cmake-buildsystem(7)

Contents

- cmake-buildsystem(7)
  - Introduction
  - Binary Targets
    - Binary Executables
    - Binary Library Types
      - Normal Libraries
        - Apple Frameworks
      - Object Libraries
  - Build Specification and Usage Requirements
    - Target Properties
    - Transitive Usage Requirements
    - Compatible Interface Properties
    - Property Origin Debugging
    - Build Specification with Generator Expressions
      - Include Directories and Usage Requirements
    - Link Libraries and Generator Expressions
    - Output Artifacts
      - Runtime Output Artifacts
      - Library Output Artifacts
      - Archive Output Artifacts
    - Directory-Scoped Commands
  - Build Configurations
    - Case Sensitivity
    - Default And Custom Configurations
  - Pseudo Targets
    - Imported Targets
    - Alias Targets
    - Interface Libraries

Introduction

A CMake-based buildsystem is organized as a set of high-level logical targets. Each target corresponds to an executable or library, or is a custom target containing custom commands. Dependencies between the targets are expressed in the buildsystem to determine the build order and the rules for regeneration in response to change.

Binary Targets

Executables and libraries are defined using the `add_executable()` and `add_library()` commands. The resulting binary files have appropriate `PREFIX`, `SUFFIX` and extensions for the platform targeted. Dependencies between binary targets are expressed using the `target_link_libraries()` command:

```
add_library(archive archive.cpp zip.cpp lzma.cpp)
add_executable(zipapp zipapp.cpp)
target_link_libraries(zipapp archive)
```

`archive` is defined as a `STATIC` library -- an archive containing objects compiled from `archive.cpp`, `zip.cpp`, and `lzma.cpp`. `zipapp` is defined as an executable formed by compiling and linking `zipapp.cpp`. When linking the `zipapp` executable, the `archive` static library is linked in.

Binary Executables

The **`add_executable()`** command defines an executable target:

```
add_executable(mytool mytool.cpp)
```

Commands such as **`add_custom_command()`**, which generates rules to be run at build time can transparently use an **`EXECUTABLE`** target as a `COMMAND` executable. The buildsystem rules will ensure that the executable is built before attempting to run the command.

Binary Library Types

Normal Libraries

By default, the **`add_library()`** command defines a `STATIC` library, unless a type is specified. A type may be specified when using the command:

```
add_library(archive SHARED archive.cpp zip.cpp lzma.cpp)
```

```
add_library(archive STATIC archive.cpp zip.cpp lzma.cpp)
```

The **`BUILD_SHARED_LIBS`** variable may be enabled to change the behavior of **`add_library()`** to build shared libraries by default.

In the context of the buildsystem definition as a whole, it is largely irrelevant whether particular libraries are `SHARED` or `STATIC` -- the commands, dependency specifications and other APIs work similarly regardless of the library type. The `MODULE` library type is dissimilar in that it is generally not linked to -- it is not used in the right-hand-side of the **`target_link_libraries()`** command. It is a type which is loaded as a plugin using runtime techniques. If the library does not export any unmanaged symbols (e.g. Windows resource DLL, C++/CLI DLL), it is required that the library not be a `SHARED` library because CMake expects `SHARED` libraries to export at least one symbol.

```
add_library(archive MODULE 7z.cpp)
```

Apple Frameworks

A `SHARED` library may be marked with the **`FRAMEWORK`** target property to create an macOS or iOS Framework Bundle. A library with the `FRAMEWORK` target property should also set the **`FRAMEWORK_VERSION`** target property. This property is typically set to the value of "A" by macOS conventions. The `MACOSX_FRAMEWORK_IDENTIFIER` sets the `CFBundleIdentifier` key and it uniquely identifies the bundle.

```
add_library(MyFramework SHARED MyFramework.cpp)
set_target_properties(MyFramework PROPERTIES
  FRAMEWORK TRUE
  FRAMEWORK_VERSION A # Version "A" is macOS convention
  MACOSX_FRAMEWORK_IDENTIFIER org.cmake.MyFramework
)
```

Object Libraries

The `OBJECT` library type defines a non-archival collection of object files resulting from compiling the given source files. The object files collection may be used as source inputs to other targets by using the syntax **`$<TARGET_OBJECTS:name>`**. This is a **`generator expression`** that can be used to supply the `OBJECT` library content to other targets:

```
add_library(archive OBJECT archive.cpp zip.cpp lzma.cpp)

add_library(archiveExtras STATIC $<TARGET_OBJECTS:archive> extras.cpp)

add_executable(test_exe $<TARGET_OBJECTS:archive> test.cpp)
```

The link (or archiving) step of those other targets will use the object files collection in addition to those from their own sources.

Alternatively, object libraries may be linked into other targets:

```
add_library(archive OBJECT archive.cpp zip.cpp lzma.cpp)

add_library(archiveExtras STATIC extras.cpp)
target_link_libraries(archiveExtras PUBLIC archive)

add_executable(test_exe test.cpp)
target_link_libraries(test_exe archive)
```

The link (or archiving) step of those other targets will use the object files from `OBJECT` libraries that are *directly* linked. Additionally, usage requirements of the `OBJECT` libraries will be honored when compiling sources in those other targets. Furthermore, those usage requirements will propagate transitively to dependents of those other targets.

Object libraries may not be used as the `TARGET` in a use of the **`add_custom_command(TARGET)`** command signature. However, the list of objects can be used by **`add_custom_command(OUTPUT)`** or **`file(GENERATE)`** by using `$<TARGET_OBJECTS:objlib>`.

Build Specification and Usage Requirements

The **`target_include_directories()`**, **`target_compile_definitions()`** and **`target_compile_options()`** commands specify the build specifications and the usage requirements of binary targets. The commands populate the **`INCLUDE_DIRECTORIES`**, **`COMPILE_DEFINITIONS`** and **`COMPILE_OPTIONS`** target properties respectively, and/or the **`INTERFACE_INCLUDE_DIRECTORIES`**, **`INTERFACE_COMPILE_DEFINITIONS`** and **`INTERFACE_COMPILE_OPTIONS`** target properties.

Each of the commands has a PRIVATE, PUBLIC and INTERFACE mode. The PRIVATE mode populates only the non-INTERFACE_ variant of the target property and the INTERFACE mode populates only the INTERFACE_ variants. The PUBLIC mode populates both variants of the respective target property. Each command may be invoked with multiple uses of each keyword:

```
target_compile_definitions(archive
  PRIVATE BUILDING_WITH_LZMA
  INTERFACE USING_ARCHIVE_LIB
)
```

Note that usage requirements are not designed as a way to make downstreams use particular **COMPILE_OPTIONS** or **COMPILE_DEFINITIONS** etc for convenience only. The contents of the properties must be **requirements**, not merely recommendations or convenience.

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.

Target Properties

The contents of the **INCLUDE_DIRECTORIES**, **COMPILE_DEFINITIONS** and **COMPILE_OPTIONS** target properties are used appropriately when compiling the source files of a binary target.

Entries in the **INCLUDE_DIRECTORIES** are added to the compile line with -I or -isystem prefixes and in the order of appearance in the property value.

Entries in the **COMPILE_DEFINITIONS** are prefixed with -D or /D and added to the compile line in an unspecified order. The **DEFINE_SYMBOL** target property is also added as a compile definition as a special convenience case for SHARED and MODULE library targets.

Entries in the **COMPILE_OPTIONS** are escaped for the shell and added in the order of appearance in the property value. Several compile options have special separate handling, such as **POSITION_INDEPENDENT_CODE**.

The contents of the **INTERFACE_INCLUDE_DIRECTORIES**, **INTERFACE_COMPILE_DEFINITIONS** and **INTERFACE_COMPILE_OPTIONS** target properties are *Usage Requirements* -- they specify content which consumers must use to correctly compile and link with the target they appear on. For any binary target, the contents of each INTERFACE_ property on each target specified in a **target_link_libraries()** command is consumed:

```
set(srcs archive.cpp zip.cpp)
if (LZMA_FOUND)
  list(APPEND srcs lzma.cpp)
endif()
add_library(archive SHARED ${srcs})
if (LZMA_FOUND)
  # The archive library sources are compiled with -DBUILDING_WITH_LZMA
  target_compile_definitions(archive PRIVATE BUILDING_WITH_LZMA)
endif()
target_compile_definitions(archive INTERFACE USING_ARCHIVE_LIB)

add_executable(consumer)
# Link consumer to archive and consume its usage requirements. The consumer
```

```
# executable sources are compiled with -DUSING_ARCHIVE_LIB.
target_link_libraries(consumer archive)
```

Because it is common to require that the source directory and corresponding build directory are added to the **`INCLUDE_DIRECTORIES`**, the **`CMAKE_INCLUDE_CURRENT_DIR`** variable can be enabled to conveniently add the corresponding directories to the **`INCLUDE_DIRECTORIES`** of all targets. The variable **`CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE`** can be enabled to add the corresponding directories to the **`INTERFACE_INCLUDE_DIRECTORIES`** of all targets. This makes use of targets in multiple different directories convenient through use of the **`target_link_libraries()`** command.

Transitive Usage Requirements

The usage requirements of a target can transitively propagate to the dependents. The **`target_link_libraries()`** command has `PRIVATE`, `INTERFACE` and `PUBLIC` keywords to control the propagation.

```
add_library(archive archive.cpp)
target_compile_definitions(archive INTERFACE USING_ARCHIVE_LIB)

add_library(serialization serialization.cpp)
target_compile_definitions(serialization INTERFACE USING_SERIALIZATION_LIB)

add_library(archiveExtras extras.cpp)
target_link_libraries(archiveExtras PUBLIC archive)
target_link_libraries(archiveExtras PRIVATE serialization)
# archiveExtras is compiled with -DUSING_ARCHIVE_LIB
# and -DUSING_SERIALIZATION_LIB

add_executable(consumer consumer.cpp)
# consumer is compiled with -DUSING_ARCHIVE_LIB
target_link_libraries(consumer archiveExtras)
```

Because the `archive` is a `PUBLIC` dependency of `archiveExtras`, the usage requirements of it are propagated to `consumer` too.

Because `serialization` is a `PRIVATE` dependency of `archiveExtras`, the usage requirements of it are not propagated to `consumer`.

Generally, a dependency should be specified in a use of **`target_link_libraries()`** with the `PRIVATE` keyword if it is used by only the implementation of a library, and not in the header files. If a dependency is additionally used in the header files of a library (e.g. for class inheritance), then it should be specified as a `PUBLIC` dependency. A dependency which is not used by the implementation of a library, but only by its headers should be specified as an `INTERFACE` dependency. The **`target_link_libraries()`** command may be invoked with multiple uses of each keyword:

```
target_link_libraries(archiveExtras
  PUBLIC archive
  PRIVATE serialization
)
```

Usage requirements are propagated by reading the `INTERFACE_` variants of target properties from dependencies and appending the values to the non-`INTERFACE_` variants of the operand. For example, the **`INTERFACE_INCLUDE_DIRECTORIES`** of dependencies is read and appended to the

INCLUDE_DIRECTORIES of the operand. In cases where order is relevant and maintained, and the order resulting from the **target_link_libraries()** calls does not allow correct compilation, use of an appropriate command to set the property directly may update the order.

For example, if the linked libraries for a target must be specified in the order `lib1 lib2 lib3` , but the include directories must be specified in the order `lib3 lib1 lib2`:

```
target_link_libraries(myExe lib1 lib2 lib3)
target_include_directories(myExe
  PRIVATE $<TARGET_PROPERTY:lib3,INTERFACE_INCLUDE_DIRECTORIES>)
```

Note that care must be taken when specifying usage requirements for targets which will be exported for installation using the **install(EXPORT)** command. See Creating Packages for more.

Compatible Interface Properties

Some target properties are required to be compatible between a target and the interface of each dependency. For example, the **POSITION_INDEPENDENT_CODE** target property may specify a boolean value of whether a target should be compiled as position-independent-code, which has platform-specific consequences. A target may also specify the usage requirement **INTERFACE_POSITION_INDEPENDENT_CODE** to communicate that consumers must be compiled as position-independent-code.

```
add_executable(exe1 exe1.cpp)
set_property(TARGET exe1 PROPERTY POSITION_INDEPENDENT_CODE ON)

add_library(lib1 SHARED lib1.cpp)
set_property(TARGET lib1 PROPERTY INTERFACE_POSITION_INDEPENDENT_CODE ON)

add_executable(exe2 exe2.cpp)
target_link_libraries(exe2 lib1)
```

Here, both `exe1` and `exe2` will be compiled as position-independent-code. `lib1` will also be compiled as position-independent-code because that is the default setting for `SHARED` libraries. If dependencies have conflicting, non-compatible requirements **cmake(1)** issues a diagnostic:

```
add_library(lib1 SHARED lib1.cpp)
set_property(TARGET lib1 PROPERTY INTERFACE_POSITION_INDEPENDENT_CODE ON)

add_library(lib2 SHARED lib2.cpp)
set_property(TARGET lib2 PROPERTY INTERFACE_POSITION_INDEPENDENT_CODE OFF)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 lib1)
set_property(TARGET exe1 PROPERTY POSITION_INDEPENDENT_CODE OFF)

add_executable(exe2 exe2.cpp)
target_link_libraries(exe2 lib1 lib2)
```

The `lib1` requirement `INTERFACE_POSITION_INDEPENDENT_CODE` is not "compatible" with the **POSITION_INDEPENDENT_CODE** property of the `exe1` target. The library requires that consumers are built as position-independent-code, while the executable specifies to not built as position-independent-code, so a diagnostic is issued.

The `lib1` and `lib2` requirements are not "compatible". One of them requires that consumers are built as position-independent-code, while the other requires that consumers are not built as position-independent-code. Because `exe2` links to both and they are in conflict, a CMake error message is issued:

```
CMake Error: The INTERFACE_POSITION_INDEPENDENT_CODE property of "lib2" does
not agree with the value of POSITION_INDEPENDENT_CODE already determined
for "exe2".
```

To be "compatible", the **POSITION_INDEPENDENT_CODE** property, if set must be either the same, in a boolean sense, as the **INTERFACE_POSITION_INDEPENDENT_CODE** property of all transitively specified dependencies on which that property is set.

This property of "compatible interface requirement" may be extended to other properties by specifying the property in the content of the **COMPATIBLE_INTERFACE_BOOL** target property. Each specified property must be compatible between the consuming target and the corresponding property with an `INTERFACE_` prefix from each dependency:

```
add_library(lib1Version2 SHARED lib1_v2.cpp)
set_property(TARGET lib1Version2 PROPERTY INTERFACE_CUSTOM_PROP ON)
set_property(TARGET lib1Version2 APPEND PROPERTY
  COMPATIBLE_INTERFACE_BOOL CUSTOM_PROP
)

add_library(lib1Version3 SHARED lib1_v3.cpp)
set_property(TARGET lib1Version3 PROPERTY INTERFACE_CUSTOM_PROP OFF)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 lib1Version2) # CUSTOM_PROP will be ON

add_executable(exe2 exe2.cpp)
target_link_libraries(exe2 lib1Version2 lib1Version3) # Diagnostic
```

Non-boolean properties may also participate in "compatible interface" computations. Properties specified in the **COMPATIBLE_INTERFACE_STRING** property must be either unspecified or compare to the same string among all transitively specified dependencies. This can be useful to ensure that multiple incompatible versions of a library are not linked together through transitive requirements of a target:

```
add_library(lib1Version2 SHARED lib1_v2.cpp)
set_property(TARGET lib1Version2 PROPERTY INTERFACE_LIB_VERSION 2)
set_property(TARGET lib1Version2 APPEND PROPERTY
  COMPATIBLE_INTERFACE_STRING LIB_VERSION
)

add_library(lib1Version3 SHARED lib1_v3.cpp)
set_property(TARGET lib1Version3 PROPERTY INTERFACE_LIB_VERSION 3)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 lib1Version2) # LIB_VERSION will be "2"

add_executable(exe2 exe2.cpp)
target_link_libraries(exe2 lib1Version2 lib1Version3) # Diagnostic
```

The **COMPATIBLE_INTERFACE_NUMBER_MAX** target property specifies that content will be evaluated numerically and the maximum number among all specified will be calculated:

```
add_library(lib1Version2 SHARED lib1_v2.cpp)
set_property(TARGET lib1Version2 PROPERTY INTERFACE_CONTAINER_SIZE_REQUIRED 200)
set_property(TARGET lib1Version2 APPEND PROPERTY
  COMPATIBLE_INTERFACE_NUMBER_MAX CONTAINER_SIZE_REQUIRED
)

add_library(lib1Version3 SHARED lib1_v3.cpp)
set_property(TARGET lib1Version3 PROPERTY INTERFACE_CONTAINER_SIZE_REQUIRED 1000)

add_executable(exe1 exe1.cpp)
# CONTAINER_SIZE_REQUIRED will be "200"
target_link_libraries(exe1 lib1Version2)

add_executable(exe2 exe2.cpp)
# CONTAINER_SIZE_REQUIRED will be "1000"
target_link_libraries(exe2 lib1Version2 lib1Version3)
```

Similarly, the **`COMPATIBLE_INTERFACE_NUMBER_MIN`** may be used to calculate the numeric minimum value for a property from dependencies.

Each calculated "compatible" property value may be read in the consumer at generate-time using generator expressions.

Note that for each dependee, the set of properties specified in each compatible interface property must not intersect with the set specified in any of the other properties.

Property Origin Debugging

Because build specifications can be determined by dependencies, the lack of locality of code which creates a target and code which is responsible for setting build specifications may make the code more difficult to reason about. **`cmake(1)`** provides a debugging facility to print the origin of the contents of properties which may be determined by dependencies. The properties which can be debugged are listed in the **`CMAKE_DEBUG_TARGET_PROPERTIES`** variable documentation:

```
set(CMAKE_DEBUG_TARGET_PROPERTIES
  INCLUDE_DIRECTORIES
  COMPILE_DEFINITIONS
  POSITION_INDEPENDENT_CODE
  CONTAINER_SIZE_REQUIRED
  LIB_VERSION
)
add_executable(exe1 exe1.cpp)
```

In the case of properties listed in **`COMPATIBLE_INTERFACE_BOOL`** or **`COMPATIBLE_INTERFACE_STRING`**, the debug output shows which target was responsible for setting the property, and which other dependencies also defined the property. In the case of **`COMPATIBLE_INTERFACE_NUMBER_MAX`** and **`COMPATIBLE_INTERFACE_NUMBER_MIN`**, the debug output shows the value of the property from each dependency, and whether the value determines the new extreme.

Build Specification with Generator Expressions

Build specifications may use **`generator expressions`** containing content which may be conditional or known only at generate-time. For example, the calculated "compatible" value of a property may be read with the `TARGET_PROPERTY` expression:

```
add_library(lib1Version2 SHARED lib1_v2.cpp)
set_property(TARGET lib1Version2 PROPERTY
  INTERFACE_CONTAINER_SIZE_REQUIRED 200)
set_property(TARGET lib1Version2 APPEND PROPERTY
  COMPATIBLE_INTERFACE_NUMBER_MAX CONTAINER_SIZE_REQUIRED
)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 lib1Version2)
target_compile_definitions(exe1 PRIVATE
    CONTAINER_SIZE=$<TARGET_PROPERTY:CONTAINER_SIZE_REQUIRED>
)
```

In this case, the `exe1` source files will be compiled with `-DCONTAINER_SIZE=200`.

The unary `TARGET_PROPERTY` generator expression and the `TARGET_POLICY` generator expression are evaluated with the consuming target context. This means that a usage requirement specification may be evaluated differently based on the consumer:

```
add_library(lib1 lib1.cpp)
target_compile_definitions(lib1 INTERFACE
  $<$<STREQUAL:$<TARGET_PROPERTY:TYPE>,EXECUTABLE>:LIB1_WITH_EXE>
  $<$<STREQUAL:$<TARGET_PROPERTY:TYPE>,SHARED_LIBRARY>:LIB1_WITH_SHARED_LIB>
  $<$<TARGET_POLICY:CMP0041>:CONSUMER_CMP0041_NEW>
)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 lib1)

cmake_policy(SET CMP0041 NEW)

add_library(shared_lib shared_lib.cpp)
target_link_libraries(shared_lib lib1)
```

The `exe1` executable will be compiled with `-DLIB1_WITH_EXE`, while the `shared_lib` shared library will be compiled with `-DLIB1_WITH_SHARED_LIB` and `-DCONSUMER_CMP0041_NEW`, because policy **`CMP0041`** is `NEW` at the point where the `shared_lib` target is created.

The `BUILD_INTERFACE` expression wraps requirements which are only used when consumed from a target in the same buildsystem, or when consumed from a target exported to the build directory using the **`export()`** command. The `INSTALL_INTERFACE` expression wraps requirements which are only used when consumed from a target which has been installed and exported with the **`install(EXPORT)`** command:

```
add_library(ClimbingStats climbingstats.cpp)
target_compile_definitions(ClimbingStats INTERFACE
  $<BUILD_INTERFACE:ClimbingStats_FROM_BUILD_LOCATION>
  $<INSTALL_INTERFACE:ClimbingStats_FROM_INSTALLED_LOCATION>
)
install(TARGETS ClimbingStats EXPORT libExport ${InstallArgs})
install(EXPORT libExport NAMESPACE Upstream::
        DESTINATION lib/cmake/ClimbingStats)
export(EXPORT libExport NAMESPACE Upstream::)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 ClimbingStats)
```

In this case, the `exe1` executable will be compiled with `-DClimbingStats_FROM_BUILD_LOCATION`. The exporting commands generate **`IMPORTED`** targets with either the `INSTALL_INTERFACE` or the

BUILD_INTERFACE omitted, and the *_INTERFACE marker stripped away. A separate project consuming the ClimbingStats package would contain:

```
find_package(ClimbingStats REQUIRED)

add_executable(Downstream main.cpp)
target_link_libraries(Downstream Upstream::ClimbingStats)
```

Depending on whether the ClimbingStats package was used from the build location or the install location, the Downstream target would be compiled with either -DClimbingStats_FROM_BUILD_LOCATION or -DClimbingStats_FROM_INSTALL_LOCATION. For more about packages and exporting see the **cmake-packages(7)** manual.

Include Directories and Usage Requirements

Include directories require some special consideration when specified as usage requirements and when used with generator expressions. The **target_include_directories()** command accepts both relative and absolute include directories:

```
add_library(lib1 lib1.cpp)
target_include_directories(lib1 PRIVATE
  /absolute/path
  relative/path
)
```

Relative paths are interpreted relative to the source directory where the command appears. Relative paths are not allowed in the **INTERFACE_INCLUDE_DIRECTORIES** of **IMPORTED** targets.

In cases where a non-trivial generator expression is used, the INSTALL_PREFIX expression may be used within the argument of an INSTALL_INTERFACE expression. It is a replacement marker which expands to the installation prefix when imported by a consuming project.

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:

```
add_library(ClimbingStats climbingstats.cpp)
target_include_directories(ClimbingStats INTERFACE
  $<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}/generated>
  $<INSTALL_INTERFACE:/absolute/path>
  $<INSTALL_INTERFACE:relative/path>
  $<INSTALL_INTERFACE:$<INSTALL_PREFIX>/$<CONFIG>/generated>
)
```

Two convenience APIs are provided relating to include directories usage requirements. The **CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE** variable may be enabled, with an equivalent effect to:

```
set_property(TARGET tgt APPEND PROPERTY INTERFACE_INCLUDE_DIRECTORIES
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR};${CMAKE_CURRENT_BINARY_DIR}>
)
```

for each target affected. The convenience for installed targets is an `INCLUDES DESTINATION` component with the **`install(TARGETS)`** command:

```
install(TARGETS foo bar bat EXPORT tgts ${dest_args}
  INCLUDES DESTINATION include
)
install(EXPORT tgts ${other_args})
install(FILES ${headers} DESTINATION include)
```

This is equivalent to appending `${CMAKE_INSTALL_PREFIX}/include` to the **`INTERFACE_INCLUDE_DIRECTORIES`** of each of the installed **`IMPORTED`** targets when generated by **`install(EXPORT)`**.

When the **`INTERFACE_INCLUDE_DIRECTORIES`** of an imported target is consumed, the entries in the property may be treated as system include directories. The effects of that are toolchain-dependent, but one common effect is to omit compiler warnings for headers found in those directories. The **`SYSTEM`** property of the installed target determines this behavior (see the **`EXPORT_NO_SYSTEM`** property for how to modify the installed value for a target). It is also possible to change how consumers interpret the system behavior of consumed imported targets by setting the **`NO_SYSTEM_FROM_IMPORTED`** target property on the *consumer*.

If a binary target is linked transitively to a macOS **`FRAMEWORK`**, the `Headers` directory of the framework is also treated as a usage requirement. This has the same effect as passing the framework directory as an include directory.

Link Libraries and Generator Expressions

Like build specifications, **`link libraries`** may be specified with generator expression conditions. However, as consumption of usage requirements is based on collection from linked dependencies, there is an additional limitation that the link dependencies must form a "directed acyclic graph". That is, if linking to a target is dependent on the value of a target property, that target property may not be dependent on the linked dependencies:

```
add_library(lib1 lib1.cpp)
add_library(lib2 lib2.cpp)
target_link_libraries(lib1 PUBLIC
  $<$<TARGET_PROPERTY:POSITION_INDEPENDENT_CODE>:lib2>
)
add_library(lib3 lib3.cpp)
set_property(TARGET lib3 PROPERTY INTERFACE_POSITION_INDEPENDENT_CODE ON)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 lib1 lib3)
```

As the value of the **`POSITION_INDEPENDENT_CODE`** property of the `exe1` target is dependent on the linked libraries (`lib3`), and the edge of linking `exe1` is determined by the same **`POSITION_INDEPENDENT_CODE`** property, the dependency graph above contains a cycle. **`cmake(1)`** issues an error message.

Output Artifacts

The buildsystem targets created by the **`add_library()`** and **`add_executable()`** commands create rules to create binary outputs. The exact output location of the binaries can only be determined at generate-time because it can depend on the build-configuration and the link-language of linked dependencies etc. `TARGET_FILE`, `TARGET_LINKER_FILE` and related expressions can be used to access the name and location of generated binaries. These expressions do not work for `OBJECT` libraries however, as there is no single file generated by such libraries which is relevant to the expressions.

There are three kinds of output artifacts that may be build by targets as detailed in the following sections. Their classification differs between DLL platforms and non-DLL platforms. All Windows-based systems including Cygwin are DLL platforms.

Runtime Output Artifacts

A *runtime* output artifact of a buildsystem target may be:

- The executable file (e.g. `.exe`) of an executable target created by the **`add_executable()`** command.
- On DLL platforms: the executable file (e.g. `.dll`) of a shared library target created by the **`add_library()`** command with the `SHARED` option.

The **`RUNTIME_OUTPUT_DIRECTORY`** and **`RUNTIME_OUTPUT_NAME`** target properties may be used to control runtime output artifact locations and names in the build tree.

Library Output Artifacts

A *library* output artifact of a buildsystem target may be:

- The loadable module file (e.g. `.dll` or `.so`) of a module library target created by the **`add_library()`** command with the `MODULE` option.
- On non-DLL platforms: the shared library file (e.g. `.so` or `.dylib`) of a shared library target created by the **`add_library()`** command with the `SHARED` option.

The **`LIBRARY_OUTPUT_DIRECTORY`** and **`LIBRARY_OUTPUT_NAME`** target properties may be used to control library output artifact locations and names in the build tree.

Archive Output Artifacts

An *archive* output artifact of a buildsystem target may be:

- The static library file (e.g. `.lib` or `.a`) of a static library target created by the **`add_library()`** command with the `STATIC` option.
- On DLL platforms: the import library file (e.g. `.lib`) of a shared library target created by the **`add_library()`** command with the `SHARED` option. This file is only guaranteed to exist if the library exports at least one unmanaged symbol.
- On DLL platforms: the import library file (e.g. `.lib`) of an executable target created by the **`add_executable()`** command when its **`ENABLE_EXPORTS`** target property is set.
- On AIX: the linker import file (e.g. `.imp`) of an executable target created by the **`add_executable()`** command when its **`ENABLE_EXPORTS`** target property is set.
- On macOS: the linker import file (e.g. `.tbd`) of a shared library target created by the **`add_library()`** command with the `SHARED` option and when its **`ENABLE_EXPORTS`** target property is set.

The `ARCHIVE_OUTPUT_DIRECTORY` and `ARCHIVE_OUTPUT_NAME` target properties may be used to control archive output artifact locations and names in the build tree.

Directory-Scoped Commands

The `target_include_directories()`, `target_compile_definitions()` and `target_compile_options()` commands have an effect on only one target at a time. The commands `add_compile_definitions()`, `add_compile_options()` and `include_directories()` have a similar function, but operate at directory scope instead of target scope for convenience.

Build Configurations

Configurations determine specifications for a certain type of build, such as `Release` or `Debug`. The way this is specified depends on the type of `generator` being used. For single configuration generators like Makefile Generators and `Ninja`, the configuration is specified at configure time by the `CMAKE_BUILD_TYPE` variable. For multi-configuration generators like Visual Studio, `Xcode`, and `Ninja Multi-Config`, the configuration is chosen by the user at build time and `CMAKE_BUILD_TYPE` is ignored. In the multi-configuration case, the set of *available* configurations is specified at configure time by the `CMAKE_CONFIGURATION_TYPES` variable, but the actual configuration used cannot be known until the build stage. This difference is often misunderstood, leading to problematic code like the following:

```
# WARNING: This is wrong for multi-config generators because they don't use
#          and typically don't even set CMAKE_BUILD_TYPE
string(TOLOWER ${CMAKE_BUILD_TYPE} build_type)
if (build_type STREQUAL debug)
  target_compile_definitions(exe1 PRIVATE DEBUG_BUILD)
endif()
```

`Generator expressions` should be used instead to handle configuration-specific logic correctly, regardless of the generator used. For example:

```
# Works correctly for both single and multi-config generators
target_compile_definitions(exe1 PRIVATE
  $<$<CONFIG:Debug>:DEBUG_BUILD>
)
```

In the presence of `IMPORTED` targets, the content of `MAP_IMPORTED_CONFIG_DEBUG` is also accounted for by the above `$<CONFIG:Debug>` expression.

Case Sensitivity

`CMAKE_BUILD_TYPE` and `CMAKE_CONFIGURATION_TYPES` are just like other variables in that any string comparisons made with their values will be case-sensitive. The `$<CONFIG>` generator expression also preserves the casing of the configuration as set by the user or CMake defaults. For example:

```
# NOTE: Don't use these patterns, they are for illustration purposes only.

set(CMAKE_BUILD_TYPE Debug)
if(CMAKE_BUILD_TYPE STREQUAL DEBUG)
  # ... will never get here, "Debug" != "DEBUG"
```

```
endif()
add_custom_target(print_config ALL
  # Prints "Config is Debug" in this single-config case
  COMMAND ${CMAKE_COMMAND} -E echo "Config is $<CONFIG>"
  VERBATIM
)

set(CMAKE_CONFIGURATION_TYPES Debug Release)
if(DEBUG IN_LIST CMAKE_CONFIGURATION_TYPES)
  # ... will never get here, "Debug" != "DEBUG"
endif()
```

In contrast, CMake treats the configuration type case-insensitively when using it internally in places that modify behavior based on the configuration. For example, the **`$<CONFIG:Debug>`** generator expression will evaluate to 1 for a configuration of not only `Debug`, but also `DEBUG`, `debug` or even `DeBuG`. Therefore, you can specify configuration types in **`CMAKE_BUILD_TYPE`** and **`CMAKE_CONFIGURATION_TYPES`** with any mixture of upper and lowercase, although there are strong conventions (see the next section). If you must test the value in string comparisons, always convert the value to upper or lowercase first and adjust the test accordingly.

Default And Custom Configurations

By default, CMake defines a number of standard configurations:

- `Debug`
- `Release`
- `RelWithDebInfo`
- `MinSizeRel`

In multi-config generators, the **`CMAKE_CONFIGURATION_TYPES`** variable will be populated with (potentially a subset of) the above list by default, unless overridden by the project or user. The actual configuration used is selected by the user at build time.

For single-config generators, the configuration is specified with the **`CMAKE_BUILD_TYPE`** variable at configure time and cannot be changed at build time. The default value will often be none of the above standard configurations and will instead be an empty string. A common misunderstanding is that this is the same as `Debug`, but that is not the case. Users should always explicitly specify the build type instead to avoid this common problem.

The above standard configuration types provide reasonable behavior on most platforms, but they can be extended to provide other types. Each configuration defines a set of compiler and linker flag variables for the language in use. These variables follow the convention **`CMAKE_<LANG>_FLAGS_<CONFIG>`**, where `<CONFIG>` is always the uppercase configuration name. When defining a custom configuration type, make sure these variables are set appropriately, typically as cache variables.

Pseudo Targets

Some target types do not represent outputs of the buildsystem, but only inputs such as external dependencies, aliases or other non-build artifacts. Pseudo targets are not represented in the generated buildsystem.

Imported Targets

An **IMPORTED** target represents a pre-existing dependency. Usually such targets are defined by an upstream package and should be treated as immutable. After declaring an **IMPORTED** target one can adjust its target properties by using the customary commands such as **target_compile_definitions()**, **target_include_directories()**, **target_compile_options()** or **target_link_libraries()** just like with any other regular target.

IMPORTED targets may have the same usage requirement properties populated as binary targets, such as **INTERFACE_INCLUDE_DIRECTORIES**, **INTERFACE_COMPILE_DEFINITIONS**, **INTERFACE_COMPILE_OPTIONS**, **INTERFACE_LINK_LIBRARIES**, and **INTERFACE_POSITION_INDEPENDENT_CODE**.

The **LOCATION** may also be read from an IMPORTED target, though there is rarely reason to do so. Commands such as **add_custom_command()** can transparently use an **IMPORTED EXECUTABLE** target as a `COMMAND` executable.

The scope of the definition of an **IMPORTED** target is the directory where it was defined. It may be accessed and used from subdirectories, but not from parent directories or sibling directories. The scope is similar to the scope of a cmake variable.

It is also possible to define a `GLOBAL` **IMPORTED** target which is accessible globally in the buildsystem.

See the **cmake-packages(7)** manual for more on creating packages with **IMPORTED** targets.

Alias Targets

An `ALIAS` target is a name which may be used interchangeably with a binary target name in read-only contexts. A primary use-case for `ALIAS` targets is for example or unit test executables accompanying a library, which may be part of the same buildsystem or built separately based on user configuration.

```
add_library(lib1 lib1.cpp)
install(TARGETS lib1 EXPORT lib1Export ${dest_args})
install(EXPORT lib1Export NAMESPACE Upstream:: ${other_args})

add_library(Upstream::lib1 ALIAS lib1)
```

In another directory, we can link unconditionally to the `Upstream::lib1` target, which may be an **IMPORTED** target from a package, or an `ALIAS` target if built as part of the same buildsystem.

```
if (NOT TARGET Upstream::lib1)
  find_package(lib1 REQUIRED)
endif()
add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 Upstream::lib1)
```

`ALIAS` targets are not mutable, installable or exportable. They are entirely local to the buildsystem description. A name can be tested for whether it is an `ALIAS` name by reading the **ALIASED_TARGET** property from it:

```
get_target_property(_aliased Upstream::lib1 ALIASED_TARGET)
if(_aliased)
  message(STATUS "The name Upstream::lib1 is an ALIAS for ${_aliased}.")
endif()
```

Interface Libraries

An `INTERFACE` library target does not compile sources and does not produce a library artifact on disk, so it has no **`LOCATION`**.

It may specify usage requirements such as **`INTERFACE_INCLUDE_DIRECTORIES`**, **`INTERFACE_COMPILE_DEFINITIONS`**, **`INTERFACE_COMPILE_OPTIONS`**, **`INTERFACE_LINK_LIBRARIES`**, **`INTERFACE_SOURCES`**, and **`INTERFACE_POSITION_INDEPENDENT_CODE`**. Only the `INTERFACE` modes of the **`target_include_directories()`**, **`target_compile_definitions()`**, **`target_compile_options()`**, **`target_sources()`**, and **`target_link_libraries()`** commands may be used with `INTERFACE` libraries.

Since CMake 3.19, an `INTERFACE` library target may optionally contain source files. An interface library that contains source files will be included as a build target in the generated buildsystem. It does not compile sources, but may contain custom commands to generate other sources. Additionally, IDEs will show the source files as part of the target for interactive reading and editing.

A primary use-case for `INTERFACE` libraries is header-only libraries. Since CMake 3.23, header files may be associated with a library by adding them to a header set using the **`target_sources()`** command:

```
add_library(Eigen INTERFACE)

target_sources(Eigen PUBLIC
  FILE_SET HEADERS
    BASE_DIRS src
    FILES src/eigen.h src/vector.h src/matrix.h
)

add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 Eigen)
```

When we specify the `FILE_SET` here, the `BASE_DIRS` we define automatically become include directories in the usage requirements for the target `Eigen`. The usage requirements from the target are consumed and used when compiling, but have no effect on linking.

Another use-case is to employ an entirely target-focussed design for usage requirements:

```
add_library(pic_on INTERFACE)
set_property(TARGET pic_on PROPERTY INTERFACE_POSITION_INDEPENDENT_CODE ON)
add_library(pic_off INTERFACE)
set_property(TARGET pic_off PROPERTY INTERFACE_POSITION_INDEPENDENT_CODE OFF)

add_library(enable_rtti INTERFACE)
target_compile_options(enable_rtti INTERFACE
  $<$<OR:$<COMPILER_ID:GNU>,$<COMPILER_ID:Clang>>:-rtti>
)
```

```
add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 pic_on enable_rtti)
```

This way, the build specification of `exe1` is expressed entirely as linked targets, and the complexity of compiler-specific flags is encapsulated in an `INTERFACE` library target.

`INTERFACE` libraries may be installed and exported. We can install the default header set along with the target:

```
add_library(Eigen INTERFACE)

target_sources(Eigen INTERFACE
  FILE_SET HEADERS
    BASE_DIRS src
    FILES src/eigen.h src/vector.h src/matrix.h
)

install(TARGETS Eigen EXPORT eigenExport
  FILE_SET HEADERS DESTINATION include/Eigen)
install(EXPORT eigenExport NAMESPACE Upstream::
  DESTINATION lib/cmake/Eigen
)
```

Here, the headers defined in the header set are installed to `include/Eigen`. The install destination automatically becomes an include directory that is a usage requirement for consumers.