the link command so that all libraries specified on the command line will be linked into the target. This will result in the link producing a list of libraries that provide no symbols used by this target but are being linked to it.

NOTE:
For now, it is only supported for **ELF** platforms and is only applicable to executable and shared or module library targets. This property will be ignored for any other targets and configurations.

This property is initialized by the value of the **CMAKE_LINK_WHAT_YOU_USE** variable if it is set when a target is created.

LINKER_LANGUAGE
Specifies language whose compiler will invoke the linker.

For executables, shared libraries, and modules, this sets the language whose compiler is used to link the target (such as "C" or "CXX"). A typical value for an executable is the language of the source file providing the program entry point (main). If not set, the language with the highest linker preference value is the default. See documentation of**CMAKE_<LANG>_LINKER_PREFERENCE** v ariables.

If this property is not set by the user, it will be calculated at generate−time by CMake.

LOCATION
Read−only location of a target on disk.

For an imported target, this read−only property returns the value of the **LOCATION_<CONFIG>** property for an unspecified configuration **<CONFIG>** provided by the target.

For a non−imported target, this property is provided for compatibility with CMake 2.4 and below. It was meant to get the location of an executable target's output file for use in **add_custom_command()**. The path may contain a build−system−specific portion that is replaced at build time with the configuration getting built (such as **$(ConfigurationName)** in VS). In CMake 2.6 and above **add_custom_command()** automatically recognizes a target name in its **COMMAND** and **DEPENDS** options and computes the target location. In CMake 2.8.4 and above **add_custom_command()** recognizes **generator expressions** to refer to target locations anywhere in the command. Therefore this property is not needed for creating custom commands.

Do not set properties that affect the location of a target after reading this property. These include properties whose names match **(RUNTIME|LIBRARY|ARCHIVE)_OUTPUT_(NAME|DIRECTORY)(_<CONFIG>)?**, **(IMPLIB_)?(PREFIX|SUFFIX)**, or "LINKER_LANGUAGE". Failure to follow this rule is not diagnosed and leaves the location of the target undefined.

LOCATION_<CONFIG>
Read−only property providing a target location on disk.

A read−only property that indicates where a target's main file is located on disk for the configuration **<CONFIG>**. The property is defined only for library and executable targets. An imported target may provide a set of configurations different from that of the importing project. By default CMake looks for an exact−match but otherwise uses an arbitrary available configuration. Use the **MAP_IMPORTED_CONFIG_<CONFIG>** property to map imported configurations explicitly.

Do not set properties that affect the location of a target after reading this property. These include properties whose names match **(RUNTIME|LIBRARY|ARCHIVE)_OUTPUT_(NAME|DIRECTORY)(_<CONFIG>)?**, **(IMPLIB_)?(PREFIX|SUFFIX)**, or **LINKER_LANGU AGE**. Failure to follow this rule is not diagnosed and leaves the location of the target undefined.

MACHO_COMPATIBILITY_VERSION
New in version 3.17.

What compatibility version number is this target for Mach−O binaries.

For shared libraries on Mach−O systems (e.g. macOS, iOS) the **MACHO_COMPATIBILITY_VERSION** property corresponds to the *compatibility version* and **MACHO_CURRENT_VERSION** corresponds to the *current version*. These are both embedded in the shared library binary and can be checked with the **otool −L <binary>** command.

It should be noted that the **MACHO_CURRENT_VERSION** and **MACHO_COMPATIBILITY_VERSION** properties do not affect the file names or version−related symlinks that CMake generates for the library. The**VERSION** and **SO VERSION** target properties still control the file and symlink names. The **install_name** is also still controlled by **SOVERSION**.

When **MACHO_CURRENT_VERSION** and **MACHO_COMPATIBILITY_VERSION** are not given, **VERSION** and **SOVERSION** are used for the version details to be embedded in the binaries respectively. The **MACHO_CURRENT_VERSION** and **MACHO_COMPATIBILITY_VERSION** properties only need to be given if the project needs to decouple the file and symlink naming from the version details embedded in the binaries (e.g. to match libtool conventions).

MACHO_CURRENT_VERSION
New in version 3.17.

What current version number is this target for Mach−O binaries.

For shared libraries on Mach−O systems (e.g. macOS, iOS) the **MACHO_COMPATIBILITY_VERSION** property corresponds to the *compatibility version* and **MACHO_CURRENT_VERSION** corresponds to the *current version*. These are both embedded in the shared library binary and can be checked with the **otool −L <binary>** command.

It should be noted that the **MACHO_CURRENT_VERSION** and **MACHO_COMPATIBILITY_VERSION** properties do not affect the file names or version−related symlinks that CMake generates for the library. The**VERSION** and **SO VERSION** target properties still control the file and symlink names. The **install_name** is also still controlled by **SOVERSION**.

When **MACHO_CURRENT_VERSION** and **MACHO_COMPATIBILITY_VERSION** are not given, **VERSION** and **SOVERSION** are used for the version details to be embedded in the binaries respectively. The **MACHO_CURRENT_VERSION** and **MACHO_COMPATIBILITY_VERSION** properties only need to be given if the project needs to decouple the file and symlink naming from the version details embedded in the binaries (e.g. to match libtool conventions).

MACOSX_BUNDLE
Build an executable as an Application Bundle on macOS or iOS.

When this property is set to **TRUE** the executable when built on macOS or iOS will be created as an application bundle. This makes it a GUI executable that can be launched from the Finder. See the**MACOSX_BUNDLE_INFO_PLIST** target property for information about creation of the **Info.plist** file for the application bundle. This property is initialized by the value of the variable **CMAKE_MACOSX_BUNDLE** if it is set when a target is created.

MACOSX_BUNDLE_INFO_PLIST
Specify a custom **Info.plist** template for a macOS and iOS Application Bundle.

An executable target with **MACOSX_BUNDLE** enabled will be built as an application bundle on macOS. By default its **Info.plist** file is created by configuring a template called **MacOSXBundleInfo.plist.in** located in the **CMAKE_MODULE_PATH**. This property specifies an alternative template file name which may be a full path.

The following target properties may be set to specify content to be configured into the file:

MACOSX_BUNDLE_BUNDLE_NAME
Sets **CFBundleName**.

MACOSX_BUNDLE_BUNDLE_VERSION
Sets **CFBundleVersion**.

MACOSX_BUNDLE_COPYRIGHT
Sets **NSHumanReadableCopyright**.

MACOSX_BUNDLE_GUI_IDENTIFIER
Sets **CFBundleIdentifier**.

MACOSX_BUNDLE_ICON_FILE
Sets **CFBundleIconFile**.

MACOSX_BUNDLE_INFO_STRING
Sets **CFBundleGetInfoString**.

MACOSX_BUNDLE_LONG_VERSION_STRING
Sets **CFBundleLongVersionString**.

MACOSX_BUNDLE_SHORT_VERSION_STRING
Sets **CFBundleShortVersionString**.

CMake variables of the same name may be set to affect all targets in a directory that do not have each specific property set. If a custom **Info.plist** is specified by this property it may of course hard−code all the settings instead of using the target properties.

MACOSX_FRAMEWORK_INFO_PLIST
Specify a custom **Info.plist** template for a macOS and iOS Framework.

A library target with **FRAMEWORK** enabled will be built as a framework on macOS. By default its **Info.plist** file is created by configuring a template called **MacOSXFrameworkInfo.plist.in** located in the **CMAKE_MODULE_PATH**. This property specifies an alternative template file name which may be a full path.

The following target properties may be set to specify content to be configured into the file:

MACOSX_FRAMEWORK_BUNDLE_VERSION
Sets **CFBundleVersion**.

MACOSX_FRAMEWORK_ICON_FILE
Sets **CFBundleIconFile**.

MACOSX_FRAMEWORK_IDENTIFIER
Sets **CFBundleIdentifier**.

MACOSX_FRAMEWORK_SHORT_VERSION_STRING
Sets **CFBundleShortVersionString**.

CMake variables of the same name may be set to affect all targets in a directory that do not have each specific property set. If a custom **Info.plist** is specified by this property it may of course hard−code all the settings instead of using the target properties.

MACOSX_RPATH
Whether this target on macOS or iOS is located at runtime using rpaths.

When this property is set to **TRUE**, the directory portion of the **install_name** field of this shared library will be **@rpath** unless overridden by **INSTALL_NAME_DIR**. This indicates the shared library is to be found at runtime using runtime paths (rpaths).

This property is initialized by the value of the variable **CMAKE_MACOSX_RPATH** if it is set when a target is created.

Runtime paths will also be embedded in binaries using this target and can be controlled by the **INSTALL_RPATH** target property on the target linking to this target.

Policy **CMP0042** was introduced to change the default value of **MACOSX_RPATH** to **TRUE**. This is because use of **@rpath** is a more flexible and powerful alternative to **@executable_path** and **@loader_path**.

MANUALLY_ADDED_DEPENDENCIES
New in version 3.8.

Get manually added dependencies to other top−level targets.

This read−only property can be used to query all dependencies that were added for this target with the **add_dependencies()** command.

MAP_IMPORTED_CONFIG_<CONFIG>
Map from project configuration to imported target's configuration.

Set this to the list of configurations of an imported target that may be used for the current project's **<CONFIG>** configuration. Targets imported from another project may not provide the same set of configuration names available in the current project. Setting this property tells CMake what imported configurations are suitable for use when building the **<CONFIG>** configuration. The first configuration in the list found to be provided by the imported target (i.e. via **IMPORTED_LOCATION_<CONFIG>** for the mapped−to **<CONFIG>**) is selected. As a special case, an empty list element refers to the configuration−less imported target location (i.e. **IMPORTED_LOCATION**).

If this property is set and no matching configurations are available, then the imported target is considered to be not found. This property is ignored for non−imported targets.

This property is initialized by the value of the **CMAKE_MAP_IMPORTED_CONFIG_<CONFIG>** variable if it is set when a target is created.

Example
For example creating imported C++ library **foo**:

```
add_library(foo STATIC IMPORTED)
```

Use **foo_debug** path for **Debug** build type:

```
set_property(
  TARGET foo APPEND PROPERTY IMPORTED_CONFIGURATIONS DEBUG
  )

set_target_properties(foo PROPERTIES
  IMPORTED_LINK_INTERFACE_LANGUAGES_DEBUG "CXX"
  IMPORTED_LOCATION_DEBUG "${foo_debug}"
  )
```

Use **foo_release** path for **Release** build type:

```
set_property(
  TARGET foo APPEND PROPERTY IMPORTED_CONFIGURATIONS RELEASE
  )

set_target_properties(foo PROPERTIES
  IMPORTED_LINK_INTERFACE_LANGUAGES_RELEASE "CXX"
  IMPORTED_LOCATION_RELEASE "${foo_release}"
  )
```

Use **Release** version of library for **MinSizeRel** and **RelWithDebInfo** build types:

```
set_target_properties(foo PROPERTIES
```

```
MAP_IMPORTED_CONFIG_MINSIZEREL Release
MAP_IMPORTED_CONFIG_RELWITHDEBINFO Release
)
```

MSVC_RUNTIME_LIBRARY
New in version 3.15.

Select the MSVC runtime library for use by compilers targeting the MSVC ABI.

The allowed values are:

MultiThreaded
Compile with **-MT** or equivalent flag(s) to use a multi-threaded statically-linked runtime library.

MultiThreadedDLL
Compile with **-MD** or equivalent flag(s) to use a multi-threaded dynamically-linked runtime library.

MultiThreadedDebug
Compile with **-MTd** or equivalent flag(s) to use a multi-threaded statically-linked runtime library.

MultiThreadedDebugDLL
Compile with **-MDd** or equivalent flag(s) to use a multi-threaded dynamically-linked runtime library.

The value is ignored on non-MSVC compilers but an unsupported value will be rejected as an error when using a compiler targeting the MSVC ABI.

The value may also be the empty string ("") in which case no runtime library selection flag will be added explicitly by CMake. Note that with Visual Studio Generators the native build system may choose to add its own default runtime library selection flag.

Use **generator expressions** to support per-configuration specification. For example, the code:

```
add_executable(foo foo.c)
set_property(TARGET foo PROPERTY
  MSVC_RUNTIME_LIBRARY "MultiThreaded$<$<CONFIG:Debug>:Debug>")
```

selects for the target **foo** a multi-threaded statically-linked runtime library with or without debug information depending on the configuration.

If this property is not set then CMake uses the default value **MultiThreaded$<$<CONFIG:Debug>:Debug>DLL** to select a MSVC runtime library.

NOTE:
This property has effect only when policy **CMP0091** is set to **NEW** prior to the first **project()** or **enable_language()** command that enables a language using a compiler targeting the MSVC ABI.

NAME
Logical name for the target.

Read-only logical name for the target as used by CMake.

NO_SONAME
Whether to set **soname** when linking a shared library.

Enable this boolean property if a generated **SHARED** library should not have **soname** set. Default is to set

soname on all shared libraries as long as the platform supports it. Generally, use this property only for leaf private libraries or plugins. If you use it on normal shared libraries which other targets link against, on some platforms a linker will insert a full path to the library (as specified at link time) into the dynamic section of the dependent binary. Therefore, once installed, dynamic loader may eventually fail to locate the library for the binary.

NO_SYSTEM_FROM_IMPORTED
Do not treat include directories from the interfaces of consumed imported targets as **SYSTEM**.

The contents of the **INTERFACE_INCLUDE_DIRECTORIES** target property of imported targets are treated as **SYSTEM** includes by default. If this property is enabled on a target, compilation of sources in that target will not treat the contents of the **INTERFACE_INCLUDE_DIRECTORIES** of consumed imported targets as system includes.

This property is initialized by the value of the **CMAKE_NO_SYSTEM_FROM_IMPORTED** variable if it is set when a target is created.

OBJC_EXTENSIONS
New in version 3.16.

Boolean specifying whether compiler specific extensions are requested.

This property specifies whether compiler specific extensions should be used. For some compilers, this results in adding a flag such as **−std=gnu11** instead of **−std=c11** to the compile line. This property is **ON** by default. The basic OBJC standard level is controlled by the **OBJC_STANDARD** target property.

If the property is not set, and the project has set the **C_EXTENSIONS**, the value of **C_EXTENSIONS** is set for *OBJC_EXTENSIONS*.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_OBJC_EXTENSIONS** variable if set when a target is created and otherwise by the value of **CMAKE_OBJC_EXTENSIONS_DEFAULT** (see **CMP0128**).

OBJC_STANDARD
New in version 3.16.

The OBJC standard whose features are requested to build this target.

This property specifies the OBJC standard whose features are requested to build this target. For some compilers, this results in adding a flag such as **−std=gnu11** to the compile line.

Supported values are:

90 Objective C89/C90

99 Objective C99

11 Objective C11

If the value requested does not result in a compile flag being added for the compiler in use, a previous standard flag will be added instead. This means that using:

```
set_property(TARGET tgt PROPERTY OBJC_STANDARD 11)
```

with a compiler which does not support **−std=gnu11** or an equivalent flag will not result in an error or warning, but will instead add the **−std=gnu99** or **−std=gnu90** flag if supported. This "decay" behavior may be controlled with the **OBJC_STANDARD_REQUIRED** target property. Additionally, the **OBJC_EXTENSIONS** target property may be used to control whether compiler−specific extensions are enabled on a per−target basis.

If the property is not set, and the project has set the **C_STANDARD**, the value of **C_STANDARD** is set for *OBJC_STANDARD*.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_OBJC_STANDARD** variable if it is set when a target is created.

OBJC_STANDARD_REQUIRED
New in version 3.16.

Boolean describing whether the value of **OBJC_STANDARD** is a requirement.

If this property is set to **ON**, then the value of the **OBJC_STANDARD** target property is treated as a requirement. If this property is**OFF** or unset, the **OBJC_ST ANDARD** target property is treated as optional and may "decay" to a previous standard if the requested is not available.

If the property is not set, and the project has set the **C_STANDARD_REQUIRED**, the value of **C_STANDARD_REQUIRED** is set for *OBJC_STANDARD_REQUIRED*.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_OBJC_STANDARD_REQUIRED** variable if it is set when a target is created.

OBJCXX_EXTENSIONS
New in version 3.16.

Boolean specifying whether compiler specific extensions are requested.

This property specifies whether compiler specific extensions should be used. For some compilers, this results in adding a flag such as **−std=gnu++11** instead of **−std=c++11** to the compile line. This property is **ON** by default. The basic ObjC++ standard level is controlled by the **OBJCXX_STANDARD** target property.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

If the property is not set, and the project has set the **CXX_EXTENSIONS**, the value of **CXX_EXTENSIONS** is set for *OBJCXX_EXTENSIONS*.

This property is initialized by the value of the **CMAKE_OBJCXX_EXTENSIONS** variable if set when a target is created and otherwise by the value of **CMAKE_OBJCXX_EXTENSIONS_DEFAULT** (see **CMP0128**).

OBJCXX_STANDARD
New in version 3.16.

The ObjC++ standard whose features are requested to build this target.

This property specifies the ObjC++ standard whose features are requested to build this target. For some compilers, this results in adding a flag such as **−std=gnu++11** to the compile line.

Supported values are:

98 Objective C++98

11 Objective C++11

14 Objective C++14

17 Objective C++17

20 Objective C++20

23 New in version 3.20.

Objective C++23

If the value requested does not result in a compile flag being added for the compiler in use, a previous standard flag will be added instead. This means that using:

```
set_property(TARGET tgt PROPERTY OBJCXX_STANDARD 11)
```

with a compiler which does not support **−std=gnu++11** or an equivalent flag will not result in an error or warning, but will instead add the **−std=gnu++98** flag if supported. This "decay" behavior may be controlled with the **OBJCXX_STANDARD_REQUIRED** target property. Additionally, the **OBJCXX_EXTENSIONS** target property may be used to control whether compiler−specific extensions are enabled on a per−target basis.

If the property is not set, and the project has set the **CXX_STANDARD**, the value of **CXX_STANDARD** is set for *OBJCXX_STANDARD*.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_OBJCXX_STANDARD** variable if it is set when a target is created.

OBJCXX_STANDARD_REQUIRED
New in version 3.16.

Boolean describing whether the value of **OBJCXX_STANDARD** is a requirement.

If this property is set to**ON**, then the v alue of the **OBJCXX_STANDARD** target property is treated as a requirement. If this property is**OFF** or unset, the **OBJCXX_ST ANDARD** target property is treated as optional and may "decay" to a previous standard if the requested is not available.

If the property is not set, and the project has set the **CXX_STANDARD_REQUIRED**, the value of **CXX_STANDARD_REQUIRED** is set for *OBJCXX_STANDARD_REQUIRED*.

See the **cmake−compile−features(7)** manual for information on compile features and a list of supported compilers.

This property is initialized by the value of the **CMAKE_OBJCXX_STANDARD_REQUIRED** variable if it is set when a target is created.

OPTIMIZE_DEPENDENCIES
New in version 3.19.

Activates dependency optimization of static and object libraries.

When this property is set to true, some dependencies for a static or object library may be removed at generation time if they are not necessary to build the library, since static and object libraries don't actually link against anything.

If a static or object library has dependency optimization enabled, it first discards all dependencies. Then, it looks through all of the direct and indirect dependencies that it initially had, and adds them back if they meet any of the following criteria:

- The dependency was added to the library by **add_dependencies()**.
- The dependency was added to the library through a source file in the library generated by a custom command that uses the dependency.
- The dependency has any **PRE_BUILD**, **PRE_LINK**, or **POST_BUILD** custom commands associated with it.
- The dependency contains any source files that were generated by a custom command.
- The dependency contains any languages which produce side effects that are relevant to the library. Currently, all languages except C, C++, Objective−C, Objective−C++, assembly, and CUDA are assumed to produce side effects. However, side effects from one language are assumed not to be relevant to another (for example, a Fortran library is assumed to not have any side effects that are relevant for a Swift library.)

As an example, assume you have a static Fortran library which depends on a static C library, which in turn depends on a static Fortran library. The top−level Fortran library has optimization enabled, but the middle C library does not. If you build the top Fortran library, the bottom Fortran library will also build, but not the middle C library, since the C library does not have any side effects that are relevant for the Fortran library. However, if you build the middle C library, the bottom Fortran library will also build, even though it does not have any side effects that are relevant to the C library, since the C library does not have optimization enabled.

This property is initialized by the value of the **CMAKE_OPTIMIZE_DEPENDENCIES** variable when the target is created.

OSX_ARCHITECTURES
Target specific architectures for macOS.

The **OSX_ARCHITECTURES** property sets the target binary architecture for targets on macOS (**−arch**). This property is initialized by the value of the variable **CMAKE_OSX_ARCHITECTURES** if it is set when a target is created. Use **OSX_ARCHITECTURES_<CONFIG>** to set the binary architectures on a per−configuration basis, where **<CONFIG>** is an upper−case name (e.g. **OSX_ARCHITECTURES_DEBUG**).

OSX_ARCHITECTURES_<CONFIG>
Per−configuration macOS and iOS binary architectures for a target.

This property is the configuration−specific version of **OSX_ARCHITECTURES**.

OUTPUT_NAME
Output name for target files.

This sets the base name for output files created for an executable or library target. If not set, the logical target name is used by default during generation. The value is not set by default during configuration.

Contents of **OUTPUT_NAME** and the variants listed below may use **generator expressions**.

See also the variants:

- **OUTPUT_NAME_<CONFIG>**
- **ARCHIVE_OUTPUT_NAME_<CONFIG>**
- **ARCHIVE_OUTPUT_NAME**
- **LIBRARY_OUTPUT_NAME_<CONFIG>**
- **LIBRARY_OUTPUT_NAME**
- **RUNTIME_OUTPUT_NAME_<CONFIG>**
- **RUNTIME_OUTPUT_NAME**

OUTPUT_NAME_<CONFIG>
Per−configuration target file base name.

This is the configuration−specific version of the **OUTPUT_NAME** target property.

PCH_WARN_INVALID
New in version 3.18.

When this property is set to true, the precompile header compiler options will contain a compiler flag which should warn about invalid precompiled headers e.g. **−Winvalid−pch** for GNU compiler.

This property is initialized by the value of the **CMAKE_PCH_WARN_INVALID** variable if it is set when a target is created. If that variable is not set, the property defaults to **ON**.

PCH_INSTANTIATE_TEMPLATES
New in version 3.19.

When this property is set to true, the precompiled header compiler options will contain a flag to instantiate templates during the generation of the PCH if supported. This can significantly improve compile times. Supported in Clang since version 11.

This property is initialized by the value of the **CMAKE_PCH_INSTANTIATE_TEMPLATES** variable if it is set when a target is created. If that variable is not set, the property defaults to **ON**.

PDB_NAME
Output name for the MS debug symbol **.pdb** file generated by the linker for an executable or shared library target.

This property specifies the base name for the debug symbols file. If not set, the **OUTPUT_NAME** target property value or logical target name is used by default.

NOTE:
This property does not apply to STATIC library targets because no linker is invoked to produce them so they have no linker−generated **.pdb** file containing debug symbols.

The linker−generated program database files are specified by the **/pdb** linker flag and are not the same as compiler−generated program database files specified by the **/Fd** compiler flag. Use the **COMPILE_PDB_NAME** property to specify the latter.

PDB_NAME_<CONFIG>
Per−configuration output name for the MS debug symbol **.pdb** file generated by the linker for an executable or shared library target.

This is the configuration−specific version of **PDB_NAME**.

NOTE:
This property does not apply to STATIC library targets because no linker is invoked to produce them so they have no linker−generated **.pdb** file containing debug symbols.

The linker−generated program database files are specified by the **/pdb** linker flag and are not the same as compiler−generated program database files specified by the **/Fd** compiler flag. Use the **COMPILE_PDB_NAME_<CONFIG>** property to specify the latter.

PDB_OUTPUT_DIRECTORY
Output directory for the MS debug symbols **.pdb** file generated by the linker for an executable or shared library target.

This property specifies the directory into which the MS debug symbols will be placed by the linker. The property value may use **generator expressions**. Multi−configuration generators append a per−configuration subdirectory to the specified directory unless a generator expression is used.

This property is initialized by the value of the **CMAKE_PDB_OUTPUT_DIRECTORY** variable if it is set when a target is created.

NOTE:
This property does not apply to STATIC library targets because no linker is invoked to produce them so they have no linker−generated **.pdb** file containing debug symbols.

The linker−generated program database files are specified by the **/pdb** linker flag and are not the same as compiler−generated program database files specified by the **/Fd** compiler flag. Use the **COMPILE_PDB_OUTPUT_DIRECTORY** property to specify the latter.

PDB_OUTPUT_DIRECTORY_<CONFIG>
Per−configuration output directory for the MS debug symbol **.pdb** file generated by the linker for an executable or shared library target.

This is a per−configuration version of **PDB_OUTPUT_DIRECTORY**, but multi−configuration generators (Visual Studio Generators, **Xcode**) do NOT append a per−configuration subdirectory to the specified directory. This property is initialized by the value of the **CMAKE_PDB_OUTPUT_DIRECTORY_<CONFIG>** variable if it is set when a target is created.

Contents of **PDB_OUTPUT_DIRECTORY_<CONFIG>** may use **generator expressions**.

NOTE:
This property does not apply to STATIC library targets because no linker is invoked to produce them so they have no linker−generated **.pdb** file containing debug symbols.

The linker−generated program database files are specified by the **/pdb** linker flag and are not the same as compiler−generated program database files specified by the **/Fd** compiler flag. Use the **COMPILE_PDB_OUTPUT_DIRECTORY_<CONFIG>** property to specify the latter.

POSITION_INDEPENDENT_CODE
Whether to create a position−independent target

The **POSITION_INDEPENDENT_CODE** property determines whether position independent executables or shared libraries will be created. This property is **True** by default for **SHARED** and **MODULE** library targets and **False** otherwise. This property is initialized by the value of the **CMAKE_POSITION_INDEPENDENT_CODE** variable if it is set when a target is created.

NOTE:
For executable targets, the link step is controlled by the **CMP0083** policy and the **CheckPIESupported** module.

PRECOMPILE_HEADERS
New in version 3.16.

List of header files to precompile.

This property holds a semicolon−separated list of header files to precompile specified so far for its target. Use the **target_precompile_headers()** command to append more header files.

This property supports **generator expressions**.

PRECOMPILE_HEADERS_REUSE_FROM
New in version 3.16.

Target from which to reuse the precompiled headers build artifact.

See the second signature of **target_precompile_headers()** command for more detailed information.

PREFIX
What comes before the library name.

A target property that can be set to override the prefix (such as **lib**) on a library name.

PRIVATE_HEADER
Specify private header files in a **FRAMEWORK** shared library target.

Shared library targets marked with the **FRAMEWORK** property generate frameworks on macOS, iOS and normal shared libraries on other platforms. This property may be set to a list of header files to be placed in the PrivateHeaders directory inside the framework folder. On non−Apple platforms these headers may be installed using the **PRIVATE_HEADER** option to the **install(TARGETS)** command.

PROJECT_LABEL
Change the name of a target in an IDE.

Can be used to change the name of the target in an IDE like Visual Studio.

PUBLIC_HEADER
Specify public header files in a **FRAMEWORK** shared library target.

Shared library targets marked with the **FRAMEWORK** property generate frameworks on macOS, iOS and normal shared libraries on other platforms. This property may be set to a list of header files to be placed in the **Headers** directory inside the framework folder. On non−Apple platforms these headers may be installed using the **PUBLIC_HEADER** option to the **install(TARGETS)** command.

RESOURCE

Specify resource files in a **FRAMEWORK** or **BUNDLE**.

Target marked with the **FRAMEWORK** or **BUNDLE** property generate framework or application bundle (both macOS and iOS is supported) or normal shared libraries on other platforms. This property may be set to a list of files to be placed in the corresponding directory (eg. **Resources** directory for macOS) inside the bundle. On non−Apple platforms these files may be installed using the**RESOURCE** option to the **install(TARGETS)** command.

Following example of Application Bundle:

```
add_executable(ExecutableTarget
  addDemo.c
  resourcefile.txt
  appresourcedir/appres.txt)

target_link_libraries(ExecutableTarget heymath mul)

set(RESOURCE_FILES
  resourcefile.txt
  appresourcedir/appres.txt)

set_target_properties(ExecutableTarget PROPERTIES
  MACOSX_BUNDLE TRUE
  MACOSX_FRAMEWORK_IDENTIFIER org.cmake.ExecutableTarget
  RESOURCE "${RESOURCE_FILES}")
```

will produce flat structure for iOS systems:

```
ExecutableTarget.app
  appres.txt
  ExecutableTarget
  Info.plist
  resourcefile.txt
```

For macOS systems it will produce following directory structure:

```
ExecutableTarget.app/
  Contents
    Info.plist
    MacOS
      ExecutableTarget
    Resources
      appres.txt
      resourcefile.txt
```

For Linux, such CMake script produce following files:

```
ExecutableTarget
Resources
  appres.txt
  resourcefile.txt
```

RULE_LAUNCH_COMPILE
Specify a launcher for compile rules.

See the global property of the same name for details. This overrides the global and directory property for a target.

RULE_LAUNCH_CUSTOM
Specify a launcher for custom rules.

See the global property of the same name for details. This overrides the global and directory property for a target.

RULE_LAUNCH_LINK
Specify a launcher for link rules.

See the global property of the same name for details. This overrides the global and directory property for a target.

RUNTIME_OUTPUT_DIRECTORY
Output directory in which to build RUNTIME target files.

This property specifies the directory into which runtime target files should be built. The property value may use **generator expressions**. Multi−configuration generators (Visual Studio, **Xcode**, **Ninja Multi−Config**) append a per−configuration subdirectory to the specified directory unless a generator expression is used.

This property is initialized by the value of the **CMAKE_RUNTIME_OUTPUT_DIRECTORY** variable if it is set when a target is created.

See also the **RUNTIME_OUTPUT_DIRECTORY_<CONFIG>** target property.

RUNTIME_OUTPUT_DIRECTORY_<CONFIG>
Per−configuration output directory for RUNTIME target files.

This is a per−configuration version of the **RUNTIME_OUTPUT_DIRECTORY** target property, but multi−configuration generators (Visual Studio Generators, **Xcode**) do NOT append a per−configuration subdirectory to the specified directory. This property is initialized by the value of the **CMAKE_RUNTIME_OUTPUT_DIRECTORY_<CONFIG>** variable if it is set when a target is created.

Contents of **RUNTIME_OUTPUT_DIRECTORY_<CONFIG>** may use **generator expressions**.

RUNTIME_OUTPUT_NAME
Output name for RUNTIME target files.

This property specifies the base name for runtime target files. It overrides **OUTPUT_NAME** and **OUTPUT_NAME_<CONFIG>** properties.

See also the **RUNTIME_OUTPUT_NAME_<CONFIG>** target property.

RUNTIME_OUTPUT_NAME_<CONFIG>
Per−configuration output name for RUNTIME target files.

This is the configuration−specific version of the **RUNTIME_OUTPUT_NAME** target property.

SKIP_BUILD_RPATH
Should rpaths be used for the build tree.

SKIP_BUILD_RPATH is a boolean specifying whether to skip automatic generation of an rpath allowing the target to run from the build tree. This property is initialized by the value of the variable

CMAKE_SKIP_BUILD_RPATH if it is set when a target is created.

SOURCE_DIR
New in version 3.4.

This read−only property reports the value of the **CMAKE_CURRENT_SOURCE_DIR** variable in the directory in which the target was defined.

SOURCES
Source names specified for a target.

List of sources specified for a target.

SOVERSION
What version number is this target.

For shared libraries **VERSION** and **SOVERSION** can be used to specify the build version and API version respectively. When building or installing appropriate symlinks are created if the platform supports symlinks and the linker supports so−names. If only one of both is specified the missing is assumed to have the same version number. **SOVERSION** is ignored if **NO_SONAME** property is set.

Windows Versions
For shared libraries and executables on Windows the **VERSION** attribute is parsed to extract a **<major>.<minor>** version number. These numbers are used as the image version of the binary.

Mach−O Versions
For shared libraries and executables on Mach−O systems (e.g. macOS, iOS), the **SOVERSION** property corresponds to the *compatibility version* and **VERSION** corresponds to the *current version* (unless Mach−O specific overrides are provided, as discussed below). See the**FRAMEW ORK** target property for an example.

For shared libraries, the **MACHO_COMPATIBILITY_VERSION** and **MACHO_CURRENT_VERSION** properties can be used to override the *compatibility version* and *current version* respectively. Note that **SOVERSION** will still be used to form the **install_name** and both **SOVERSION** and **VERSION** may also affect the file and symlink names.

Versions of Mach−O binaries may be checked with the **otool −L <binary>** command.

STATIC_LIBRARY_FLAGS
Archiver (or MSVC librarian) flags for a static library target. Targets that are shared libraries, modules, or executables need to use the **LINK_OPTIONS** or **LINK_FLAGS** target properties.

The **STATIC_LIBRARY_FLAGS** property, managed as a string, can be used to add extra flags to the link step of a static library target. **STATIC_LIBRARY_FLAGS_<CONFIG>** will add to the configuration **<CONFIG>**, for example, **DEBUG**, **RELEASE**, **MINSIZEREL**, **RELWITHDEBINFO**, ...

NOTE:
This property has been superseded by **STATIC_LIBRARY_OPTIONS** property.

STATIC_LIBRARY_FLAGS_<CONFIG>
Per−configuration archiver (or MSVC librarian) flags for a static library target.

This is the configuration−specific version of **STATIC_LIBRARY_FLAGS**.

NOTE:
This property has been superseded by **STATIC_LIBRARY_OPTIONS** property.

STATIC_LIBRARY_OPTIONS
New in version 3.13.

Archiver (or MSVC librarian) flags for a static library target. Targets that are shared libraries, modules, or executables need to use the **LINK_OPTIONS** target property.

This property holds a semicolon−separated list of options specified so far for its target. Use**set_tar - get_properties()** or **set_property()** commands to set its content.

Contents of **STATIC_LIBRARY_OPTIONS** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. See the**cmak e−buildsystem(7)** manual for more on defining buildsystem properties.

NOTE:
This property must be used in preference to **STATIC_LIBRARY_FLAGS** property.

Option De−duplication
The final set of options used for a target is constructed by accumulating options from the current target and the usage requirements of its dependencies. The set of options is de−duplicated to avoid repetition.

New in version 3.12: While beneficial for individual options, the de−duplication step can break up option groups. For example, **−option A −option B** becomes **−option A B**. One may specify a group of options using shell−like quoting along with a **SHELL:** prefix. The **SHELL:** prefix is dropped, and the rest of the option string is parsed using the **separate_arguments() UNIX_COMMAND** mode. For example, **"SHELL:−option A" "SHELL:−option B"** becomes **−option A −option B**.

SUFFIX
What comes after the target name.

A target property that can be set to override the suffix (such as **.so** or **.exe**) on the name of a library, module or executable.

Swift_DEPENDENCIES_FILE
New in version 3.15.

This property sets the path for the Swift dependency file (swiftdep) for the target. If one is not specified, it will default to **<TARGET>.swiftdeps**.

Swift_LANGUAGE_VERSION
New in version 3.16.

This property sets the language version for the Swift sources in the target. If one is not specified, it will default to **<CMAKE_Swift_LANGUAGE_VERSION>** if specified, otherwise it is the latest version supported by the compiler.

Swift_MODULE_DIRECTORY
New in version 3.15.

Specify output directory for Swift modules provided by the target.

If the target contains Swift source files, this specifies the directory in which the modules will be placed. When this property is not set, the modules will be placed in the build directory corresponding to the target's

source directory. If the variable **CMAKE_Swift_MODULE_DIRECTORY** is set when a target is created its value is used to initialize this property.

Swift_MODULE_NAME
New in version 3.15.

This property specifies the name of the Swift module. It is defaulted to the name of the target.

TYPE
The type of the target.

This read−only property can be used to test the type of the given target. It will be one of**ST ATIC_LIBRARY**, **MODULE_LIBRARY**, **SHARED_LIBRARY**, **OBJECT_LIBRARY**, **INTERFACE_LIBRARY**, **EXECUTABLE** or one of the internal target types.

UNITY_BUILD
New in version 3.16.

When this property is set to true, the target source files will be combined into batches for faster compilation. This is done by creating a (set of) unity sources which**#include** the original sources, then compiling these unity sources instead of the originals. This is known as a *Unity* or *Jumbo* build.

CMake provides different algorithms for selecting which sources are grouped together into a *bucket*. Algorithm selection is decided by the **UNITY_BUILD_MODE** target property, which has the following acceptable values:

- **BATCH** When in this mode CMake determines which files are grouped together. The **UNITY_BUILD_BATCH_SIZE** property controls the upper limit on how many sources can be combined per unity source file.
- **GROUP** When in this mode each target explicitly specifies how to group source files. Each source file that has the same **UNITY_GROUP** value will be grouped together. Any sources that don't have this property will be compiled individually. The **UNITY_BUILD_BATCH_SIZE** property is ignored when using this mode.

If no explicit **UNITY_BUILD_MODE** has been specified, CMake will default to **BATCH**.

Unity builds are not currently supported for all languages. CMake version 3.22.1 supports combining **C** and **CXX** source files. For targets that mix source files from more than one language, CMake will separate the languages such that each generated unity source file only contains sources for a single language.

This property is initialized by the value of the **CMAKE_UNITY_BUILD** variable when a target is created.

NOTE:
Projects should not directly set the **UNITY_BUILD** property or its associated **CMAKE_UNITY_BUILD** variable to true. Depending on the capabilities of the build machine and compiler used, it might or might not be appropriate to enable unity builds. Therefore, this feature should be under developer control, which would normally be through the developer choosing whether or not to set the **CMAKE_UNITY_BUILD** variable on the **cmake(1)** command line or some other equivalent method. However, it IS recommended to set the **UNITY_BUILD** target property to false if it is known that enabling unity builds for the target can lead to problems.

ODR (One definition rule) errors
When multiple source files are included into one source file, as is done for unity builds, it can potentially lead to ODR errors. CMake provides a number of measures to help address such problems:

- Any source file that has a non−empty **COMPILE_OPTIONS**, **COMPILE_DEFINITIONS**, **COMPILE_FLAGS**, or **INCLUDE_DIRECTORIES** source property will not be combined into a unity source.
- Projects can prevent an individual source file from being combined into a unity source by setting its **SKIP_UNITY_BUILD_INCLUSION** source property to true. This can be a more effective way to prevent problems with specific files than disabling unity builds for an entire target.
- Projects can set **UNITY_BUILD_UNIQUE_ID** to cause a valid C−identifier to be generated which is unique per file in a unity build. This can be used to avoid problems with anonymous namespaces in unity builds.
- The **UNITY_BUILD_CODE_BEFORE_INCLUDE** and **UNITY_BUILD_CODE_AFTER_INCLUDE** target properties can be used to inject code into the unity source files before and after every **#include** statement.
- The order of source files added to the target via commands like **add_library()**, **add_executable()** or **target_sources()** will be preserved in the generated unity source files. This can be used to manually enforce a specific grouping based on the **UNITY_BUILD_BATCH_SIZE** target property.

UNITY_BUILD_BATCH_SIZE
New in version 3.16.

Specifies the maximum number of source files that can be combined into any one unity source file when unity builds are enabled by the **UNITY_BUILD** target property. The original source files will be distributed across as many unity source files as necessary to honor this limit.

The initial value for this property is taken from the **CMAKE_UNITY_BUILD_BATCH_SIZE** variable when the target is created. If that variable has not been set, the initial value will be 8.

The batch size needs to be selected carefully. If set too high, the size of the combined source files could result in the compiler using excessive memory or hitting other similar limits. In extreme cases, this can even result in build failure. On the other hand, if the batch size is too low, there will be little gain in build performance.

Although strongly discouraged, the batch size may be set to a value of 0 to combine all the sources for the target into a single unity file, regardless of how many sources are involved. This runs the risk of creating an excessively large unity source file and negatively impacting the build performance, so a value of 0 is not generally recommended.

UNITY_BUILD_CODE_AFTER_INCLUDE
New in version 3.16.

Code snippet which is included verbatim by the **UNITY_BUILD** feature just after every **#include** statement in the generated unity source files. For example:

```
set(after [[
#if defined(NOMINMAX)
#undef NOMINMAX
#endif
]])
set_target_properties(myTarget PROPERTIES
  UNITY_BUILD_CODE_AFTER_INCLUDE "${after}"
)
```

See also **UNITY_BUILD_CODE_BEFORE_INCLUDE**.

UNITY_BUILD_CODE_BEFORE_INCLUDE
New in version 3.16.

Code snippet which is included verbatim by the **UNITY_BUILD** feature just before every **#include** statement in the generated unity source files. For example:

```
set(before [[
#if !defined(NOMINMAX)
#define NOMINMAX
#endif
]])
set_target_properties(myTarget PROPERTIES
  UNITY_BUILD_CODE_BEFORE_INCLUDE "${before}"
)
```

See also **UNITY_BUILD_CODE_AFTER_INCLUDE**.

UNITY_BUILD_MODE
New in version 3.18.

CMake provides different algorithms for selecting which sources are grouped together into a *bucket*. Selection is decided by this property, which has the following acceptable values:

BATCH
When in this mode CMake determines which files are grouped together. The **UNITY_BUILD_BATCH_SIZE** property controls the upper limit on how many sources can be combined per unity source file.

Example usage:

```
add_library(example_library
            source1.cxx
            source2.cxx
            source3.cxx
            source4.cxx)

set_target_properties(example_library PROPERTIES
                      UNITY_BUILD_MODE BATCH
                      UNITY_BUILD_BATCH_SIZE 2
                      )
```

GROUP
When in this mode each target explicitly specifies how to group source files. Each source file that has the same **UNITY_GROUP** value will be grouped together. Any sources that don't have this property will be compiled individually. The **UNITY_BUILD_BATCH_SIZE** property is ignored when using this mode.

Example usage:

```
add_library(example_library
            source1.cxx
            source2.cxx
            source3.cxx
            source4.cxx)
```

```
set_target_properties(example_library PROPERTIES
                      UNITY_BUILD_MODE GROUP
                      )

set_source_files_properties(source1.cxx source2.cxx source3.cxx
                            PROPERTIES UNITY_GROUP "bucket1"
                            )
set_source_files_properties(source4.cxx
                            PROPERTIES UNITY_GROUP "bucket2"
                            )
```

If no explicit *UNITY_BUILD_MODE* has been specified, CMake will default to **BATCH**.

UNITY_BUILD_UNIQUE_ID
New in version 3.20.

The name of a valid C−identifier which is set to a unique per−file value during unity builds.

When this property is populated and when **UNITY_BUILD** is true, the property value is used to define a compiler definition of the specified name. The value of the defined symbol is unspecified, but it is unique per file path.

Given:

```
set_target_properties(myTarget PROPERTIES
  UNITY_BUILD "ON"
  UNITY_BUILD_UNIQUE_ID "MY_UNITY_ID"
)
```

the **MY_UNITY_ID** symbol is defined to a unique per−file value.

One known use case for this identifier is to disambiguate the variables in an anonymous namespace in a limited scope. Anonymous namespaces present a problem for unity builds because they are used to ensure that certain variables and declarations are scoped to a translation unit which is approximated by a single source file. When source files are combined in a unity build file, those variables in different files are combined in a single translation unit and the names clash. This property can be used to avoid that with code like the following:

```
// Needed for when unity builds are disabled
#ifndef MY_UNITY_ID
#define MY_UNITY_ID
#endif

namespace { namespace MY_UNITY_ID {
  // The name 'i' clashes (or could clash) with other
  // variables in other anonymous namespaces
  int i = 42;
}}

int use_var()
{
  return MY_UNITY_ID::i;
}
```

The pseudonymous namespace is used within a truly anonymous namespace. On many platforms, this maintains the invariant that the symbols within do not get external linkage when performing a unity build.

VERSION
What version number is this target.

For shared libraries **VERSION** and **SOVERSION** can be used to specify the build version and API version respectively. When building or installing appropriate symlinks are created if the platform supports symlinks and the linker supports so−names. If only one of both is specified the missing is assumed to have the same version number. For executables **VERSION** can be used to specify the build version. When building or installing appropriate symlinks are created if the platform supports symlinks.

Windows Versions
For shared libraries and executables on Windows the **VERSION** attribute is parsed to extract a **<major>.<minor>** version number. These numbers are used as the image version of the binary.

Mach−O Versions
For shared libraries and executables on Mach−O systems (e.g. macOS, iOS), the **SOVERSION** property corresponds to the *compatibility version* and **VERSION** corresponds to the *current version* (unless Mach−O specific overrides are provided, as discussed below). See the**FRAMEW ORK** target property for an example.

For shared libraries, the **MACHO_COMPATIBILITY_VERSION** and **MACHO_CURRENT_VERSION** properties can be used to override the *compatibility version* and *current version* respectively. Note that **SOVERSION** will still be used to form the **install_name** and both **SOVERSION** and **VERSION** may also affect the file and symlink names.

Versions of Mach−O binaries may be checked with the **otool −L <binary>** command.

VISIBILITY_INLINES_HIDDEN
Whether to add a compile flag to hide symbols of inline functions

The **VISIBILITY_INLINES_HIDDEN** property determines whether a flag for hiding symbols for inline functions, such as **−fvisibility−inlines−hidden**, should be used when invoking the compiler. This property affects compilation in sources of all types of targets (subject to policy **CMP0063**).

This property is initialized by the value of the **CMAKE_VISIBILITY_INLINES_HIDDEN** variable if it is set when a target is created.

VS_CONFIGURATION_TYPE
New in version 3.6.

Visual Studio project configuration type.

Sets the **ConfigurationType** attribute for a generated Visual Studio project. The property value may use **generator expressions**. If this property is set, it overrides the default setting that is based on the target type (e.g. **StaticLibrary**, **Application**, ...).

Supported on Visual Studio Generators for VS 2010 and higher.

VS_DEBUGGER_COMMAND
New in version 3.12.

Sets the local debugger command for Visual Studio C++ targets. The property value may use **generator expressions**. This is defined in**<LocalDeb uggerCommand>** in the Visual Studio project file.

This property only works for Visual Studio 2010 and above; it is ignored on other generators.

VS_DEBUGGER_COMMAND_ARGUMENTS
New in version 3.13.

Sets the local debugger command line arguments for Visual Studio C++ targets. The property value may use **generator expressions**. This is defined in**<LocalDeb uggerCommandArguments>** in the Visual Studio project file.

This property only works for Visual Studio 2010 and above; it is ignored on other generators.

VS_DEBUGGER_ENVIRONMENT
New in version 3.13.

Sets the local debugger environment for Visual Studio C++ targets. The property value may use **generator expressions**. This is defined in**<LocalDeb uggerEnvironment>** in the Visual Studio project file.

This property only works for Visual Studio 2010 and above; it is ignored on other generators.

VS_DEBUGGER_WORKING_DIRECTORY
New in version 3.8.

Sets the local debugger working directory for Visual Studio C++ targets. The property value may use **generator expressions**. This is defined in**<LocalDeb uggerWorkingDirectory>** in the Visual Studio project file.

This property only works for Visual Studio 2010 and above; it is ignored on other generators.

VS_DESKTOP_EXTENSIONS_VERSION
New in version 3.4.

Visual Studio Windows 10 Desktop Extensions Version

Specifies the version of the Desktop Extensions that should be included in the target. For example **10.0.10240.0**. If the value is not specified, the Desktop Extensions will not be included. To use the same version of the extensions as the Windows 10 SDK that is being used, you can use the **CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION** variable.

VS_DOTNET_DOCUMENTATION_FILE
New in version 3.17.

Visual Studio managed project .NET documentation output

Sets the target XML documentation file output.

VS_DOTNET_REFERENCE_<refname>
New in version 3.8.

Visual Studio managed project .NET reference with name **<refname>** and hint path.

Adds one .NET reference to generated Visual Studio project. The reference will have the name **<refname>** and will point to the assembly given as value of the property.

See also **VS_DOTNET_REFERENCES** and **VS_DOTNET_REFERENCES_COPY_LOCAL**

VS_DOTNET_REFERENCEPROP_<refname>_TAG_<tagname>
New in version 3.10.

Defines an XML property **<tagname>** for a .NET reference **<refname>**.

Reference properties can be set for .NET references which are defined by the target properties **VS_DOTNET_REFERENCES**, **VS_DOTNET_REFERENCE_<refname>** and also for project references to other C# targets which are established by **target_link_libraries()**.

This property is only applicable to C# targets and Visual Studio generators 2010 and later.

VS_DOTNET_REFERENCES
Visual Studio managed project .NET references

Adds one or more semicolon−delimited .NET references to a generated Visual Studio project. For example, "System;System.Windows.Forms".

VS_DOTNET_REFERENCES_COPY_LOCAL
New in version 3.8.

Sets the **Copy Local** property for all .NET hint references in the target

Boolean property to enable/disable copying of .NET hint references to output directory. The default is **ON**.

VS_DOTNET_TARGET_FRAMEWORK_VERSION
Specify the .NET target framework version.

Used to specify the .NET target framework version for C++/CLI. For example, "v4.5".

This property is deprecated and should not be used anymore. Use **DOTNET_TARGET_FRAMEWORK** or **DOTNET_TARGET_FRAMEWORK_VERSION** instead.

VS_DPI_AWARE
New in version 3.16.

Set the Manifest Tool −> Input and Output −> DPI Awareness in the Visual Studio target project properties.

Valid values are **PerMonitor**, **ON**, or **OFF**.

For example:

```
add_executable(myproject myproject.cpp)
set_property(TARGET myproject PROPERTY VS_DPI_AWARE "PerMonitor")
```

VS_GLOBAL_KEYWORD
Visual Studio project keyword for VS 10 (2010) and newer.

Sets the "keyword" attribute for a generated Visual Studio project. Defaults to "Win32Proj". You may wish to override this value with "ManagedCProj", for example, in a Visual Studio managed C++ unit test project.

Use the **VS_KEYWORD** target property to set the keyword for Visual Studio 9 (2008) and older.

VS_GLOBAL_PROJECT_TYPES
Visual Studio project type(s).

Can be set to one or more UUIDs recognized by Visual Studio to indicate the type of project. This value is copied verbatim into the generated project file. Example for a managed C++ unit testing project:

```
{3AC096D0-A1C2-E12C-1390-A8335801FDAB};{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}
```

UUIDs are semicolon-delimited.

VS_GLOBAL_ROOTNAMESPACE
Visual Studio project root namespace.

Sets the "RootNamespace" attribute for a generated Visual Studio project. The attribute will be generated only if this is set.

VS_GLOBAL_<variable>
Visual Studio project-specific global variable.

Tell the Visual Studio generator to set the global variable '<variable>' to a given value in the generated Visual Studio project. Ignored on other generators. Qt integration works better if VS_GLOBAL_QtVersion is set to the version FindQt4.cmake found. For example, "4.7.3"

VS_IOT_EXTENSIONS_VERSION
New in version 3.4.

Visual Studio Windows 10 IoT Extensions Version

Specifies the version of the IoT Extensions that should be included in the target. For example **10.0.10240.0**. If the value is not specified, the IoT Extensions will not be included. To use the same version of the extensions as the Windows 10 SDK that is being used, you can use the **CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION** variable.

VS_IOT_STARTUP_TASK
New in version 3.4.

Visual Studio Windows 10 IoT Continuous Background Task

Specifies that the target should be compiled as a Continuous Background Task library.

VS_JUST_MY_CODE_DEBUGGING
New in version 3.15.

Enable Just My Code with Visual Studio debugger.

Supported on Visual Studio Generators for VS 2010 and higher, Makefile Generators and the **Ninja** generators.

This property is initialized by the **CMAKE_VS_JUST_MY_CODE_DEBUGGING** variable if it is set when a target is created.

VS_KEYWORD
Visual Studio project keyword for VS 9 (2008) and older.

Can be set to change the visual studio keyword, for example Qt integration works better if this is set to

Qt4VSv1.0.

Use the **VS_GLOBAL_KEYWORD** target property to set the keyword for Visual Studio 10 (2010) and newer.

VS_MOBILE_EXTENSIONS_VERSION
New in version 3.4.

Visual Studio Windows 10 Mobile Extensions Version

Specifies the version of the Mobile Extensions that should be included in the target. For example **10.0.10240.0**. If the value is not specified, the Mobile Extensions will not be included. To use the same version of the extensions as the Windows 10 SDK that is being used, you can use the **CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION** variable.

VS_NO_SOLUTION_DEPLOY
New in version 3.15.

Specify that the target should not be marked for deployment to a Windows CE or Windows Phone device in the generated Visual Studio solution.

Be default, all EXE and shared library (DLL) targets are marked to deploy to the target device in the generated Visual Studio solution.

Generator expressions are supported.

There are reasons one might want to exclude a target / generated project from deployment:

- The library or executable may not be necessary in the primary deploy/debug scenario, and excluding from deployment saves time in the develop/download/debug cycle.
- There may be insufficient space on the target device to accommodate all of the build products.
- Visual Studio 2013 requires a target device IP address be entered for each target marked for deployment. For large numbers of targets, this can be tedious. NOTE: Visual Studio *will* deploy all project dependencies of a project tagged for deployment to the IP address configured for that project even if those dependencies are not tagged for deployment.

Example 1
This shows setting the variable for the target foo.

```
add_library(foo SHARED foo.cpp)
set_property(TARGET foo PROPERTY VS_NO_SOLUTION_DEPLOY ON)
```

Example 2
This shows setting the variable for the Release configuration only.

```
add_library(foo SHARED foo.cpp)
set_property(TARGET foo PROPERTY VS_NO_SOLUTION_DEPLOY "$<CONFIG:Release>")
```

VS_PACKAGE_REFERENCES
New in version 3.15.

Visual Studio package references for nuget.

Adds one or more semicolon−delimited package references to a generated Visual Studio project. The

version of the package will be underscore delimited. For example, **boost_1.7.0;nunit_3.12.***.

```
set_property(TARGET ${TARGET_NAME} PROPERTY
  VS_PACKAGE_REFERENCES "boost_1.7.0")
```

VS_PLATFORM_TOOLSET
New in version 3.18.

Overrides the platform toolset used to build a target.

Only supported when the compiler used by the given toolset is the same as the compiler used to build the whole source tree.

This is especially useful to create driver projects with the toolsets "WindowsUserModeDriver10.0" or "WindowsKernelModeDriver10.0".

VS_PROJECT_IMPORT
New in version 3.15.

Visual Studio managed project imports

Adds to a generated Visual Studio project one or more semicolon−delimited paths to .props files needed when building projects from some NuGet packages. For example, **my_packages_path/MyPackage.1.0.0/build/MyPackage.props**.

VS_SCC_AUXPATH
Visual Studio Source Code Control Aux Path.

Can be set to change the visual studio source code control auxpath property.

VS_SCC_LOCALPATH
Visual Studio Source Code Control Local Path.

Can be set to change the visual studio source code control local path property.

VS_SCC_PROJECTNAME
Visual Studio Source Code Control Project.

Can be set to change the visual studio source code control project name property.

VS_SCC_PROVIDER
Visual Studio Source Code Control Provider.

Can be set to change the visual studio source code control provider property.

VS_SDK_REFERENCES
New in version 3.7.

Visual Studio project SDK references. Specify a semicolon−separated list of SDK references to be added to a generated Visual Studio project, e.g. **Microsoft.AdMediatorWindows81, Version=1.0**.

VS_SOLUTION_DEPLOY
New in version 3.18.

Specify that the target should be marked for deployment when not targeting Windows CE, Windows Phone

or a Windows Store application.

If the target platform doesn't support deployment, this property won't have any effect.

Generator expressions are supported.

Examples

Always deploy target **foo**:

```
add_executable(foo SHARED foo.cpp)
set_property(TARGET foo PROPERTY VS_SOLUTION_DEPLOY ON)
```

Deploy target **foo** for all configurations except **Release**:

```
add_executable(foo SHARED foo.cpp)
set_property(TARGET foo PROPERTY VS_SOLUTION_DEPLOY "$<NOT:$<CONFIG:Release>>"
```

VS_SOURCE_SETTINGS_<tool>

New in version 3.18.

Set any item metadata on all non−built files that use <tool>.

Takes a list of **Key=Value** pairs. Tells the Visual Studio generator to set **Key** to **Value** as item metadata on all non−built files that use **<tool>**.

For example:

```
set_property(TARGET main PROPERTY VS_SOURCE_SETTINGS_FXCompile "Key=Value" "Key
```

will set **Key** to **Value** and **Key2** to **Value2** for all non−built files that use **FXCompile**.

Generator expressions are supported.

VS_USER_PROPS

New in version 3.8.

Sets the user props file to be included in the visual studio C++ project file. The standard path is **$(UserRootDir)\\Microsoft.Cpp.$(Platform).user.props**, which is in most cases the same as **%LOCALAPPDATA%\\Microsoft\\MSBuild\\v4.0\\Microsoft.Cpp.Win32.user.props** or **%LOCALAPPDATA%\\Microsoft\\MSBuild\\v4.0\\Microsoft.Cpp.x64.user.props**.

The ***.user.props** files can be used for Visual Studio wide configuration which is independent from cmake.

VS_WINDOWS_TARGET_PLATFORM_MIN_VERSION

New in version 3.4.

Visual Studio Windows Target Platform Minimum Version

For Windows 10. Specifies the minimum version of the OS that is being targeted. For example **10.0.10240.0**. If the value is not specified, the value of **CMAKE_VS_WINDOWS_TARGET_PLATFORM_VERSION** will be used on WindowsStore projects otherwise the target platform minimum version will not be specified for the project.

VS_WINRT_COMPONENT
New in version 3.1.

Mark a target as a Windows Runtime component for the Visual Studio generator. Compile the target with **C++/CX** language extensions for Windows Runtime. For **SHARED** and **MODULE** libraries, this also defines the **_WINRT_DLL** preprocessor macro.

NOTE:
Currently this is implemented only by Visual Studio generators. Support may be added to other generators in the future.

VS_WINRT_EXTENSIONS
Deprecated. Use**VS_WINR T_COMPONENT** instead. This property was an experimental partial implementation of that one.

VS_WINRT_REFERENCES
Visual Studio project Windows Runtime Metadata references

Adds one or more semicolon−delimited WinRT references to a generated Visual Studio project. For example, "Windows;Windows.UI.Core".

WIN32_EXECUTABLE
Build an executable with a WinMain entry point on windows.

When this property is set to true the executable when linked on Windows will be created with a WinMain() entry point instead of just main(). This makes it a GUI executable instead of a console application. See the **CMAKE_MFC_FLAG** variable documentation to configure use of the Microsoft Foundation Classes (MFC) for WinMain executables. This property is initialized by the value of the **CMAKE_WIN32_EXECUTABLE** variable if it is set when a target is created.

This property supports **generator expressions**, except if the target is managed (contains C# code.)

WINDOWS_EXPORT_ALL_SYMBOLS
New in version 3.4.

This property is implemented only for MS−compatible tools on Windows.

Enable this boolean property to automatically create a module definition (**.def**) file with all global symbols found in the input **.obj** files for a **SHARED** library (or executable with **ENABLE_EXPORTS**) on Windows. The module definition file will be passed to the linker causing all symbols to be exported from the **.dll**. For global *data* symbols, **__declspec(dllimport)** must still be used when compiling against the code in the **.dll**. All other function symbols will be automatically exported and imported by callers. This simplifies porting projects to Windows by reducing the need for explicit **dllexport** markup, even in **C++** classes.

When this property is enabled, zero or more **.def** files may also be specified as source files of the target. The exports named by these files will be merged with those detected from the object files to generate a single module definition file to be passed to the linker. This can be used to export symbols from a **.dll** that are not in any of its object files but are added by the linker from dependencies (e.g. **msvcrt.lib**).

This property is initialized by the value of the **CMAKE_WINDOWS_EXPORT_ALL_SYMBOLS** variable if it is set when a target is created.

XCODE_ATTRIBUTE_<an−attribute>
Set Xcode target attributes directly.

Tell the **Xcode** generator to set **<an−attribute>** to a given value in the generated Xcode project. Ignored on other generators.

This offers low−level control over the generated Xcode project file. It is meant as a last resort for specifying settings that CMake does not otherwise have a way to control. Although this can override a setting CMake normally produces on its own, doing so bypasses CMake's model of the project and can break things.

See the **CMAKE_XCODE_ATTRIBUTE_<an−attribute>** variable to set attributes on all targets in a directory tree.

Contents of **XCODE_ATTRIBUTE_<an−attribute>** may use "generator expressions" with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions. See the **cmake−buildsystem(7)** manual for more on defining buildsystem properties.

XCODE_EMBED_FRAMEWORKS_CODE_SIGN_ON_COPY
New in version 3.20.

Tell the **Xcode** generator to perform code signing for all the frameworks and libraries that are embedded using the **XCODE_EMBED_FRAMEWORKS** property.

New in version 3.21.

This property was generalized to other types of embedded items. See**XCODE_EMBED_<type>_CODE_SIGN_ON_COPY** for the more general form.

XCODE_EMBED_FRAMEWORKS_REMOVE_HEADERS_ON_COPY
New in version 3.20.

Tell the **Xcode** generator to remove headers from all the frameworks that are embedded using the **XCODE_EMBED_FRAMEWORKS** property.

New in version 3.21.

This property was generalized to other types of embedded items. See **XCODE_EMBED_<type>_REMOVE_HEADERS_ON_COPY** for the more general form.

XCODE_EMBED_<type>
New in version 3.20.

Tell the **Xcode** generator to embed the specified list of items into the target bundle. **<type>** specifies the embed build phase to use. See the Xcode documentation for the base location of each **<type>**.

The supported values for **<type>** are:

FRAMEWORKS
The specified items will be added to the **Embed Frameworks** build phase. The items can be CMake target names or paths to frameworks or libraries.

APP_EXTENSIONS
New in version 3.21.

The specified items will be added to the **Embed App Extensions** build phase. They must be CMake target names.

See also **XCODE_EMBED_<type>_PATH**, **XCODE_EMBED_<type>_REMOVE_HEADERS_ON_COPY** and **XCODE_EMBED_<type>_CODE_SIGN_ON_COPY**.

XCODE_EMBED_<type>_CODE_SIGN_ON_COPY
New in version 3.20.

Boolean property used only by the **Xcode** generator. It specifies whether to perform code signing for the items that are embedded using the **XCODE_EMBED_<type>** property.

The supported values for **<type>** are:

FRAMEWORKS

APP_EXTENSIONS
New in version 3.21.

If a **XCODE_EMBED_<type>_CODE_SIGN_ON_COPY** property is not defined on the target, no code signing on copy will be performed for that **<type>**.

XCODE_EMBED_<type>_PATH
New in version 3.20.

This property is used only by the **Xcode** generator. When defined, it specifies the relative path to use when embedding the items specified by **XCODE_EMBED_<type>**. The path is relative to the base location of the **Embed XXX** build phase associated with **<type>**. See the Xcode documentation for the base location of each **<type>**.

The supported values for **<type>** are:

FRAMEWORKS

APP_EXTENSIONS
New in version 3.21.

XCODE_EMBED_<type>_REMOVE_HEADERS_ON_COPY
New in version 3.20.

Boolean property used only by the **Xcode** generator. It specifies whether to remove headers from all the frameworks that are embedded using the **XCODE_EMBED_<type>** property.

The supported values for **<type>** are:

FRAMEWORKS
If the **XCODE_EMBED_FRAMEWORKS_REMOVE_HEADERS_ON_COPY** property is not defined, headers will not be removed on copy by default.

APP_EXTENSIONS
New in version 3.21.

If the **XCODE_EMBED_APP_EXTENSIONS_REMOVE_HEADERS_ON_COPY** property

is not defined, headers WILL be removed on copy by default.

XCODE_EXPLICIT_FILE_TYPE
New in version 3.8.

Set the Xcode **explicitFileType** attribute on its reference to a target. CMake computes a default based on target type but can be told explicitly with this property.

See also **XCODE_PRODUCT_TYPE**.

XCODE_GENERATE_SCHEME
New in version 3.15.

If enabled, the **Xcode** generator will generate schema files. These are useful to invoke analyze, archive, build−for−testing and test actions from the command line.

This property is initialized by the value of the variable **CMAKE_XCODE_GENERATE_SCHEME** if it is set when a target is created.

The following target properties overwrite the default of the corresponding settings on the "Diagnostic" tab for each schema file. Each of those is initialized by the respective **CMAKE_** variable at target creation time.

- **XCODE_SCHEME_ADDRESS_SANITIZER**
- **XCODE_SCHEME_ADDRESS_SANITIZER_USE_AFTER_RETURN**
- **XCODE_SCHEME_DISABLE_MAIN_THREAD_CHECKER**
- **XCODE_SCHEME_DYNAMIC_LIBRARY_LOADS**
- **XCODE_SCHEME_DYNAMIC_LINKER_API_USAGE**
- **XCODE_SCHEME_GUARD_MALLOC**
- **XCODE_SCHEME_MAIN_THREAD_CHECKER_STOP**
- **XCODE_SCHEME_MALLOC_GUARD_EDGES**
- **XCODE_SCHEME_MALLOC_SCRIBBLE**
- **XCODE_SCHEME_MALLOC_STACK**
- **XCODE_SCHEME_THREAD_SANITIZER**
- **XCODE_SCHEME_THREAD_SANITIZER_STOP**
- **XCODE_SCHEME_UNDEFINED_BEHAVIOUR_SANITIZER**
- **XCODE_SCHEME_UNDEFINED_BEHAVIOUR_SANITIZER_STOP**
- **XCODE_SCHEME_ZOMBIE_OBJECTS**

The following target properties will be applied on the "Info", "Arguments", and "Options" tab:

- **XCODE_SCHEME_ARGUMENTS**
- **XCODE_SCHEME_DEBUG_AS_ROOT**
- **XCODE_SCHEME_DEBUG_DOCUMENT_VERSIONING**
- **XCODE_SCHEME_ENVIRONMENT**
- **XCODE_SCHEME_EXECUTABLE**

- **XCODE_SCHEME_WORKING_DIRECTORY**

XCODE_LINK_BUILD_PHASE_MODE
New in version 3.19.

When using the **Xcode** generator, libraries to be linked will be specified in the Xcode project file using either the "Link Binary With Libraries" build phase or directly as linker flags. The former allows Xcode to manage build paths, which may be necessary when creating Xcode archives because it may use different build paths to a regular build.

This property controls usage of "Link Binary With Libraries" build phase for a target that is an app bundle, executable, shared library, shared framework or a module library.

Possible values are:

- **NONE** The libraries will be linked by specifying the linker flags directly.
- **BUILT_ONLY** The "Link Binary With Libraries" build phase will be used to link to another target under the following conditions:
  - The target to be linked to is a regular non−imported, non−interface library target.
  - The output directory of the target being built has not been changed from its default (see **RUNTIME_OUTPUT_DIRECTORY** and **LIBRARY_OUTPUT_DIRECTORY**).
- **KNOWN_LOCATION** The "Link Binary With Libraries" build phase will be used to link to another target under the same conditions as with **BUILT_ONLY** and also:
  - Imported library targets except those of type **UNKNOWN**.
  - Any non−target library specified directly with a path.

For all other cases, the libraries will be linked by specifying the linker flags directly.

WARNING:
Libraries linked using "Link Binary With Libraries" are linked after the ones linked through regular linker flags. This order should be taken into account when different static libraries contain symbols with the same name, as the former ones will take precedence over the latter.

WARNING:
If two or more directories contain libraries with identical file names and some libraries are linked from those directories, the library search path lookup will end up linking libraries from the first directory. This is a known limitation of Xcode.

This property is initialized by the value of the **CMAKE_XCODE_LINK_BUILD_PHASE_MODE** variable if it is set when a target is created.

XCODE_PRODUCT_TYPE
New in version 3.8.

Set the Xcode **productType** attribute on its reference to a target. CMake computes a default based on target type but can be told explicitly with this property.

See also **XCODE_EXPLICIT_FILE_TYPE**.

XCODE_SCHEME_ADDRESS_SANITIZER
New in version 3.13.

Whether to enable **Address Sanitizer** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_ADDRESS_SANITIZER** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_ADDRESS_SANITIZER_USE_AFTER_RETURN
New in version 3.13.

Whether to enable **Detect use of stack after return** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_ADDRESS_SANITIZER_USE_AFTER_RETURN** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_ARGUMENTS
New in version 3.13.

Specify command line arguments that should be added to the Arguments section of the generated Xcode scheme.

If set to a list of arguments those will be added to the scheme.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_DEBUG_AS_ROOT
New in version 3.15.

Whether to debug the target as 'root'.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_DEBUG_DOCUMENT_VERSIONING
New in version 3.16.

Whether to enable **Allow debugging when using document Versions Browser** in the Options section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_DEBUG_DOCUMENT_VERSIONING** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_DISABLE_MAIN_THREAD_CHECKER
New in version 3.13.

Whether to disable the **Main Thread Checker** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_DISABLE_MAIN_THREAD_CHECKER** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_DYNAMIC_LIBRARY_LOADS
New in version 3.13.

Whether to enable **Dynamic Library Loads** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_DYNAMIC_LIBRARY_LOADS** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_DYNAMIC_LINKER_API_USAGE
New in version 3.13.

Whether to enable **Dynamic Linker API usage** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_DYNAMIC_LINKER_API_USAGE** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_ENVIRONMENT
New in version 3.13.

Specify environment variables that should be added to the Arguments section of the generated Xcode scheme.

If set to a list of environment variables and values of the form **MYVAR=value** those environment variables will be added to the scheme.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_EXECUTABLE
New in version 3.13.

Specify path to executable in the Info section of the generated Xcode scheme. If not set the schema generator will select the current target if it is actually executable.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode

schema related properties.

XCODE_SCHEME_GUARD_MALLOC
New in version 3.13.

Whether to enable **Guard Malloc** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_GUARD_MALLOC** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_MAIN_THREAD_CHECKER_STOP
New in version 3.13.

Whether to enable the **Main Thread Checker** option **Pause on issues** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_MAIN_THREAD_CHECKER_STOP** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_MALLOC_GUARD_EDGES
New in version 3.13.

Whether to enable **Malloc Guard Edges** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_MALLOC_GUARD_EDGES** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_MALLOC_SCRIBBLE
New in version 3.13.

Whether to enable **Malloc Scribble** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_MALLOC_SCRIBBLE** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_MALLOC_STACK
New in version 3.13.

Whether to enable **Malloc Stack** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_MALLOC_STACK** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_THREAD_SANITIZER
New in version 3.13.

Whether to enable **Thread Sanitizer** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_THREAD_SANITIZER** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_THREAD_SANITIZER_STOP
New in version 3.13.

Whether to enable **Thread Sanitizer − Pause on issues** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_THREAD_SANITIZER_STOP** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_UNDEFINED_BEHAVIOUR_SANITIZER
New in version 3.13.

Whether to enable **Undefined Behavior Sanitizer** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_UNDEFINED_BEHAVIOUR_SANITIZER** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_UNDEFINED_BEHAVIOUR_SANITIZER_STOP
New in version 3.13.

Whether to enable **Undefined Behavior Sanitizer** option **Pause on issues** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_UNDEFINED_BEHAVIOUR_SANITIZER_STOP** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_WORKING_DIRECTORY
New in version 3.17.

Specify the **Working Directory** of the *Run* and *Profile* actions in the generated Xcode scheme. In case the value contains generator expressions those are evaluated.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_WORKING_DIRECTORY** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCODE_SCHEME_ZOMBIE_OBJECTS
New in version 3.13.

Whether to enable **Zombie Objects** in the Diagnostics section of the generated Xcode scheme.

This property is initialized by the value of the variable **CMAKE_XCODE_SCHEME_ZOMBIE_OBJECTS** if it is set when a target is created.

Please refer to the **XCODE_GENERATE_SCHEME** target property documentation to see all Xcode schema related properties.

XCTEST
New in version 3.3.

This target is a XCTest CFBundle on the Mac.

This property will usually get set via the **xctest_add_bundle()** macro in **FindXCTest** module.

If a module library target has this property set to true it will be built as a CFBundle when built on the Mac. It will have the directory structure required for a CFBundle.

This property depends on **BUNDLE** to be effective.

PROPERTIES ON TESTS

ATTACHED_FILES
Attach a list of files to a dashboard submission.

Set this property to a list of files that will be encoded and submitted to the dashboard as an addition to the test result.

ATTACHED_FILES_ON_FAIL
Attach a list of files to a dashboard submission if the test fails.

Same as **ATTACHED_FILES**, but these files will only be included if the test does not pass.

COST
This property describes the cost of a test. When parallel testing is enabled, tests in the test set will be run in descending order of cost. Projects can explicitly define the cost of a test by setting this property to a floating point value.

When the cost of a test is not defined by the project, **ctest** will initially use a default cost of **0**. It computes a weighted average of the cost each time a test is run and uses that as an improved estimate of the cost for

the next run. The more a test is re−run in the same build directory, the more representative the cost should become.

DEPENDS
Specifies that this test should only be run after the specified list of tests.

Set this to a list of tests that must finish before this test is run. The results of those tests are not considered, the dependency relationship is purely for order of execution (i.e. it is really just a *run after* relationship). Consider using test fixtures with setup tests if a dependency with successful completion is required (see **FIXTURES_REQUIRED**).

Examples

```
add_test(NAME baseTest1 ...)
add_test(NAME baseTest2 ...)
add_test(NAME dependsTest12 ...)

set_tests_properties(dependsTest12 PROPERTIES DEPENDS "baseTest1;baseTest2")
# dependsTest12 runs after baseTest1 and baseTest2, even if they fail
```

DISABLED
New in version 3.9.

If set to **True**, the test will be skipped and its status will be 'Not Run'. A **DISABLED** test will not be counted in the total number of tests and its completion status will be reported to CDash as **Disabled**.

A **DISABLED** test does not participate in test fixture dependency resolution. If a **DISABLED** test has fixture requirements defined in its **FIXTURES_REQUIRED** property, it will not cause setup or cleanup tests for those fixtures to be added to the test set.

If a test with the **FIXTURES_SETUP** property set is **DISABLED**, the fixture behavior will be as though that setup test was passing and any test case requiring that fixture will still run.

ENVIRONMENT
Specify environment variables that should be defined for running a test.

If set to a list of environment variables and values of the form **MYVAR=value** those environment variables will be defined while running the test. The environment changes from this property do not affect other tests.

ENVIRONMENT_MODIFICATION
New in version 3.22.

Specify environment variables that should be modified for running a test. Note that the operations performed by this property are performed after the **ENVIRONMENT** property is already applied.

If set to a list of environment variables and values of the form **MYVAR=OP:VALUE**, where **MYVAR** is the case−sensitive name of an environment variable to be modified. Entries are considered in the order specified in the property's value. The **OP** may be one of:

- **reset**: Reset to the unmodified value, ignoring all modifications to **MYVAR** prior to this entry. Note that this will reset the variable to the value set by **ENVIRONMENT**, if it was set, and otherwise to its state from the rest of the CTest execution.
- **set**: Replaces the current value of **MYVAR** with **VALUE**.
- **unset**: Unsets the current value of **MYVAR**.

- **string_append**: Appends **VALUE** to the current value of **MYVAR**.
- **string_prepend**: Prepends **VALUE** to the current value of **MYVAR**.
- **path_list_append**: Appends **VALUE** to the current value of **MYVAR** using the host platform's path list separator (**;** on Windows and **:** elsewhere).
- **path_list_prepend**: Prepends **VALUE** to the current value of **MYVAR** using the host platform's path list separator (**;** on Windows and **:** elsewhere).
- **cmake_list_append**: Appends **VALUE** to the current value of **MYVAR** using **;** as the separator.
- **cmake_list_prepend**: Prepends **VALUE** to the current value of **MYVAR** using **;** as the separator.

Unrecognized **OP** values will result in the test failing before it is executed. This is so that future operations may be added without changing valid behavior of existing tests.

The environment changes from this property do not affect other tests.

FAIL_REGULAR_EXPRESSION
If the output matches this regular expression the test will fail, regardless of the process exit code.

If set, if the output matches one of specified regular expressions, the test will fail. Example:

```
set_tests_properties(mytest PROPERTIES
  FAIL_REGULAR_EXPRESSION "[^a-z]Error;ERROR;Failed"
)
```

FAIL_REGULAR_EXPRESSION expects a list of regular expressions.

See also the **PASS_REGULAR_EXPRESSION** and **SKIP_REGULAR_EXPRESSION** test properties.

FIXTURES_CLEANUP
New in version 3.7.

Specifies a list of fixtures for which the test is to be treated as a cleanup test. These fixture names are distinct from test case names and are not required to have any similarity to the names of tests associated with them.

Fixture cleanup tests are ordinary tests with all of the usual test functionality. Setting the **FIXTURES_CLEANUP** property for a test has two primary effects:

- CTest will ensure the test executes after all other tests which list any of the fixtures in its **FIXTURES_REQUIRED** property.
- If CTest is asked to run only a subset of tests (e.g. using regular expressions or the **--rerun-failed** option) and the cleanup test is not in the set of tests to run, it will automatically be added if any tests in the set require any fixture listed in **FIXTURES_CLEANUP**.

A cleanup test can have multiple fixtures listed in its **FIXTURES_CLEANUP** property. It will execute only once for the whole CTest run, not once for each fixture. A fixture can also have more than one cleanup test defined. If there are multiple cleanup tests for a fixture, projects can control their order with the usual **DEPENDS** test property if necessary.

A cleanup test is allowed to require other fixtures, but not any fixture listed in its **FIXTURES_CLEANUP** property. For example:

```
# Ok: Dependent fixture is different to cleanup
set_tests_properties(cleanupFoo PROPERTIES
```

```
  FIXTURES_CLEANUP  Foo
  FIXTURES_REQUIRED Bar
)

# Error: cannot require same fixture as cleanup
set_tests_properties(cleanupFoo PROPERTIES
  FIXTURES_CLEANUP  Foo
  FIXTURES_REQUIRED Foo
)
```

Cleanup tests will execute even if setup or regular tests for that fixture fail or are skipped.

See **FIXTURES_REQUIRED** for a more complete discussion of how to use test fixtures.

FIXTURES_REQUIRED
New in version 3.7.

Specifies a list of fixtures the test requires. Fixture names are case sensitive and they are not required to have any similarity to test names.

Fixtures are a way to attach setup and cleanup tasks to a set of tests. If a test requires a given fixture, then all tests marked as setup tasks for that fixture will be executed first (once for the whole set of tests, not once per test requiring the fixture). After all tests requiring a particular fixture have completed, CTest will ensure all tests marked as cleanup tasks for that fixture are then executed. Tests are marked as setup tasks with the **FIXTURES_SETUP** property and as cleanup tasks with the **FIXTURES_CLEANUP** property. If any of a fixture's setup tests fail, all tests listing that fixture in their **FIXTURES_REQUIRED** property will not be executed. The cleanup tests for the fixture will always be executed, even if some setup tests fail.

When CTest is asked to execute only a subset of tests (e.g. by the use of regular expressions or when run with the **−−rerun−failed** command line option), it will automatically add any setup or cleanup tests for fixtures required by any of the tests that are in the execution set. This behavior can be overridden with the **−FS**, **−FC** and **−FA** command line options to **ctest(1)** if desired.

Since setup and cleanup tasks are also tests, they can have an ordering specified by the **DEPENDS** test property just like any other tests. This can be exploited to implement setup or cleanup using multiple tests for a single fixture to modularise setup or cleanup logic.

The concept of a fixture is different to that of a resource specified by **RESOURCE_LOCK**, but they may be used together. A fixture defines a set of tests which share setup and cleanup requirements, whereas a resource lock has the effect of ensuring a particular set of tests do not run in parallel. Some situations may need both, such as setting up a database, serializing test access to that database and deleting the database again at the end. For such cases, tests would populate both **FIXTURES_REQUIRED** and **RESOURCE_LOCK** to combine the two behaviors. Names used for **RESOURCE_LOCK** have no relationship with names of fixtures, so note that a resource lock does not imply a fixture and vice versa.

Consider the following example which represents a database test scenario similar to that mentioned above:

```
add_test(NAME testsDone   COMMAND emailResults)
add_test(NAME fooOnly     COMMAND testFoo)
add_test(NAME dbOnly      COMMAND testDb)
add_test(NAME dbWithFoo   COMMAND testDbWithFoo)
add_test(NAME createDB    COMMAND initDB)
add_test(NAME setupUsers  COMMAND userCreation)
add_test(NAME cleanupDB   COMMAND deleteDB)
```

```
add_test(NAME cleanupFoo  COMMAND removeFoos)

set_tests_properties(setupUsers PROPERTIES DEPENDS createDB)

set_tests_properties(createDB   PROPERTIES FIXTURES_SETUP    DB)
set_tests_properties(setupUsers PROPERTIES FIXTURES_SETUP    DB)
set_tests_properties(cleanupDB  PROPERTIES FIXTURES_CLEANUP  DB)
set_tests_properties(cleanupFoo PROPERTIES FIXTURES_CLEANUP  Foo)
set_tests_properties(testsDone  PROPERTIES FIXTURES_CLEANUP  "DB;Foo")

set_tests_properties(fooOnly    PROPERTIES FIXTURES_REQUIRED Foo)
set_tests_properties(dbOnly     PROPERTIES FIXTURES_REQUIRED DB)
set_tests_properties(dbWithFoo  PROPERTIES FIXTURES_REQUIRED "DB;Foo")

set_tests_properties(dbOnly dbWithFoo createDB setupUsers cleanupDB
                     PROPERTIES RESOURCE_LOCK DbAccess)
```

Key points from this example:

- Two fixtures are defined: **DB** and **Foo**. Tests can require a single fixture as **fooOnly** and **dbOnly** do, or they can depend on multiple fixtures like **dbWithFoo** does.
- A **DEPENDS** relationship is set up to ensure **setupUsers** happens after **createDB**, both of which are setup tests for the **DB** fixture and will therefore be executed before the **dbOnly** and **dbWithFoo** tests automatically.
- No explicit **DEPENDS** relationships were needed to make the setup tests run before or the cleanup tests run after the regular tests.
- The **Foo** fixture has no setup tests defined, only a single cleanup test.
- **testsDone** is a cleanup test for both the **DB** and **Foo** fixtures. Therefore, it will only execute once regular tests for both fixtures have finished (i.e. after **fooOnly**, **dbOnly** and **dbWithFoo**). No **DEPENDS** relationship was specified for **testsDone**, so it is free to run before, after or concurrently with other cleanup tests for either fixture.
- The setup and cleanup tests never list the fixtures they are for in their own **FIXTURES_REQUIRED** property, as that would result in a dependency on themselves and be considered an error.

FIXTURES_SETUP
New in version 3.7.

Specifies a list of fixtures for which the test is to be treated as a setup test. These fixture names are distinct from test case names and are not required to have any similarity to the names of tests associated with them.

Fixture setup tests are ordinary tests with all of the usual test functionality. Setting the **FIXTURES_SETUP** property for a test has two primary effects:

- CTest will ensure the test executes before any other test which lists the fixture name(s) in its **FIXTURES_REQUIRED** property.
- If CTest is asked to run only a subset of tests (e.g. using regular expressions or the **−−rerun−failed** option) and the setup test is not in the set of tests to run, it will automatically be added if any tests in the set require any fixture listed in **FIXTURES_SETUP**.

A setup test can have multiple fixtures listed in its **FIXTURES_SETUP** property. It will execute only once for the whole CTest run, not once for each fixture. A fixture can also have more than one setup test defined. If there are multiple setup tests for a fixture, projects can control their order with the usual **DEPENDS** test property if necessary.

A setup test is allowed to require other fixtures, but not any fixture listed in its **FIXTURES_SETUP** property. For example:

```
# Ok: dependent fixture is different to setup
set_tests_properties(setupFoo PROPERTIES
  FIXTURES_SETUP    Foo
  FIXTURES_REQUIRED Bar
)

# Error: cannot require same fixture as setup
set_tests_properties(setupFoo PROPERTIES
  FIXTURES_SETUP    Foo
  FIXTURES_REQUIRED Foo
)
```

If any of a fixture's setup tests fail, none of the tests listing that fixture in its **FIXTURES_REQUIRED** property will be run. Cleanup tests will, however, still be executed.

See **FIXTURES_REQUIRED** for a more complete discussion of how to use test fixtures.

LABELS
Specify a list of text labels associated with a test. The labels are reported in both the **ctest** output summary and in dashboard submissions. They can also be used to filter the set of tests to be executed (see the **ctest −L** and **ctest −LE** CTest Options).

See Additional Labels for adding labels to a test dynamically during test execution.

MEASUREMENT
Specify a **CDASH** measurement and value to be reported for a test.

If set to a name then that name will be reported to **CDASH** as a named measurement with a value of **1**. You may also specify a value by setting **MEASUREMENT** to **measurement=value**.

PASS_REGULAR_EXPRESSION
The output must match this regular expression for the test to pass. The process exit code is ignored.

If set, the test output will be checked against the specified regular expressions and at least one of the regular expressions has to match, otherwise the test will fail. Example:

```
set_tests_properties(mytest PROPERTIES
  PASS_REGULAR_EXPRESSION "TestPassed;All ok"
)
```

PASS_REGULAR_EXPRESSION expects a list of regular expressions.

See also the **FAIL_REGULAR_EXPRESSION** and **SKIP_REGULAR_EXPRESSION** test properties.

PROCESSOR_AFFINITY
New in version 3.12.

Set to a true value to ask CTest to launch the test process with CPU affinity for a fixed set of processors. If enabled and supported for the current platform, CTest will choose a set of processors to place in the CPU affinity mask when launching the test process. The number of processors in the set is determined by the **PROCESSORS** test property or the number of processors available to CTest, whichever is smaller. The set of processors chosen will be disjoint from the processors assigned to other concurrently running tests that also have the **PROCESSOR_AFFINITY** property enabled.

PROCESSORS
Set to specify how many process slots this test requires. If not set, the default is **1** processor.

Denotes the number of processors that this test will require. This is typically used for MPI tests, and should be used in conjunction with the **ctest_test() PARALLEL_LEVEL** option.

This will also be used to display a weighted test timing result in label and subproject summaries in the command line output of **ctest(1)**. The wall clock time for the test run will be multiplied by this property to give a better idea of how much cpu resource CTest allocated for the test.

See also the **PROCESSOR_AFFINITY** test property.

REQUIRED_FILES
List of files required to run the test. The filenames are relative to the test **WORKING_DIRECTORY** unless an absolute path is specified.

If set to a list of files, the test will not be run unless all of the files exist.

Examples
Suppose that **test.txt** is created by test **baseTest** and **none.txt** does not exist:

```
add_test(NAME baseTest ...)   # Assumed to create test.txt
add_test(NAME fileTest ...)

# The following ensures that if baseTest is successful, test.txt will
# have been created before fileTest is run
set_tests_properties(fileTest PROPERTIES
  DEPENDS baseTest
  REQUIRED_FILES test.txt
)

add_test(NAME notRunTest ...)

# The following makes notRunTest depend on two files. Nothing creates
# the none.txt file, so notRunTest will fail with status "Not Run".
set_tests_properties(notRunTest PROPERTIES
  REQUIRED_FILES "test.txt;none.txt"
)
```

The above example demonstrates how **REQUIRED_FILES** works, but it is not the most robust way to implement test ordering with failure detection. For that, test fixtures are a better alternative (see **FIXTURES_REQUIRED**).

RESOURCE_GROUPS
New in version 3.16.

Specify resources required by a test, grouped in a way that is meaningful to the test. See resource allocation for more information on how this property integrates into the CTest resource allocation feature.

The **RESOURCE_GROUPS** property is a semicolon−separated list of group descriptions. Each entry consists of an optional number of groups using the description followed by a series of resource requirements for those groups. These requirements (and the number of groups) are separated by commas. The resource requirements consist of the name of a resource type, followed by a colon, followed by an unsigned integer specifying the number of slots required on one resource of the given type.

The **RESOURCE_GROUPS** property tells CTest what resources a test expects to use grouped in a way meaningful to the test. The test itself must read the environment variables to determine which resources have been allocated to each group. For example, each group may correspond to a process the test will spawn when executed.

Consider the following example:

```
add_test(NAME MyTest COMMAND MyExe)
set_property(TEST MyTest PROPERTY RESOURCE_GROUPS
  "2,gpus:2"
  "gpus:4,crypto_chips:2")
```

In this example, there are two group descriptions (implicitly separated by a semicolon.) The content of the first description is **2,gpus:2**. This description specifies 2 groups, each of which requires 2 slots from a single GPU. The content of the second description is **gpus:4,crypto_chips:2**. This description does not specify a group count, so a default of 1 is assumed. This single group requires 4 slots from a single GPU and 2 slots from a single cryptography chip. In total, 3 resource groups are specified for this test, each with its own unique requirements.

Note that the number of slots following the resource type specifies slots from a *single* instance of the resource. If the resource group can tolerate receiving slots from different instances of the same resource, it can indicate this by splitting the specification into multiple requirements of one slot. For example:

```
add_test(NAME MyTest COMMAND MyExe)
set_property(TEST MyTest PROPERTY RESOURCE_GROUPS
  "gpus:1,gpus:1,gpus:1,gpus:1")
```

In this case, the single resource group indicates that it needs four GPU slots, all of which may come from separate GPUs (though they don't have to; CTest may still assign slots from the same GPU.)

When CTest sets the environment variables for a test, it assigns a group number based on the group description, starting at 0 on the left and the number of groups minus 1 on the right. For example, in the example above, the two groups in the first description would have IDs of 0 and 1, and the single group in the second description would have an ID of 2.

Both the **RESOURCE_GROUPS** and **RESOURCE_LOCK** properties serve similar purposes, but they are distinct and orthogonal. Resources specified by **RESOURCE_GROUPS** do not affect **RESOURCE_LOCK**, and vice versa. Whereas**RESOURCE_LOCK** is a simpler property that is used for locking one global resource, **RESOURCE_GROUPS** is a more advanced property that allows multiple tests to simultaneously use multiple resources of the same type, specifying their requirements in a fine−grained manner.

RESOURCE_LOCK
Specify a list of resources that are locked by this test.

If multiple tests specify the same resource lock, they are guaranteed not to run concurrently.

See also **FIXTURES_REQUIRED** if the resource requires any setup or cleanup steps.

Both the **RESOURCE_GROUPS** and **RESOURCE_LOCK** properties serve similar purposes, but they are distinct and orthogonal. Resources specified by **RESOURCE_GROUPS** do not affect **RESOURCE_LOCK**, and vice versa. Whereas**RESOURCE_LOCK** is a simpler property that is used for locking one global resource, **RESOURCE_GROUPS** is a more advanced property that allows multiple tests to simultaneously use multiple resources of the same type, specifying their requirements in a fine−grained manner.

RUN_SERIAL
Do not run this test in parallel with any other test.

Use this option in conjunction with the ctest_test **PARALLEL_LEVEL** option to specify that this test should not be run in parallel with any other tests.

SKIP_REGULAR_EXPRESSION
New in version 3.16.

If the output matches this regular expression the test will be marked as skipped.

If set, if the output matches one of specified regular expressions, the test will be marked as skipped. Example:

```
set_property(TEST mytest PROPERTY
  SKIP_REGULAR_EXPRESSION "[^a-z]Skip" "SKIP" "Skipped"
)
```

SKIP_REGULAR_EXPRESSION expects a list of regular expressions.

See also the **SKIP_RETURN_CODE**, **PASS_REGULAR_EXPRESSION**, and **FAIL_REGULAR_EXPRESSION** test properties.

SKIP_RETURN_CODE
Return code to mark a test as skipped.

Sometimes only a test itself can determine if all requirements for the test are met. If such a situation should not be considered a hard failure a return code of the process can be specified that will mark the test as **Not Run** if it is encountered. Valid values are in the range of 0 to 255, inclusive.

See also the **SKIP_REGULAR_EXPRESSION** property.

TIMEOUT
How many seconds to allow for this test.

This property if set will limit a test to not take more than the specified number of seconds to run. If it exceeds that the test process will be killed and ctest will move to the next test. This setting takes precedence over **CTEST_TEST_TIMEOUT**.

TIMEOUT_AFTER_MATCH
New in version 3.6.

Change a test's timeout duration after a matching line is encountered in its output.

Usage

```
add_test(mytest ...)
set_property(TEST mytest PROPERTY TIMEOUT_AFTER_MATCH "${seconds}" "${regex}")
```

Description
Allow a test **seconds** to complete after **regex** is encountered in its output.

When the test outputs a line that matches **regex** its start time is reset to the current time and its timeout duration is changed to **seconds**. Prior to this, the timeout duration is determined by the**TIMEOUT** property or the **CTEST_TEST_TIMEOUT** variable if either of these are set. Because the test's start time is reset, its execution time will not include any time that was spent waiting for the matching output.

TIMEOUT_AFTER_MATCH is useful for avoiding spurious timeouts when your test must wait for some system resource to become available before it can execute. Set**TIMEOUT** to a longer duration that accounts for resource acquisition and use *TIMEOUT_AFTER_MATCH* to control how long the actual test is allowed to run.

If the required resource can be controlled by CTest you should use **RESOURCE_LOCK** instead of *TIMEOUT_AFTER_MATCH*. This property should be used when only the test itself can determine when its required resources are available.

WILL_FAIL
If set to true, this will invert the pass/fail flag of the test.

This property can be used for tests that are expected to fail and return a non zero return code.

WORKING_DIRECTORY
The directory from which the test executable will be called.

If this is not set, the test will be run with the working directory set to the binary directory associated with where the test was created (i.e. the **CMAKE_CURRENT_BINARY_DIR** for where **add_test()** was called).

PROPERTIES ON SOURCE FILES

ABSTRACT
Is this source file an abstract class.

A property on a source file that indicates if the source file represents a class that is abstract. This only makes sense for languages that have a notion of an abstract class and it is only used by some tools that wrap classes into other languages.

AUTORCC_OPTIONS
Additional options for **rcc** when using **AUTORCC**

This property holds additional command line options which will be used when **rcc** is executed during the build via **AUTORCC**, i.e. it is equivalent to the optional **OPTIONS** argument of the **qt4_add_resources()** macro.

By default it is empty.

The options set on the **.qrc** source file may override **AUTORCC_OPTIONS** set on the target.

EXAMPLE

```
# ...
set_property(SOURCE resources.qrc PROPERTY AUTORCC_OPTIONS "--compress;9")
# ...
```

AUTOUIC_OPTIONS
Additional options for **uic** when using **AUTOUIC**

This property holds additional command line options which will be used when **uic** is executed during the build via **AUTOUIC**, i.e. it is equivalent to the optional **OPTIONS** argument of the **qt4_wrap_ui()** macro.

By default it is empty.

The options set on the **.ui** source file may override **AUTOUIC_OPTIONS** set on the target.

EXAMPLE

```
# ...
set_property(SOURCE widget.ui PROPERTY AUTOUIC_OPTIONS "--no-protection")
# ...
```

COMPILE_DEFINITIONS
Preprocessor definitions for compiling a source file.

The **COMPILE_DEFINITIONS** property may be set to a semicolon−separated list of preprocessor definitions using the syntax **VAR** or **VAR=value**. Function−style definitions are not supported. CMake will automatically escape the value correctly for the native build system (note that CMake language syntax may require escapes to specify some values). This property may be set on a per−configuration basis using the name **COMPILE_DEFINITIONS_<CONFIG>** where **<CONFIG>** is an upper−case name (ex. **COMPILE_DEFINITIONS_DEBUG**).

CMake will automatically drop some definitions that are not supported by the native build tool. Xcode does not support per−configuration definitions on source files.

Disclaimer: Most native build tools have poor support for escaping certain values. CMake has work−arounds for many cases but some values may just not be possible to pass correctly. If a value does not seem to be escaped correctly, do not attempt to work−around the problem by adding escape sequences to the value. Your work−around may break in a future version of CMake that has improved escape support. Instead consider defining the macro in a (configured) header file. Then report the limitation. Known limitations include:

```
#          − broken almost everywhere
;          − broken in VS IDE 7.0 and Borland Makefiles
,          − broken in VS IDE
%          − broken in some cases in NMake
& |        − broken in some cases on MinGW
^ < > \"   − broken in most Make tools on Windows
```

CMake does not reject these values outright because they do work in some cases. Use with caution.

Contents of **COMPILE_DEFINITIONS** may use **cmake−generator−expressions(7)** with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions. However, **Xcode** does not support per−config per−source settings, so expressions that depend on the build configuration are not allowed with that generator.

Generator expressions should be preferred instead of setting the alternative per−configuration property.

COMPILE_FLAGS
Additional flags to be added when compiling this source file.

The **COMPILE_FLAGS** property, managed as a string, sets additional compiler flags used that will be added to the list of compile flags when this source file builds. The flags will be added after target−wide flags (except in some cases not supported by the **Visual Studio 9 2008** generator).

Use **COMPILE_DEFINITIONS** to pass additional preprocessor definitions.

Contents of **COMPILE_FLAGS** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. However, **Xcode** does not support per−config per−source settings, so expressions that depend on the build configuration are not allowed with that generator.

NOTE:
This property has been superseded by the **COMPILE_OPTIONS** property.

COMPILE_OPTIONS
New in version 3.11.

List of additional options to pass to the compiler.

This property holds a semicolon−separated list of options and will be added to the list of compile flags when this source file builds. The options will be added after target−wide options (except in some cases not supported by the **Visual Studio 9 2008** generator).

Contents of **COMPILE_OPTIONS** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. However, **Xcode** does not support per−config per−source settings, so expressions that depend on the build configuration are not allowed with that generator.

Usage example:

```
set_source_files_properties(foo.cpp PROPERTIES COMPILE_OPTIONS "−Wno−unused−pa
```

Related properties:

- Prefer this property over **COMPILE_FLAGS**.
- Use **COMPILE_DEFINITIONS** to pass additional preprocessor definitions.
- Use **INCLUDE_DIRECTORIES** to pass additional include directories.

Related commands:

- **add_compile_options()** for directory−wide settings
- **target_compile_options()** for target−specific settings

EXTERNAL_OBJECT
If set to true then this is an object file.

If this property is set to **True** then the source file is really an object file and should not be compiled. It will still be linked into the target though.

Fortran_FORMAT
Set to **FIXED** or **FREE** to indicate the Fortran source layout.

This property tells CMake whether a given Fortran source file uses fixed−format or free−format. CMake will pass the corresponding format flag to the compiler. Consider using the target−wide **Fortran_FORMAT** property if all source files in a target share the same format.

NOTE:
For some compilers, **NAG**, **PGI** and **Solaris Studio**, setting this to **OFF** will have no effect.

Fortran_PREPROCESS
New in version 3.18.

Control whether the Fortran source file should be unconditionally preprocessed.

If unset or empty, rely on the compiler to determine whether the file should be preprocessed. If explicitly set to **OFF** then the file does not need to be preprocessed. If explicitly set to **ON**, then the file does need to be preprocessed as part of the compilation step.

When using the **Ninja** generator, all source files are first preprocessed in order to generate module dependency information. Setting this property to **OFF** will make **Ninja** skip this step.

Consider using the target−wide **Fortran_PREPROCESS** property if all source files in a target need to be

preprocessed.

GENERATED
Is this source file generated as part of the build or CMake process.

Changed in version 3.20: The GENERATED source file property is now visible in all directories.

Tells the internal CMake engine that a source file is generated by an outside process such as another build step, or the execution of CMake itself. This information is then used to exempt the file from any existence or validity checks.

Any file that is

- created by the execution of commands such as **add_custom_command()** and **file(GENERATE)**
- listed as one of the **BYPRODUCTS** of an **add_custom_command()** or **add_custom_target()** command, or
- created by a CMake **AUTOGEN** operation such as **AUTOMOC**, **AUTORCC**, or **AUTOUIC**

will be marked with the **GENERATED** property.

When a generated file created as the **OUTPUT** of an **add_custom_command()** command is explicitly listed as a source file for any target in the same directory scope (which usually means the same **CMakeLists.txt** file), CMake will automatically create a dependency to make sure the file is generated before building that target.

The Makefile Generators will remove **GENERATED** files during **make clean**.

Generated sources may be hidden in some IDE tools, while in others they might be shown. For the special case of sources generated by CMake's **AUTOMOC**, **AUTORCC** or **AUTOUIC** functionality, the **AUTOGEN_SOURCE_GROUP**, **AUTOMOC_SOURCE_GROUP**, **AUTORCC_SOURCE_GROUP** and **AUTOUIC_SOURCE_GROUP** target properties may influence where the generated sources are grouped in the project's file lists.

NOTE:
Starting with CMake 3.20 the **GENERATED** source file property can be set and retrieved from any directory scope. It is an all−or−nothing property. It also can no longer be removed or unset if it was set to **TRUE**. Policy **CMP0118** was introduced to allow supporting the **OLD** behavior for some time.

HEADER_FILE_ONLY
Is this source file only a header file.

A property on a source file that indicates if the source file is a header file with no associated implementation. This is set automatically based on the file extension and is used by CMake to determine if certain dependency information should be computed.

By setting this property to **ON**, you can disable compilation of the given source file, even if it should be compiled because it is part of the library's/executable's sources.

This is useful if you have some source files which you somehow pre−process, and then add these pre−processed sources via **add_library()** or **add_executable()**. Normally, in IDE, there would be no reference of the original sources, only of these pre−processed sources. So by setting this property for all the original source files to **ON**, and then either calling **add_library()** or **add_executable()** while passing both the pre−processed sources and the original sources, or by using **target_sources()** to add original source files will do exactly what would one expect, i.e. the original source files would be visible in IDE, and will not

be built.

INCLUDE_DIRECTORIES
New in version 3.11.

List of preprocessor include file search directories.

This property holds a semicolon−separated list of paths and will be added to the list of include directories when this source file builds. These directories will take precedence over directories defined at target level except for **Xcode** generator due to technical limitations.

Relative paths should not be added to this property directly.

Contents of **INCLUDE_DIRECTORIES** may use "generator expressions" with the syntax **$<...>**. See the **cmake−generator−expressions(7)** manual for available expressions. However, **Xcode** does not support per−config per−source settings, so expressions that depend on the build configuration are not allowed with that generator.

KEEP_EXTENSION
Make the output file have the same extension as the source file.

If this property is set then the file extension of the output file will be the same as that of the source file. Normally the output file extension is computed based on the language of the source file, for example **.cxx** will go to a **.o** extension.

LABELS
Specify a list of text labels associated with a source file.

This property has meaning only when the source file is listed in a target whose **LABELS** property is also set. No other semantics are currently specified.

LANGUAGE
Specify the programming language in which a source file is written.

A property that can be set to indicate what programming language the source file is. If it is not set the language is determined based on the file extension. Typical values are **CXX** (i.e. C++), **C**, **CSharp**, **CUDA**, **Fortran**, **HIP**, **ISPC**, and **ASM**. Setting this property for a file means this file will be compiled. Do not set this for headers or files that should not be compiled.

Changed in version 3.20: Setting this property causes the source file to be compiled as the specified language, using explicit flags if possible. Previously it only caused the specified language's compiler to be used. See policy **CMP0119**.

LOCATION
The full path to a source file.

A read only property on a SOURCE FILE that contains the full path to the source file.

MACOSX_PACKAGE_LOCATION
Place a source file inside a Application Bundle (**MACOSX_BUNDLE**), Core Foundation Bundle (**BUNDLE**), or Framework Bundle (**FRAMEWORK**). It is applicable for macOS and iOS.

Executable targets with the **MACOSX_BUNDLE** property set are built as macOS or iOS application bundles on Apple platforms. Shared library targets with the **FRAMEWORK** property set are built as macOS or iOS frameworks on Apple platforms. Module library targets with the **BUNDLE** property set are built as macOS **CFBundle** bundles on Apple platforms. Source files listed in the target with this property set will

be copied to a directory inside the bundle or framework content folder specified by the property value. For macOS Application Bundles the content folder is **<name>.app/Contents**. For macOS Frameworks the content folder is **<name>.framework/Versions/<version>**. For macOS CFBundles the content folder is **<name>.bundle/Contents** (unless the extension is changed). See the **PUBLIC_HEADER**, **PRIVATE_HEADER**, and **RESOURCE** target properties for specifying files meant for **Headers**, **PrivateHeaders**, or **Resources** directories.

If the specified location is equal to **Resources**, the resulting location will be the same as if the **RESOURCE** property had been used. If the specified location is a sub−folder of **Resources**, it will be placed into the respective sub−folder. Note: For iOS Apple uses a flat bundle layout where no **Resources** folder exist. Therefore CMake strips the **Resources** folder name from the specified location.

OBJECT_DEPENDS
Additional files on which a compiled object file depends.

Specifies a semicolon−separated list of full−paths to files on which any object files compiled from this source file depend. On Makefile Generators and the **Ninja** generator an object file will be recompiled if any of the named files is newer than it. Visual Studio Generators and the **Xcode** generator cannot implement such compilation dependencies.

This property need not be used to specify the dependency of a source file on a generated header file that it includes. Although the property was originally introduced for this purpose, it is no longer necessary. If the generated header file is created by a custom command in the same target as the source file, the automatic dependency scanning process will recognize the dependency. If the generated header file is created by another target, an inter−target dependency should be created with the **add_dependencies()** command (if one does not already exist due to linking relationships).

OBJECT_OUTPUTS
Additional outputs for a **Ninja** or Makefile Generators rule.

Additional outputs created by compilation of this source file. If any of these outputs is missing the object will be recompiled. This is supported only on the **Ninja** and Makefile Generators and will be ignored on other generators.

This property supports **generator expressions**.

SKIP_AUTOGEN
New in version 3.8.

Exclude the source file from **AUTOMOC**, **AUTOUIC** and **AUTORCC** processing (for Qt projects).

For finer exclusion control see **SKIP_AUTOMOC**, **SKIP_AUTOUIC** and **SKIP_AUTORCC**.

EXAMPLE

```
# ...
set_property(SOURCE file.h PROPERTY SKIP_AUTOGEN ON)
# ...
```

SKIP_AUTOMOC
New in version 3.8.

Exclude the source file from **AUTOMOC** processing (for Qt projects).

For broader exclusion control see **SKIP_AUTOGEN**.

EXAMPLE

```
# ...
set_property(SOURCE file.h PROPERTY SKIP_AUTOMOC ON)
# ...
```

SKIP_AUTORCC

New in version 3.8.

Exclude the source file from **AUTORCC** processing (for Qt projects).

For broader exclusion control see **SKIP_AUTOGEN**.

EXAMPLE

```
# ...
set_property(SOURCE file.qrc PROPERTY SKIP_AUTORCC ON)
# ...
```

SKIP_AUTOUIC

New in version 3.8.

Exclude the source file from **AUTOUIC** processing (for Qt projects).

SKIP_AUTOUIC can be set on C++ header and source files and on **.ui** files.

For broader exclusion control see **SKIP_AUTOGEN**.

EXAMPLE

```
# ...
set_property(SOURCE file.h PROPERTY SKIP_AUTOUIC ON)
set_property(SOURCE file.cpp PROPERTY SKIP_AUTOUIC ON)
set_property(SOURCE widget.ui PROPERTY SKIP_AUTOUIC ON)
# ...
```

SKIP_PRECOMPILE_HEADERS

New in version 3.16.

Is this source file skipped by **PRECOMPILE_HEADERS** feature.

This property helps with build problems that one would run into when using the **PRECOMPILE_HEADERS** feature.

One example would be the usage of Objective−C (***.m**) files, and Objective−C++ (***.mm**) files, which lead to compilation failure because they are treated (in case of Ninja / Makefile generator) as C, and CXX respectively. The precompile headers are not compatible between languages.

SKIP_UNITY_BUILD_INCLUSION

New in version 3.16.

Setting this property to true ensures the source file will be skipped by unity builds when its associated target has its **UNITY_BUILD** property set to true. The source file will instead be compiled on its own in the same way as it would with unity builds disabled.

This property helps with "ODR (One definition rule)" problems where combining a particular source file with others might lead to build errors or other unintended side effects.

Swift_DEPENDENCIES_FILE
New in version 3.15.

This property sets the path for the Swift dependency file (swiftdeps) for the source. If one is not specified, it will default to **<OBJECT>.swiftdeps**.

Swift_DIAGNOSTICS_FILE
New in version 3.15.

This property controls where the Swift diagnostics are serialized.

SYMBOLIC
Is this just a name for a rule.

If **SYMBOLIC** (boolean) is set to **True** the build system will be informed that the source file is not actually created on disk but instead used as a symbolic name for a build rule.

UNITY_GROUP
New in version 3.18.

This property controls which *bucket* the source will be part of when the **UNITY_BUILD_MODE** is set to **GROUP**.

VS_COPY_TO_OUT_DIR
New in version 3.8.

Sets the **<CopyToOutputDirectory>** tag for a source file in a Visual Studio project file. Valid values are **Never**, **Always** and **PreserveNewest**.

VS_CSHARP_<tagname>
New in version 3.8.

Visual Studio and CSharp source−file−specific configuration.

Tell the **Visual Studio generators** to set the source file tag **<tagname>** to a given value in the generated Visual Studio CSharp project. Ignored on other generators and languages. This property can be used to define dependencies between source files or set any other Visual Studio specific parameters.

Example usage:

```
set_source_files_property(<filename>
        PROPERTIES
        VS_CSHARP_DependentUpon <other file>
        VS_CSHARP_SubType "Form")
```

VS_DEPLOYMENT_CONTENT
New in version 3.1.

Mark a source file as content for deployment with a Windows Phone or Windows Store application when built with a **Visual Studio generators**. The value must evaluate to either **1** or **0** and may use **generator expressions** to make the choice based on the build configuration. The **.vcxproj** file entry for the source file will be marked either **DeploymentContent** or **ExcludedFromBuild** for values **1** and **0**, respectively.

VS_DEPLOYMENT_LOCATION
New in version 3.1.

Specifies the deployment location for a content source file with a Windows Phone or Windows Store application when built with a **Visual Studio generators**. This property is only applicable when using**VS_DEPLOYMENT_CONTENT**. The value represent the path relative to the app package and applies to all configurations.

VS_INCLUDE_IN_VSIX
New in version 3.8.

Boolean property to specify if the file should be included within a VSIX (Visual Studio Integration Extension) extension package. This is needed for development of Visual Studio extensions.

VS_RESOURCE_GENERATOR
New in version 3.8.

This property allows to specify the resource generator to be used on this file. It defaults to **PublicResXFileCodeGenerator** if not set.

This property only applies to C# projects.

VS_SETTINGS
New in version 3.18.

Set any item metadata on a file.

New in version 3.22: This property is honored for all source file types. Previously it worked only for non−built files.

Takes a list of **Key=Value** pairs. Tells the Visual Studio generator to set **Key** to **Value** as item metadata on the file.

For example:

```
set_property(SOURCE file.hlsl PROPERTY VS_SETTINGS "Key=Value" "Key2=Value2")
```

will set **Key** to **Value** and **Key2** to **Value2** on the **file.hlsl** item as metadata.

Generator expressions are supported.

VS_SHADER_DISABLE_OPTIMIZATIONS
New in version 3.11.

Disable compiler optimizations for an **.hlsl** source file. This adds the **−Od** flag to the command line for the FxCompiler tool. Specify the value **true** for this property to disable compiler optimizations.

VS_SHADER_ENABLE_DEBUG
New in version 3.11.

Enable debugging information for an **.hlsl** source file. This adds the **−Zi** flag to the command line for the FxCompiler tool. Specify the value **true** to generate debugging information for the compiled shader.

VS_SHADER_ENTRYPOINT
New in version 3.1.

Specifies the name of the entry point for the shader of a **.hlsl** source file.

VS_SHADER_FLAGS
New in version 3.2.

Set additional Visual Studio shader flags of a **.hlsl** source file.

VS_SHADER_MODEL
New in version 3.1.

Specifies the shader model of a **.hlsl** source file. Some shader types can only be used with recent shader models

VS_SHADER_OBJECT_FILE_NAME
New in version 3.12.

Specifies a file name for the compiled shader object file for an **.hlsl** source file. This adds the **−Fo** flag to the command line for the FxCompiler tool.

VS_SHADER_OUTPUT_HEADER_FILE
New in version 3.10.

Set filename for output header file containing object code of a **.hlsl** source file.

VS_SHADER_TYPE
New in version 3.1.

Set the Visual Studio shader type of a **.hlsl** source file.

VS_SHADER_VARIABLE_NAME
New in version 3.10.

Set name of variable in header file containing object code of a **.hlsl** source file.

VS_TOOL_OVERRIDE
New in version 3.7.

Override the default Visual Studio tool that will be applied to the source file with a new tool not based on the extension of the file.

VS_XAML_TYPE
New in version 3.3.

Mark a Extensible Application Markup Language (XAML) source file as a different type than the default **Page**. The most common usage would be to set the default **App.xaml** file as **ApplicationDefinition**.

WRAP_EXCLUDE
Exclude this source file from any code wrapping techniques.

Some packages can wrap source files into alternate languages to provide additional functionality.

For example, C++ code can be wrapped into Java or Python, using SWIG. If **WRAP_EXCLUDE** is set to **True**, that indicates that this source file should not be wrapped.

XCODE_EXPLICIT_FILE_TYPE
New in version 3.1.

Set the **Xcode explicitFileType** attribute on its reference to a source file. CMake computes a default based on file extension but can be told explicitly with this property.

See also **XCODE_LAST_KNOWN_FILE_TYPE**.

XCODE_FILE_ATTRIBUTES
New in version 3.7.

Add values to the **Xcode ATTRIBUTES** setting on its reference to a source file. Among other things, this can be used to set the role on a **.mig** file:

```
set_source_files_properties(defs.mig
    PROPERTIES
        XCODE_FILE_ATTRIBUTES "Client;Server"
)
```

XCODE_LAST_KNOWN_FILE_TYPE
New in version 3.1.

Set the **Xcode lastKnownFileType** attribute on its reference to a source file. CMake computes a default based on file extension but can be told explicitly with this property.

See also **XCODE_EXPLICIT_FILE_TYPE**, which is preferred over this property if set.

PROPERTIES ON CACHE ENTRIES

ADVANCED
True if entry should be hidden by default in GUIs.

This is a boolean value indicating whether the entry is considered interesting only for advanced configuration. The**mark_as_adv anced()** command modifies this property.

HELPSTRING
Help associated with entry in GUIs.

This string summarizes the purpose of an entry to help users set it through a CMake GUI.

MODIFIED
Internal management property. Do not set or get.

This is an internal cache entry property managed by CMake to track interactive user modification of entries. Ignore it.

STRINGS
Enumerate possible **STRING** entry values for GUI selection.

For cache entries with type **STRING**, this enumerates a set of values. CMake GUIs may use this to provide a selection widget instead of a generic string entry field. This is for convenience only. CMake does not enforce that the value matches one of those listed.

TYPE
Widget type for entry in GUIs.

Cache entry values are always strings, but CMake GUIs present widgets to help users set values. The GUIs use this property as a hint to determine the widget type. Valid **TYPE** values are:

```
BOOL          = Boolean ON/OFF value.
PATH          = Path to a directory.
FILEPATH      = Path to a file.
STRING        = Generic string value.
INTERNAL      = Do not present in GUI at all.
STATIC        = Value managed by CMake, do not change.
UNINITIALIZED = Type not yet specified.
```

Generally the **TYPE** of a cache entry should be set by the command which creates it (**set()**, **option()**, **find_library()**, etc.).

VALUE
Value of a cache entry.

This property maps to the actual value of a cache entry. Setting this property always sets the value without checking, so use with care.

PROPERTIES ON INSTALLED FILES

CPACK_DESKTOP_SHORTCUTS
New in version 3.3.

Species a list of shortcut names that should be created on the *Desktop* for this file.

The property is currently only supported by the **CPack WIX Generator**.

CPACK_NEVER_OVERWRITE
New in version 3.1.

Request that this file not be overwritten on install or reinstall.

The property is currently only supported by the **CPack WIX Generator**.

CPACK_PERMANENT
New in version 3.1.

Request that this file not be removed on uninstall.

The property is currently only supported by the **CPack WIX Generator**.

CPACK_START_MENU_SHORTCUTS
New in version 3.3.

Species a list of shortcut names that should be created in the *Start Menu* for this file.

The property is currently only supported by the **CPack WIX Generator**.

CPACK_STARTUP_SHORTCUTS
New in version 3.3.

Species a list of shortcut names that should be created in the *Startup* folder for this file.

The property is currently only supported by the **CPack WIX Generator**.

CPACK_WIX_ACL
New in version 3.1.

Specifies access permissions for files or directories installed by a WiX installer.

The property can contain multiple list entries, each of which has to match the following format.

```
<user>[@<domain>]=<permission>[,<permission>]
```

<**user**> and <**domain**> specify the windows user and domain for which the <**Permission**> element should be generated.

<**permission**> is any of the YesNoType attributes listed here:

```
http://wixtoolset.org/documentation/manual/v3/xsd/wix/permission.html
```

The property is currently only supported by the **CPack WIX Generator**.

DEPRECATED PROPERTIES ON DIRECTORIES

ADDITIONAL_MAKE_CLEAN_FILES
Deprecated since version 3.15: Use **ADDITIONAL_CLEAN_FILES** instead.

Additional files to remove during the clean stage.

A ;−list of files that will be removed as a part of the **make clean** target.

Arguments to *ADDITIONAL_MAKE_CLEAN_FILES* may use **generator expressions**.

This property only works for the Makefile generators. It is ignored on other generators.

COMPILE_DEFINITIONS_<CONFIG>
Ignored. See CMake Policy **CMP0043**.

Per−configuration preprocessor definitions in a directory.

This is the configuration−specific version of **COMPILE_DEFINITIONS** where <**CONFIG**> is an upper−case name (ex. **COMPILE_DEFINITIONS_DEBUG**).

This property will be initialized in each directory by its value in the directory's parent.

Contents of **COMPILE_DEFINITIONS_<CONFIG>** may use "generator expressions" with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions. See the **cmake−buildsystem(7)** manual for more on defining buildsystem properties.

Generator expressions should be preferred instead of setting this property.

TEST_INCLUDE_FILE
Deprecated. Use**TEST_INCLUDE_FILES** instead.

A cmake file that will be included when ctest is run.

If you specify **TEST_INCLUDE_FILE**, that file will be included and processed when ctest is run on the directory.

DEPRECATED PROPERTIES ON TARGETS

COMPILE_DEFINITIONS_<CONFIG>
Ignored. See CMake Policy **CMP0043**.

Per−configuration preprocessor definitions on a target.

This is the configuration−specific version of **COMPILE_DEFINITIONS** where **<CONFIG>** is an upper−case name (ex. **COMPILE_DEFINITIONS_DEBUG**).

Contents of **COMPILE_DEFINITIONS_<CONFIG>** may use "generator expressions" with the syntax **$<...>**. See the**cmak e−generator−expressions(7)** manual for available expressions. See the **cmake−buildsystem(7)** manual for more on defining buildsystem properties.

Generator expressions should be preferred instead of setting this property.

POST_INSTALL_SCRIPT
Deprecated install support.

The **PRE_INSTALL_SCRIPT** and **POST_INSTALL_SCRIPT** properties are the old way to specify CMake scripts to run before and after installing a target. They are used only when the old **INSTALL_TARGETS** command is used to install the target. Use the**install()** command instead.

PRE_INSTALL_SCRIPT
Deprecated install support.

The **PRE_INSTALL_SCRIPT** and **POST_INSTALL_SCRIPT** properties are the old way to specify CMake scripts to run before and after installing a target. They are used only when the old **INSTALL_TARGETS** command is used to install the target. Use the**install()** command instead.

DEPRECATED PROPERTIES ON SOURCE FILES

COMPILE_DEFINITIONS_<CONFIG>
Ignored. See CMake Policy **CMP0043**.

Per−configuration preprocessor definitions on a source file.

This is the configuration−specific version of **COMPILE_DEFINITIONS**. Note that**Xcode** does not support per−configuration source file flags so this property will be ignored by the **Xcode** generator.

COPYRIGHT