bcm Documentation

Release 0.1

Paul Fultz II

Jun 11, 2018

Contents

Paul Fultz II

CHAPTER 1

Motivation

This provides cmake modules that can be re-used by boost and other dependencies. It provides modules to reduce the boilerplate for installing, versioning, setting up package config, and creating tests.

CHAPTER 2

Usage

The modules can be installed using standard cmake install:

```
mkdir build
cd build
cmake ..
cmake --build . --target install
```

Once installed, the modules can be used by using `find_package` and then including the appropriate module:

```
find_package(BCM)
include(BCMDeploy)
```

CHAPTER 3

Quick Start

3.1 Building a boost library

The BCM modules provide some high-level cmake functions to take care of all the cmake boilerplate needed to build, install and configuration setup. To setup a simple boost library we can do:

```
cmake_minimum_required (VERSION 3.5)
project(boost_config)

find_package(BCM)
include(BCMDeploy)
include(BCMSetupVersion)

bcm_setup_version(VERSION 1.64.0)

add_library(boost_config INTERFACE)
add_library(boost::config ALIAS boost_config)
set_property(TARGET boost_config PROPERTY EXPORT_NAME config)

bcm_deploy(TARGETS config INCLUDE include)
```

This sets up the Boost.Config cmake with the version `1.64.0`. More importantly the user can now install the library, like this:

```
mkdir build
cd build
cmake ..
cmake --build . --target install
```

And then the user can build with Boost.Config using cmake's `find_package`:

```
project(foo)

find_package(boost_config)
```

(continues on next page)

(continued from previous page)

```
add_executable(foo foo.cpp)
target_link_libraries(foo boost::config)
```

Or if the user isn't using cmake, then `pkg-config` can be used instead:

```
g++ `pkg-config boost_config --cflags --libs` foo.cpp
```

3.2 Tests

The BCM modules provide functions for creating tests that integrate into cmake's ctest infrastructure. All tests can be built and ran using `make check`. The `bcm_test` function can add a test to be ran:

```
bcm_test(NAME config_test_c SOURCES config_test_c.c)
```

This will compile the `SOURCES` and run them. The test also needs to link in `boost_config`. This can be done with `target_link_libraries`:

```
target_link_libraries(config_test_c boost::config)
```

Or all tests in the directory can be set using `bcm_test_link_libraries`:

```
bcm_test_link_libraries(boost::config)
```

And all tests in the directory will use `boost::config`.

Also, tests can be specified as compile-only or as expected to fail:

```
bcm_test(NAME test_thread_fail1 SOURCES threads/test_thread_fail1.cpp COMPILE_ONLY␣
↪WILL_FAIL)
```

CHAPTER 4

Building

There are two scenarios where the users will consume their dependencies in the build:

- Prebuilt binaries using `find_package`
- Integrated builds using `add_subdirectory`

When we build libraries using cmake, we want to be able to support both scenarios.

The first scenario the user would build and install each dependency. With this scenario, we need to generate usage requirements that can be consumed by the user, and ultimately this is done through cmake's `find_package` mechanism.

In the integrated build scenario, the user adds the sources with `add_subdirectory`, and then all dependencies are built in the user's build. There is no need to generate usage requirements as the cmake targets are directly available in the build.

Let's first look at standalone build.

4.1 Building standalone with cmake

Let's look at building a library like Boost.Filesystem using just cmake. When we start a cmake, we start with minimuim requirement and the project name:

```
cmake_minimum_required(VERSION 3.5)
project(boost_filesystem)
```

Then we can define the library and the sources it will build:

```
add_library(boost_filesystem
    src/operations.cpp
    src/portability.cpp
    src/codecvt_error_category.cpp
    src/utf8_codecvt_facet.cpp
    src/windows_file_codecvt.cpp
```

(continues on next page)

(continued from previous page)

```
    src/unique_path.cpp
    src/path.cpp
    src/path_traits.cpp
)
```

So this will build the library named `boost_filesystem`, however, we need to supply the dependencies to `boost_filesystem` and add the include directories. To add the include directory we use `target_include_directories`. For this, we tell cmake to use local `include` directory, but since this is only valid during build and not after installation, we use the `BUILD_INTERFACE` generator expression so that cmake will only use it during build and not installation:

```
target_include_directories(boost_filesystem PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
)
```

Using `PUBLIC` means this include directory will be used internally to build, and downstream users need this include as well. Next, we need to pull in the dependencies. To do this, we call `find_package`, and for the sake of the turtorial we assume that the upstream boost libraries have already set this up:

```
find_package(boost_core)
find_package(boost_static_assert)
find_package(boost_iterator)
find_package(boost_detail)
find_package(boost_system)
find_package(boost_functional)
find_package(boost_assert)
find_package(boost_range)
find_package(boost_type_traits)
find_package(boost_smart_ptr)
find_package(boost_io)
find_package(boost_config)
```

Calling `find_package` will find those libraries and provide a target we can use to link against. The next step is to link it using `target_link_libraries`:

```
target_link_libraries(boost_filesystem PUBLIC
    boost::core
    boost::static_assert
    boost::iterator
    boost::detail
    boost::system
    boost::functional
    boost::assert
    boost::range
    boost::type_traits
    boost::smart_ptr
    boost::io
    boost::config
)
```

Now, some of these libraries are header-only, but when we call `target_link_libraries` it will add all the flags necessary to use those libraries. Next step is installation, using the `install` command:

```
install(DIRECTORY include/ DESTINATION include)

install(TARGETS boost_filesystem EXPORT boost_filesystem-targets
```

(continues on next page)

(continued from previous page)

```
    RUNTIME DESTINATION bin
    LIBRARY DESTINATION lib
    ARCHIVE DESTINATION lib
    INCLUDES DESTINATION include
)
```

So this will install the include directories and install the library. The `EXPORT` command will have cmake generate an export file that will create the target's usage requirements in cmake. This will enable the target to be used by downstream libraries, just like we used `` `boost::system ``. However, this will only tells cmake which targets are in the export file. To generate it we use `install(EXPORT)`:

```
install(EXPORT boost_filesystem-targets
    FILE boost_filesystem-targets.cmake
    NAMESPACE boost::
    DESTINATION lib/cmake/boost_filesystem
)
```

This sets a namespace `boost::` on the target, but our target is named `boost_filesystem`, and we want the exported target to be `boost::filesystem` not `boost::boost_filesystem`. We can do that by setting the export name:

```
set_property(TARGET boost_filesystem PROPERTY EXPORT_NAME filesystem)
```

We can also define a target alias to `boost::filesystem`, which helps integrated builds:

```
add_library(boost::filesystem ALIAS boost_filesystem)
```

So now have exported targets we want to generate a `boost_filesystem-config.cmake` file so it can be used with `find_package(boost_filesystem)`. To do this we generate a file the includes the export file, but it also calls `find_dependency` on each dependency so that the user does not have to call it:

```
file(WRITE "${PROJECT_BINARY_DIR}/boost_filesystem-config.cmake" "
include(CMakeFindDependencyMacro)
find_dependency(boost_core)
find_dependency(boost_static_assert)
find_dependency(boost_iterator)
find_dependency(boost_detail)
find_dependency(boost_system)
find_dependency(boost_functional)
find_dependency(boost_assert)
find_dependency(boost_range)
find_dependency(boost_type_traits)
find_dependency(boost_smart_ptr)
find_dependency(boost_io)
find_dependency(boost_config)
include(\"\${CMAKE_CURRENT_LIST_DIR}/boost_filesystem-targets.cmake\")
")
```

Besides the `boost_filesystem-config.cmake`, we also need a version file to check compatibility. This can be done using cmake's `write_basic_package_version_file` function:

```
write_basic_package_version_file("${PROJECT_BINARY_DIR}/boost_filesystem-config-
↪version.cmake"
    VERSION 1.64
    COMPATIBILITY AnyNewerVersion
)
```

Then finally we install these files:

```
install(FILES
    "${PROJECT_BINARY_DIR}/boost_filesystem-config.cmake"
    "${PROJECT_BINARY_DIR}/boost_filesystem-config-version.cmake"
    DESTINATION lib/cmake/boost_filesystem
)
```

Putting it all together we have a cmake file that looks like this:

```
cmake_minimum_required(VERSION 3.5)
project(boost_filesystem)
include(CMakePackageConfigHelpers)

find_package(boost_core)
find_package(boost_static_assert)
find_package(boost_iterator)
find_package(boost_detail)
find_package(boost_system)
find_package(boost_functional)
find_package(boost_assert)
find_package(boost_range)
find_package(boost_type_traits)
find_package(boost_smart_ptr)
find_package(boost_io)
find_package(boost_config)

add_library(boost_filesystem
  src/operations.cpp
  src/portability.cpp
  src/codecvt_error_category.cpp
  src/utf8_codecvt_facet.cpp
  src/windows_file_codecvt.cpp
  src/unique_path.cpp
  src/path.cpp
  src/path_traits.cpp
)
add_library(boost::filesystem ALIAS boost_filesystem)
set_property(TARGET boost_filesystem PROPERTY EXPORT_NAME filesystem)

target_include_directories(boost_filesystem PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
)
target_link_libraries(boost_filesystem PUBLIC
    boost::core
    boost::static_assert
    boost::iterator
    boost::detail
    boost::system
    boost::functional
    boost::assert
    boost::range
    boost::type_traits
    boost::smart_ptr
    boost::io
    boost::config
)
```

(continues on next page)

(continued from previous page)

```
install(DIRECTORY include/ DESTINATION include)

install(TARGETS boost_filesystem EXPORT boost_filesystem-targets
    RUNTIME DESTINATION bin
    LIBRARY DESTINATION lib
    ARCHIVE DESTINATION lib
    INCLUDES DESTINATION include
)

install(EXPORT boost_filesystem-targets
    FILE boost_filesystem-targets.cmake
    NAMESPACE boost::
    DESTINATION lib/cmake/boost_filesystem
)

file(WRITE "${PROJECT_BINARY_DIR}/boost_filesystem-config.cmake" "
include(CMakeFindDependencyMacro)
find_dependency(boost_core)
find_dependency(boost_static_assert)
find_dependency(boost_iterator)
find_dependency(boost_detail)
find_dependency(boost_system)
find_dependency(boost_functional)
find_dependency(boost_assert)
find_dependency(boost_range)
find_dependency(boost_type_traits)
find_dependency(boost_smart_ptr)
find_dependency(boost_io)
find_dependency(boost_config)
include(\"\${CMAKE_CURRENT_LIST_DIR}/boost_filesystem-targets.cmake\")
")

write_basic_package_version_file("${PROJECT_BINARY_DIR}/boost_filesystem-config-
↪version.cmake"
    VERSION 1.64
    COMPATIBILITY AnyNewerVersion
)

install(FILES
    "${PROJECT_BINARY_DIR}/boost_filesystem-config.cmake"
    "${PROJECT_BINARY_DIR}/boost_filesystem-config-version.cmake"
    DESTINATION lib/cmake/boost_filesystem
)
```

4.2 Building standalone with BCM

The boost cmake modules can help reduce the boilerplate needed in writing these libraries. To use these modules we just call `find_package(BCM)` first:

```
cmake_minimum_required(VERSION 3.5)
project(boost_filesystem)
find_package(BCM)
```

Next we can setup the version for the project using `bcm_setup_version`:

```
bcm_setup_version(VERSION 1.64)
```

Next, we add the library and link against the dependencies like always:

```
find_package(boost_core)
find_package(boost_static_assert)
find_package(boost_iterator)
find_package(boost_detail)
find_package(boost_system)
find_package(boost_functional)
find_package(boost_assert)
find_package(boost_range)
find_package(boost_type_traits)
find_package(boost_smart_ptr)
find_package(boost_io)
find_package(boost_config)

add_library(boost_filesystem
  src/operations.cpp
  src/portability.cpp
  src/codecvt_error_category.cpp
  src/utf8_codecvt_facet.cpp
  src/windows_file_codecvt.cpp
  src/unique_path.cpp
  src/path.cpp
  src/path_traits.cpp
)
add_library(boost::filesystem ALIAS boost_filesystem)
set_property(TARGET boost_filesystem PROPERTY EXPORT_NAME filesystem)

target_link_libraries(boost_filesystem PUBLIC
    boost::core
    boost::static_assert
    boost::iterator
    boost::detail
    boost::system
    boost::functional
    boost::assert
    boost::range
    boost::type_traits
    boost::smart_ptr
    boost::io
    boost::config
)
```

Then to install, and generate package configuration we just use `bcm_deploy`:

```
bcm_deploy(TARGETS boost_filesystem NAMESPACE boost::)
```

In addition to generating package configuration for cmake, this will also generate the package configuration for `pkgconfig`.

4.3 Integrated builds

As we were setting up cmake for standalone builds, we made sure we didn't do anything to prevent an integrated build, and even provided an alias target to help ease the process. Finally, to integrate the sources into the build is just a matter

of calling `add_subdirectory` on each project:

```
file(GLOB LIBS libs/*)
foreach(lib ${LIBS})
    add_subdirectory(${lib})
endforeach()
```

We could also use `add_subdirectory(${lib} EXCLUDE_FROM_ALL)` so it builds targets that are not necessary. Of course, every project is still calling `find_package` to find prebuilt binaries. Since we don't need to search for those libraries because they are integrated into the build we can call `bcm_ignore_package` to ignore those dependencies:

```
file(GLOB LIBS libs/*)

foreach(lib ${LIBS})
    bcm_ignore_package(${lib})
endforeach()

foreach(lib ${LIBS})
    add_subdirectory(${lib})
endforeach()
```

Of course, this assumes we have conveniently named each directory the same as its package name.

CHAPTER 5

Modules

5.1 BCMDeploy

5.1.1 bcm_deploy

This will install targets, as well as generate package configuration for both cmake and pkgconfig.

TARGETS `<target-name>...`

The name of the targets to deploy.

INCLUDE `<directory>...`

Include directories to be installed. It also makes the include directory available for targets to be installed.

NAMESPACE `<namespace>`

This is the namespace to add to the targets that are exported.

COMPATIBILITY `<compatibility>`

This uses the version compatibility specified by cmake version config.

5.2 BCMExport

5.2.1 bcm_auto_export

This generates a simple cmake config file that includes the exported targets.

EXPORT

This specifies an export file. By default, the export file will be named `${PROJECT_NAME}-targets`.

NAMESPACE `<namespace>`

This is the namespace to add to the targets that are exported.

NAME <name>

This is the name to use for the package config file. By default, this uses the project name, but this parameter can override it.

TARGETS <target>...

These include the targets to be exported.

5.3 BCMIgnorePackage

5.3.1 bcm_ignore_package

This will ignore a package so that subsequent calls to *find_package* will be treated as found. This is useful in the superproject of integrated builds because it will ingore the `find_package` calls to a dependency becaue the targets are already provided by `add_subdirectory`.

NAME

The name of the package to ignore.

5.4 BCMInstallTargets

5.4.1 bcm_install_targets

This installs the targets specified. The directories will be installed according to GNUInstallDirs. It will also install a corresponding cmake package config(which can be found with `find_package`) to link against the library targets.

TARGETS <target-name>...

The name of the targets to install.

INCLUDE <directory>...

Include directories to be installed. It also makes the include directory available for targets to be installed.

EXPORT

This specifies an export file. By default, the export file will be named `${PROJECT_NAME}-targets`.

5.5 BCMPkgConfig

5.5.1 bcm_generate_pkgconfig_file

This will generate a simple pkgconfig file.

NAME <name>

This is the name of the pkgconfig module.

LIB_DIR <directory>

This is the directory where the library is linked to. This defaults to `${CMAKE_INSTALL_LIBDIR}`.

INCLUDE_DIR <directory>

This is the include directory where the header file are installed. This defaults to `${CMAKE_INSTALL_INCLUDEDIR}`.

DESCRIPTION `<text>`

A description about the library.

TARGETS `<targets>...`

The library targets to link.

CFLAGS `<flags>...`

Additionaly, compiler flags.

LIBS `<library flags>...`

Additional libraries to be linked.

REQUIRES `<packages>...`

List of other pkgconfig packages that this module depends on.

5.5.2 bcm_auto_pkgconfig

This will auto generate pkgconfig from a given target. All the compiler and linker flags come from the target.

NAME `<name>`

This is the name of the pkgconfig module. By default, this will use the project name.

TARGET `<TARGET>`

This is the target which will be used to set the various pkgconfig fields.

5.6 BCMProperties

This module defines several properties that can be used to control language features in C++.

5.6.1 CXX_EXCEPTIONS

This property can be used to enable or disable C++ exceptions. This can be applied at global, directory or target scope. At global scope this defaults to On.

5.6.2 CXX_RTTI

This property can be used to enable or disable C++ runtime type information. This can be applied at global, directory or target scope. At global scope this defaults to On.

5.6.3 CXX_STATIC_RUNTIME

This property can be used to enable or disable linking against the static C++ runtime. This can be applied at global, directory or target scope. At global scope this defaults to Off.

5.6.4 CXX_WARNINGS

The `CXX_WARNINGS` property controls the warning level of compilers. It has the following values:

- `off` - disables all warnings.
- `on` - enables default warning level for the tool.
- `all` - enables all warnings.

Default value is `on`.

5.6.5 CXX_WARNINGS_AS_ERRORS

The `CXX_WARNINGS_AS_ERRORS` property makes it possible to treat warnings as errors and abort compilation on a warning. The value `on` enables this behaviour. The default value is `off`.

5.6.6 INTERFACE_DESCRIPTION

Description of the target.

5.6.7 INTERFACE_URL

An URL where people can get more information about and download the package.

5.6.8 INTERFACE_PKG_CONFIG_REQUIRES

A list of packages required by this package for pkgconfig. The versions of these packages may be specified using the comparison operators =, <, >, <= or >=.

5.6.9 INTERFACE_PKG_CONFIG_NAME

The name of the pkgconfig package for this target.

5.7 BCMSetupVersion

5.7.1 bcm_setup_version

This sets up the project version by setting these version variables:

```
PROJECT_VERSION, ${PROJECT_NAME}_VERSION
PROJECT_VERSION_MAJOR, ${PROJECT_NAME}_VERSION_MAJOR
PROJECT_VERSION_MINOR, ${PROJECT_NAME}_VERSION_MINOR
PROJECT_VERSION_PATCH, ${PROJECT_NAME}_VERSION_PATCH
```

It also generates a cmake package config version file as well.

VERSION `<major>.<minor>.<patch>`

This is the version to be set.

GENERATE_HEADER `<header-name>`

This is a header which will be generated with defines for the version number.

PREFIX <identifier>

By default, the upper case of the project name is used as a prefix for the version macros that are defined in the generated header: ${PREFIX}_VERSION_MAJOR, ${PREFIX}_VERSION_MINOR, ${PREFIX}_VERSION_PATCH, and ${PREFIX}_VERSION. The PREFIX option allows overriding the prefix name used for the macros.

PARSE_HEADER <header-name>

Rather than set a version and generate a header, this will parse a header with macros that define the version, and then use those values to set the version for the project.

COMPATIBILITY <compatibility>

This uses the version compatibility specified by cmake version config.

NAME <name>

This is the name to use for the package config version file. By default, this uses the project name, but this parameter can override it.

5.8 BCMTest

5.8.1 bcm_mark_as_test

This marks the target as a test, so it will be built with the tests target. If BUILD_TESTING is set to off then the target will not be built as part of the all target.

5.8.2 bcm_test_link_libraries

This sets libraries that the tests will link against by default.

5.8.3 bcm_test

This setups a test. By default, a test will be built and executed.

SOURCES <source-files>...

Source files to be compiled for the test.

CONTENT <content>

This a string that will be used to create a test to be compiled and/or ran.

NAME <name>

Name of the test.

ARGS <args>

This sets additional arguments to be passed to the test executable when it will be ran.

COMPILE_ONLY

This just compiles the test instead of running it. As such, a main function is not required.

WILL_FAIL

Specifies that the test will fail.

NO_TEST_LIBS

This won't link in the libraries specified by `bcm_test_link_libraries`

5.8.4 bcm_test_header

This creates a test to test the include of a header.

NAME `<name>`

Name of the test.

HEADER `<header-file>`

The header to include.

STATIC

Rather than just test the include, using `STATIC` option will test the include across translation units. This helps check for incorrect include guards and duplicate symbols.

NO_TEST_LIBS

This won't link in the libraries specified by `bcm_test_link_libraries`

5.8.5 bcm_add_test_subdirectory

This calls `add_subdirectory` if the `ENABLE_TESTS` property is true. The default value for the property is set by `CMAKE_ENABLE_TESTS` variable.

Index