BLT Tutorial Guide

BLT Team

Mar 20, 2020

Contents

Build, Link, and Test

BLT is a composition of CMake macros and several widely used open source tools assembled to simplify HPC software development.

BLT was released by Lawrence Livermore National Laboratory (LLNL) under a BSD-style open source license. It is developed on github under LLNL's github organization: https://github.com/llnl/blt

---

Note: BLT officially supports CMake 3.8 and above. However we only print a warning if you are below this version. Some features in earlier versions may or may not work. Use at your own risk.

CHAPTER 1

BLT at a Glance

- Simplifies setting up a CMake-based build system
  - CMake macros for:
    - Creating libraries and executables
    - Managing compiler flags
    - Managing external dependencies
  - Multi-platform support (HPC Platforms, OSX, Windows)
- Batteries included
  - Built-in support for HPC Basics: MPI, OpenMP, and CUDA
  - Built-in support for unit testing in C/C++ and Fortran
- Streamlines development processes
  - Support for documentation generation
  - Support for code health tools:
    - Runtime and static analysis, benchmarking

CHAPTER 2

Developers

- Chris White (white238@llnl.gov)
- Cyrus Harrison (harrison37@llnl.gov)
- George Zagaris (zagaris2@llnl.gov)
- Kenneth Weiss (kweiss@llnl.gov)
- Lee Taylor (taylor16@llnl.gov)
- Aaron Black (black27@llnl.gov)
- David A. Beckingsale (beckingsale1@llnl.gov)
- Richard Hornung (hornung1@llnl.gov)
- Randolph Settgast (settgast1@llnl.gov)
- Peter Robinson (robinson96@llnl.gov)

CHAPTER 3

Documentation

3.1 User Tutorial

This tutorial provides instructions for:

- Adding BLT to a CMake project
- Setting up *host-config* files to handle multiple platform configurations
- Building, linking, and installing libraries and executables
- Setting up unit tests with GTest
- Using external project dependencies
- Creating documentation with Sphinx and Doxygen

The tutorial provides several examples that calculate the value of π by approximating the integral $f(x) = \int_0^1 4/(1+x^2)$ using numerical integration. The code is adapted from: https://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples-usingmpi/simplempi/cpi_c.html.

The tutorial requires a C++ compiler and CMake, we recommend using CMake 3.8.0 or newer. Parts of the tutorial also require MPI, CUDA, Sphinx and Doxygen.

3.1.1 Setup BLT in your CMake Project

BLT is easy to include in your CMake project whether it is an existing project or you are starting from scratch. You simply pull it into your project using a CMake `include()` command.

```
include(path/to/blt/SetupBLT.cmake)
```

You can include the BLT source in your repository or pass the location of BLT at CMake configure time through the optional `BLT_SOURCE_DIR` CMake variable.

There are two standard choices for including the BLT source in your repository:

1. Add BLT as a git submodule

2. Copy BLT into a subdirectory in your repository

BLT as a Git Submodule

This example adds BLT as a submodule, commits, and pushes the changes to your repository.

```
cd <your repository>
git submodule add https://github.com/LLNL/blt.git blt
git commit -m "Adding BLT"
git push
```

Copy BLT into your repository

This example will clone BLT into your repository and remove the unneeded git files from the clone. It then commits and pushes the changes to your repository.

```
cd <your repository>
git clone https://github.com/LLNL/blt.git
rm -rf blt/.git
git commit -m "Adding BLT"
git push
```

Include BLT in your project

In most projects, including BLT is as simple as including the following CMake line in your base `CMakeLists.txt` after your `project()` call.

```
include(blt/SetupBLT.cmake)
```

This enables all of BLT's features in your project.

However if your project is likely to be used by other projects. The following is recommended:

```
if (DEFINED BLT_SOURCE_DIR)
    # Support having a shared BLT outside of the repository if given a BLT_SOURCE_DIR
    if (NOT EXISTS ${BLT_SOURCE_DIR}/SetupBLT.cmake)
        message(FATAL_ERROR "Given BLT_SOURCE_DIR does not contain SetupBLT.cmake")
    endif()
else()
    # Use internal BLT if no BLT_SOURCE_DIR is given
    set(BLT_SOURCE_DIR "${PROJECT_SOURCE_DIR}/cmake/blt" CACHE PATH "")
    if (NOT EXISTS ${BLT_SOURCE_DIR}/SetupBLT.cmake)
        message(FATAL_ERROR
            "The BLT git submodule is not present. "
            "Either run the following two commands in your git repository: \n"
            "    git submodule init\n"
            "    git submodule update\n"
            "Or add -DBLT_SOURCE_DIR=/path/to/blt to your CMake command." )
    endif()
endif()

# Default to C++11 if not set so GTest/GMock can build
if (NOT BLT_CXX_STD)
    set(BLT_CXX_STD "c++11" CACHE STRING "")
```

(continues on next page)

(continued from previous page)

```
endif()

include(${BLT_SOURCE_DIR}/SetupBLT.cmake)
```

This is a robust way of setting up BLT and supports an optional external BLT source directory. This allows the use of a common BLT across large projects. There are some helpful error messages if the BLT submodule is missing as well as the commands to solve it.

Note: Throughout this tutorial, we pass the path to BLT using `BLT_SOURCE_DIR` since our tutorial is part of the blt repository and we want this project to be automatically tested using a single clone of our repository.

Running CMake

To configure a project with CMake, first create a build directory and `cd` into it. Then run cmake with the path to your project.

```
cd <your project>
mkdir build
cd build
cmake ..
```

If you are using BLT outside of your project pass the location of BLT as follows:

```
cd <your project>
mkdir build
cd build
cmake -DBLT_SOURCE_DIR="path/to/blt" ..
```

Example: blank_project

The `blank_project` example is provided to show you some of BLT's built-in features. It demonstrates the bare minimum required for testing purposes.

Here is the entire CMakeLists.txt file for `blank_project`:

```
#------------------------------------------------------------------------------
# BLT Tutorial Example: Blank Project.
#------------------------------------------------------------------------------

cmake_minimum_required(VERSION 3.8)
project( blank )

# Note: This is specific to running our tests and shouldn't be exported to␣
↪documentation
if(NOT BLT_SOURCE_DIR)
    set(BLT_SOURCE_DIR "${CMAKE_CURRENT_SOURCE_DIR}/../../..")
endif()

#------------------------------------------------------------------------------
# Setup BLT
#------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
# _blt_tutorial_include_blt_start
if (DEFINED BLT_SOURCE_DIR)
    # Support having a shared BLT outside of the repository if given a BLT_SOURCE_DIR
    if (NOT EXISTS ${BLT_SOURCE_DIR}/SetupBLT.cmake)
        message(FATAL_ERROR "Given BLT_SOURCE_DIR does not contain SetupBLT.cmake")
    endif()
else()
    # Use internal BLT if no BLT_SOURCE_DIR is given
    set(BLT_SOURCE_DIR "${PROJECT_SOURCE_DIR}/cmake/blt" CACHE PATH "")
    if (NOT EXISTS ${BLT_SOURCE_DIR}/SetupBLT.cmake)
        message(FATAL_ERROR
            "The BLT git submodule is not present. "
            "Either run the following two commands in your git repository: \n"
            "    git submodule init\n"
            "    git submodule update\n"
            "Or add -DBLT_SOURCE_DIR=/path/to/blt to your CMake command." )
    endif()
endif()

# Default to C++11 if not set so GTest/GMock can build
if (NOT BLT_CXX_STD)
    set(BLT_CXX_STD "c++11" CACHE STRING "")
endif()

include(${BLT_SOURCE_DIR}/SetupBLT.cmake)
# _blt_tutorial_include_blt_end
```

BLT also enforces some best practices for building, such as not allowing in-source builds. This means that BLT prevents you from generating a project configuration directly in your project.

For example if you run the following commands:

```
cd <BLT repository>/docs/tutorial/blank_project
cmake -DBLT_SOURCE_DIR=../..
```

you will get the following error:

```
CMake Error at blt/SetupBLT.cmake:59 (message):
  In-source builds are not supported.  Please remove CMakeCache.txt from the
  'src' dir and configure an out-of-source build in another directory.
Call Stack (most recent call first):
  CMakeLists.txt:55 (include)


-- Configuring incomplete, errors occurred!
```

To correctly run cmake, create a build directory and run cmake from there:

```
cd <BLT repository>/docs/blank_project
mkdir build
cd build
cmake -DBLT_SOURCE_DIR=../../.. ..
```

This will generate a configured `Makefile` in your build directory to build `blank_project`. The generated makefile includes gtest and several built-in BLT *smoke* tests, depending on the features that you have enabled in your build.

To build the project, use the following command:

```
make
```

As with any other `make`-based project, you can utilize parallel job tasks to speed up the build with the following command:

```
make -j8
```

Next, run all tests in this project with the following command:

```
make test
```

If everything went correctly, you should have the following output:

```
Running tests...
Test project blt/docs/tutorial/blank_project/build
    Start 1: blt_gtest_smoke
1/1 Test #1: blt_gtest_smoke ..................   Passed    0.01 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.10 sec
```

Note that the default options for `blank_project` only include a single test `blt_gtest_smoke`. As we will see later on, BLT includes additional smoke tests that are activated when BLT is configured with other options enabled, like Fortran, MPI, OpenMP, and Cuda.

Host-configs

To capture (and revision control) build options, third party library paths, etc., we recommend using CMake's initial-cache file mechanism. This feature allows you to pass a file to CMake that provides variables to bootstrap the configuration process.

You can pass initial-cache files to cmake via the `-C` command line option.

```
cmake -C config_file.cmake
```

We call these initial-cache files `host-config` files since we typically create a file for each platform or for specific hosts, if necessary.

These files use standard CMake commands. CMake `set()` commands need to specify `CACHE` as follows:

```
set(CMAKE_VARIABLE_NAME {VALUE} CACHE PATH "")
```

Here is a snippet from a host-config file that specifies compiler details for using gcc 4.9.3 on LLNL's surface cluster.

```
set(CMAKE_C_COMPILER   "/usr/apps/gnu/4.9.3/bin/gcc" CACHE PATH "")
set(CMAKE_CXX_COMPILER "/usr/apps/gnu/4.9.3/bin/g++" CACHE PATH "")

# Fortran support
set(ENABLE_FORTRAN ON CACHE BOOL "")
set(CMAKE_Fortran_COMPILER "/usr/apps/gnu/4.9.3/bin/gfortran" CACHE PATH "")
```

3.1.2 Creating Libraries and Executables

In the previous section, we learned the basics about how to create a CMake project with BLT, how to configure the project and how to build and test BLT's built-in third party libraries.

We now move on to creating libraries and executables using two of BLT's core macros: `blt_add_library()` and `blt_add_executable()`.

We begin with a simple executable that calculates π by numerical integration (`example_1`). We will then extract that code into a library, which we link into a new executable (`example_2`).

Example 1: Basic executable

This example is as basic as it gets. After setting up a BLT CMake project, like `blank_project` in the previous section, we can start using BLT's macros. Creating an executable is as simple as calling the following macro:

```
blt_add_executable( NAME    example_1
                    SOURCES example_1.cpp )
```

This tells CMake to create an executable named `example_1` with one source file (`example_1.cpp`).

You can create this project yourself or you can run the already provided `tutorial/calc_pi` project. For ease of use, we have combined many examples into this one CMake project. After running the following commands, you will create the executable `<build dir>/bin/example_1`:

```
cd <BLT repository/docs/tutorial/calc_pi
mkdir build
cd build
cmake -DBLT_SOURCE_DIR=../../.. ..
make
```

---

blt_add_executable

This is one of the core macros that enables BLT to simplify our CMake-based project. It unifies many CMake calls into one easy to use macro. `blt_add_executable()` creates a CMake executable target with the given sources, sets the output directory to `<build dir>/bin` (unless overridden with the macro parameter `OUTPUT_DIR`) and handles internal and external dependencies in a greatly simplified manner. There will be more on that in the following section.

Example 2: One library, one executable

This example is a bit more exciting. This time we are creating a library that calculates the value of pi and then linking that library into an executable.

First, we create the library with the following BLT code:

```
blt_add_library( NAME    calc_pi
                 HEADERS calc_pi.hpp calc_pi_exports.h
                 SOURCES calc_pi.cpp )
```

Just like before, this creates a CMake library target that will get built to `<build dir>/lib/libcalc_pi.a`.

Next, we create an executable named `example_2` and link in the previously created library target:

```
blt_add_executable( NAME       example_2
                    SOURCES    example_2.cpp
                    DEPENDS_ON calc_pi)
```

The `DEPENDS_ON` parameter properly links the previously defined library into this executable without any more work or CMake function calls.

blt_add_library

This is another core BLT macro. It creates a CMake library target and associates the given sources and headers along with handling dependencies the same way as `blt_add_executable` does. It also provides a few commonly used build options, such as overriding the output name of the library and the output directory. It defaults to building a static library unless you override it with `SHARED` or with the global CMake option `BUILD_SHARED_LIBS`.

Object Libraries

BLT has simplified the use of CMake object libraries through the `blt_add_library` macro. Object libraries are a collection of object files that are not linked or archived into a library. They are used in other libraries or executables through the `DEPENDS_ON` macro argument. This is generally useful for combining smaller libraries into a larger library without the linker removing unused symbols in the larger library.

```
blt_add_library(NAME    myObjectLibrary
                SOURCES source1.cpp
                HEADERS header1.cpp
                OBJECT  TRUE)

blt_add_exectuble(NAME       helloWorld
                  SOURCES    main.cpp
                  DEPENDS_ON myObjectLibrary)
```

Note: Due to record keeping on BLT's part to make object libraries as easy to use as possible, you need to define object libraries before you use them if you need their inheritable information to be correct.

3.1.3 Portable compiler flags

To simplify the development of code that is portable across different architectures and compilers, BLT provides the `blt_append_custom_compiler_flag()` macro, which allows users to easily place a compiler dependent flag into a CMake variable.

blt_append_custom_compiler_flag

To use this macro, supply a cmake variable in which to append a flag (`FLAGS_VAR`), and the appropriate flag for each of our supported compilers.

This macro currently supports the following compilers:

- GNU
- CLANG
- XL (IBM compiler)
- INTEL (Intel compiler)
- MSVC (Microsoft Visual Studio)
- MSVC_INTEL (Intel toolchain in Microsoft Visual Studio)
- PGI
- HCC (AMD GPU)

Here is an example for setting the appropriate flag to treat warnings as errors:

```
blt_append_custom_compiler_flag(
  FLAGS_VAR BLT_WARNINGS_AS_ERRORS_FLAG
  DEFAULT  "-Werror"
  MSVC     "/WX"
  XL       "qhalt=w"
  )
```

Since values for `GNU`, `CLANG` and `INTEL` are not supplied, they will get the default value (`-Werror`) which is supplied by the macro's `DEFAULT` argument.

BLT also provides a simple macro to add compiler flags to a target. You can append the above compiler flag to an already defined executable, such as `example_1` with the following line:

```
blt_add_target_compile_flags(TO example_1
                             FLAGS BLT_WARNINGS_AS_ERRORS_FLAG )
```

Here is another example to disable warnings about unknown OpenMP pragmas in the code:

```
# Flag for disabling warnings about omp pragmas in the code
blt_append_custom_compiler_flag(
    FLAGS_VAR DISABLE_OMP_PRAGMA_WARNINGS_FLAG
    DEFAULT "-Wno-unknown-pragmas"
    XL      "-qignprag=omp"
    INTEL   "-diag-disable 3180"
    MSVC    "/wd4068"
    )
```

Note that GNU does not have a way to only disable warnings about openmp pragmas, so one must disable warnings about all unknown pragmas on this compiler.

3.1.4 Unit Testing

BLT has a built-in copy of the Google Test framework (gtest) for C and C++ unit tests and the Fortran Unit Test Framework (FRUIT) for Fortran unit tests.

Each Google Test or FRUIT file may contain multiple tests and is compiled into its own executable that can be run directly or as a `make` target.

In this section, we give a brief overview of GTest and discuss how to add unit tests using the `blt_add_test()` macro.

Configuring tests within BLT

Unit testing in BLT is controlled by the `ENABLE_TESTS` cmake option and is on by default.

For additional configuration granularity, BLT provides configuration options for the individual built-in unit testing libraries. The following additional options are available when `ENABLE_TESTS` is on:

`ENABLE_GTEST` Option to enable gtest (default: `ON`).

`ENABLE_GMOCK` Option to control gmock (default: `OFF`). Since gmock requires gtest, gtest is also enabled whenever `ENABLE_GMOCK` is true, regardless of the value of `ENABLE_GTEST`.

`ENABLE_FRUIT` Option to control FRUIT (Default `ON`). It is only active when `ENABLE_FORTRAN` is enabled.

Google Test (C++/C Tests)

The contents of a typical Google Test file look like this:

```
#include "gtest/gtest.h"

#include ...    // include headers needed to compile tests in file

// ...

TEST(<test_case_name>, <test_name_1>)
{
   // Test 1 code here...
   // ASSERT_EQ(...);
}

TEST(<test_case_name>, <test_name_2>)
{
   // Test 2 code here...
   // EXPECT_TRUE(...);
}

// Etc.
```

Each unit test is defined by the Google Test `TEST()` macro which accepts a *test case name* identifier, such as the name of the C++ class being tested, and a *test name*, which indicates the functionality being verified by the test. Within a test, failure of logical assertions (macros prefixed by `ASSERT_`) will cause the test to fail immediately, while failures of expected values (macros prefixed by `EXPECT_`) will cause the test to fail, but will continue running code within the test.

Note that the Google Test framework will generate a `main()` routine for each test file if it is not explicitly provided. However, sometimes it is necessary to provide a `main()` routine that contains operation to run before or after the unit tests in a file; e.g., initialization code or pre-/post-processing operations. A `main()` routine provided in a test file should be placed at the end of the file in which it resides.

Note that Google Test is initialized before `MPI_Init()` is called.

Other Google Test features, such as *fixtures* and *mock* objects (gmock) may be used as well.

See the Google Test Primer for a discussion of Google Test concepts, how to use them, and a listing of available assertion macros, etc.

FRUIT (Fortran Tests)

Fortran unit tests using the FRUIT framework are similar in structure to the Google Test tests for C and C++ described above.

The contents of a typical FRUIT test file look like this:

```
module <test_case_name>
  use iso_c_binding
  use fruit
  use <your_code_module_name>
  implicit none

contains

subroutine test_name_1
```

(continues on next page)

(continued from previous page)

```
!  Test 1 code here...
!  call assert_equals(...)
end subroutine test_name_1

subroutine test_name_2
!  Test 2 code here...
!  call assert_true(...)
end subroutine test_name_2

! Etc.
```

The tests in a FRUIT test file are placed in a Fortran *module* named for the *test case name*, such as the name of the C++ class whose Fortran interface is being tested. Each unit test is in its own Fortran subroutine named for the *test name*, which indicates the functionality being verified by the unit test. Within each unit test, logical assertions are defined using FRUIT methods. Failure of expected values will cause the test to fail, but other tests will continue to run.

Note that each FRUIT test file defines an executable Fortran program. The program is defined at the end of the test file and is organized as follows:

```
program fortran_test
  use fruit
  use <your_component_unit_name>
  implicit none
  logical ok

  ! initialize fruit
  call init_fruit

  ! run tests
  call test_name_1
  call test_name_2

  ! compile summary and finalize fruit
  call fruit_summary
  call fruit_finalize

  call is_all_successful(ok)
  if (.not. ok) then
    call exit(1)
  endif
end program fortran_test
```

Please refer to the FRUIT documentation for more information.

Adding a BLT unit test

After writing a unit test, we add it to the project's build system by first generating an executable for the test using the `blt_add_executable()` macro. We then register the test using the `blt_add_test()` macro.

---

blt_add_test

This macro generates a named unit test from an existing executable and allows users to pass in command line arguments.

Returning to our running example (see *Creating Libraries and Executables*), let's add a simple test for the `calc_pi` library, which has a single function with signature:

```
double calc_pi(int num_intervals);
```

We add a simple unit test that invokes `calc_pi()` and compares the result to π, within a given tolerance (`1e-6`). Here is the test code:

```
#include <gtest/gtest.h>

#include "calc_pi.hpp"

TEST(calc_pi, serial_example)
{
    double PI_REF = 3.14159265358979323846264 3;
    ASSERT_NEAR(calc_pi(1000),PI_REF,1e-6);
}
```

To add this test to the build system, we first generate a test executable:

```
blt_add_executable( NAME       test_1
                    SOURCES    test_1.cpp
                    DEPENDS_ON calc_pi gtest)
```

Note that this test executable depends on two targets: `calc_pi` and `gtest`.

We then register this executable as a test:

```
blt_add_test( NAME    test_1
              COMMAND test_1)
```

Running tests and examples

To run the tests, type the following command in the build directory:

```
$ make test
```

This will run all tests through cmake's `ctest` tool and report a summary of passes and failures. Detailed output on individual tests is suppressed.

If a test fails, you can invoke its executable directly to see the detailed output of which checks passed or failed. This is especially useful when you are modifying or adding code and need to understand how unit test details are working, for example.

Note: You can pass arguments to ctest via the `TEST_ARGS` parameter, e.g. `make test TEST_ARGS="..."` Useful arguments include:

-R	Regular expression filtering of tests. E.g. `-R foo` only runs tests whose names contain `foo`
-j	Run tests in parallel, E.g. `-j 16` will run tests using 16 processors
-VV	(Very verbose) Dump test output to stdout

3.1.5 External Dependencies

One key goal for BLT is to simplify the use of external dependencies when building your libraries and executables.

To accomplish this BLT provides a `DEPENDS_ON` option for the `blt_add_library()` and `blt_add_executable()` macros that supports both CMake targets and external dependencies registered using the `blt_register_library()` macro.

The `blt_register_library()` macro allows you to reuse all information needed for an external dependency under a single name. This includes any include directories, libraries, compile flags, link flags, defines, etc. You can also hide any warnings created by their headers by setting the `TREAT_INCLUDES_AS_SYSTEM` argument.

For example, to find and register the external dependency *axom* as a BLT registered library, you can simply use:

```
# FindAxom.cmake takes in AXOM_DIR, which is a installed Axom build and
# sets variables AXOM_INCLUDES, AXOM_LIBRARIES
include(FindAxom.cmake)
blt_register_library(NAME      axom
                     TREAT_INCLUDES_AS_SYSTEM ON
                     DEFINES   HAVE_AXOM=1
                     INCLUDES  ${AXOM_INCLUDES}
                     LIBRARIES ${AXOM_LIBRARIES})
```

Then *axom* is available to be used in the DEPENDS_ON list in the following `blt_add_executable()` or `blt_add_library()` calls.

This is especially helpful for external libraries that are not built with CMake and don't provide CMake-friendly imported targets. Our ultimate goal is to use `blt_register_library()` to import all external dependencies as first-class imported CMake targets to take full advanced of CMake's dependency lattice.

MPI, CUDA, and OpenMP are all registered via `blt_register_library()`. You can see how in `blt/thirdparty_builtin/CMakelists.txt`.

BLT also supports using `blt_register_library()` to provide additional options for existing CMake targets. The implementation doesn't modify the properties of the existing targets, it just exposes these options via BLT's support for `DEPENDS_ON`.

---

blt_register_library

A macro to register external libraries and dependencies with BLT. The named target can be added to the `DEPENDS_ON` argument of other BLT macros, like `blt_add_library()` and `blt_add_executable()`.

You have already seen one use of `DEPENDS_ON` for a BLT registered dependency in test_1: `gtest`

```
blt_add_executable( NAME       test_1
                    SOURCES    test_1.cpp
                    DEPENDS_ON calc_pi gtest)
```

`gtest` is the name for the Google Test dependency in BLT registered via `blt_register_library()`. Even though Google Test is built-in and uses CMake, `blt_register_library()` allows us to easily set defines needed by all dependent targets.

MPI Example

Our next example, `test_2`, builds and tests the `calc_pi_mpi` library, which uses MPI to parallelize the calculation over the integration intervals.

To enable MPI, we set `ENABLE_MPI`, `MPI_C_COMPILER`, and `MPI_CXX_COMPILER` in our host config file. Here is a snippet with these settings for LLNL's Surface Cluster:

```
set(ENABLE_MPI ON CACHE BOOL "")

set(MPI_C_COMPILER "/usr/local/tools/mvapich2-gnu-2.0/bin/mpicc" CACHE PATH "")

set(MPI_CXX_COMPILER "/usr/local/tools/mvapich2-gnu-2.0/bin/mpicc" CACHE PATH "")

set(MPI_Fortran_COMPILER "/usr/local/tools/mvapich2-gnu-2.0/bin/mpif90" CACHE PATH "")
```

Here, you can see how `calc_pi_mpi` and `test_2` use `DEPENDS_ON`:

```
    blt_add_library( NAME       calc_pi_mpi
                     HEADERS    calc_pi_mpi.hpp calc_pi_mpi_exports.h
                     SOURCES    calc_pi_mpi.cpp
                     DEPENDS_ON mpi)

    if(WIN32 AND BUILD_SHARED_LIBS)
        target_compile_definitions(calc_pi_mpi PUBLIC WIN32_SHARED_LIBS)
    endif()

    blt_add_executable( NAME       test_2
                        SOURCES    test_2.cpp
                        DEPENDS_ON calc_pi calc_pi_mpi gtest)
```

For MPI unit tests, you also need to specify the number of MPI Tasks to launch. We use the `NUM_MPI_TASKS` argument to `blt_add_test()` macro.

```
    blt_add_test( NAME          test_2
                  COMMAND       test_2
                  NUM_MPI_TASKS 2) # number of mpi tasks to use
```

As mentioned in *Unit Testing*, google test provides a default `main()` driver that will execute all unit tests defined in the source. To test MPI code, we need to create a main that initializes and finalizes MPI in addition to Google Test. `test_2.cpp` provides an example driver for MPI with Google Test.

```
// main driver that allows using mpi w/ google test
int main(int argc, char * argv[])
{
    int result = 0;

    ::testing::InitGoogleTest(&argc, argv);

    MPI_Init(&argc, &argv);

    result = RUN_ALL_TESTS();

    MPI_Finalize();

    return result;
}
```

Note: While we have tried to ensure that BLT chooses the correct setup information for MPI, there are several niche cases where the default behavior is insufficient. We have provided several available override variables:

- `BLT_MPI_COMPILE_FLAGS`

- BLT_MPI_INCLUDES
- BLT_MPI_LIBRARIES
- BLT_MPI_LINK_FLAGS

BLT also has the variable ENABLE_FIND_MPI which turns off all CMake's FindMPI logic and then uses the MPI wrapper directly when you provide them as the default compilers.

CUDA Example

Finally, test_3 builds and tests the calc_pi_cuda library, which uses CUDA to parallelize the calculation over the integration intervals.

To enable CUDA, we set ENABLE_CUDA, CMAKE_CUDA_COMPILER, and CUDA_TOOLKIT_ROOT_DIR in our host config file. Also before enabling the CUDA language in CMake, you need to set CMAKE_CUDA_HOST_COMPILER in CMake 3.9+ or CUDA_HOST_COMPILER in previous versions. If you do not call enable_language(CUDA), BLT will set the appropriate host compiler variable for you and enable the CUDA language.

Here is a snippet with these settings for LLNL's Surface Cluster:

```
set(ENABLE_CUDA ON CACHE BOOL "")

set(CUDA_TOOLKIT_ROOT_DIR "/opt/cudatoolkit-8.0" CACHE PATH "")
set(CMAKE_CUDA_COMPILER "/opt/cudatoolkit-8.0/bin/nvcc" CACHE PATH "")
set(CMAKE_CUDA_HOST_COMPILER "${CMAKE_CXX_COMPILER}" CACHE PATH "")
set(CUDA_SEPARABLE_COMPILATION ON CACHE BOOL "")
```

Here, you can see how calc_pi_cuda and test_3 use DEPENDS_ON:

```
    # avoid warnings about sm_20 deprecated
    set(CUDA_NVCC_FLAGS ${CUDA_NVCC_FLAGS};-arch=sm_30)

    blt_add_library( NAME       calc_pi_cuda
                     HEADERS    calc_pi_cuda.hpp calc_pi_cuda_exports.h
                     SOURCES    calc_pi_cuda.cpp
                     DEPENDS_ON cuda)

    if(WIN32 AND BUILD_SHARED_LIBS)
        target_compile_definitions(calc_pi_cuda PUBLIC WIN32_SHARED_LIBS)
    endif()



    blt_add_executable( NAME       test_3
                        SOURCES    test_3.cpp
                        DEPENDS_ON calc_pi calc_pi_cuda gtest cuda_runtime)

    blt_add_test( NAME    test_3
                  COMMAND test_3)
```

The cuda dependency for calc_pi_cuda is a little special, along with adding the normal CUDA library and headers to your library or executable, it also tells BLT that this target's C/CXX/CUDA source files need to be compiled via nvcc or cuda-clang. If this is not a requirement, you can use the dependency cuda_runtime which also adds the CUDA runtime library and headers but will not compile each source file with nvcc.

Some other useful CUDA flags are:

```
# Enable separable compilation of all CUDA files for given target or all following␣
↪targets
set(CUDA_SEPARABLE_COMPILIATION ON CACHE BOOL "")
set(CUDA_ARCH "sm_60" CACHE STRING "")
set(CMAKE_CUDA_FLAGS "-restrict -arch ${CUDA_ARCH} -std=c++11" CACHE STRING "")
set(CMAKE_CUDA_LINK_FLAGS "-Xlinker -rpath -Xlinker /path/to/mpi" CACHE STRING "")
# Needed when you have CUDA decorations exposed in libraries
set(CUDA_LINK_WITH_NVCC ON CACHE BOOL "")
```

OpenMP

To enable OpenMP, set `ENABLE_OPENMP` in your host-config file or before loading `SetupBLT.cmake`. Once OpenMP is enabled, simply add `openmp` to your library executable's `DEPENDS_ON` list.

Here is an example of how to add an OpenMP enabled executable:

```
blt_add_executable(NAME blt_openmp_smoke
                   SOURCES blt_openmp_smoke.cpp
                   OUTPUT_DIR ${TEST_OUTPUT_DIRECTORY}
                   DEPENDS_ON openmp
                   FOLDER blt/tests )
```

Note: While we have tried to ensure that BLT chooses the correct compile and link flags for OpenMP, there are several niche cases where the default options are insufficient. For example, linking with NVCC requires to link in the OpenMP libraries directly instead of relying on the compile and link flags returned by CMake's FindOpenMP package. An example of this is in `host-configs/llnl/blueos_3_ppc64le_ib_p9/clang@upstream_link_with_nvcc.cmake`. We provide two variables to override BLT's OpenMP flag logic:

- `BLT_OPENMP_COMPILE_FLAGS`
- `BLT_OPENMP_LINK_FLAGS`

Here is an example of how to add an OpenMP enabled test that sets the amount of threads used:

```
blt_add_test(NAME            blt_openmp_smoke
             COMMAND         blt_openmp_smoke
             NUM_OMP_THREADS 4)
```

Example Host-configs

Here are the full example host-config files that use gcc 4.9.3 for LLNL's Surface, Ray and Quartz Clusters.

`llnl-surface-chaos_5_x86_64_ib-gcc@4.9.3.cmake`

`llnl/blueos_3_ppc64le_ib_p9/clang@upstream_nvcc_xlf`

`llnl/toss_3_x86_64_ib/gcc@4.9.3.cmake`

Note: Quartz does not have GPUs, so CUDA is not enabled in the Quartz host-config.

Here is a full example host-config file for an OSX laptop, using a set of dependencies built with spack.

`darwin/elcapitan-x86_64/naples-clang@7.3.0.cmake`

Building and testing on Surface

Here is how you can use the host-config file to configure a build of the `calc_pi` project with MPI and CUDA enabled on Surface:

```
# load new cmake b/c default on surface is too old
ml cmake/3.9.2
# create build dir
mkdir build
cd build
# configure using host-config
cmake -C ../../host-configs/other/llnl-surface-chaos_5_x86_64_ib-gcc@4.9.3.cmake  \
      -DBLT_SOURCE_DIR=../../../../blt  ..
```

After building (`make`), you can run `make test` on a batch node (where the GPUs reside) to run the unit tests that are using MPI and CUDA:

```
bash-4.1$ salloc -A <valid bank>
bash-4.1$ make
bash-4.1$ make test

Running tests...
Test project blt/docs/tutorial/calc_pi/build
    Start 1: test_1
1/8 Test #1: test_1 ............................   Passed    0.01 sec
    Start 2: test_2
2/8 Test #2: test_2 ............................   Passed    2.79 sec
    Start 3: test_3
3/8 Test #3: test_3 ............................   Passed    0.54 sec
    Start 4: blt_gtest_smoke
4/8 Test #4: blt_gtest_smoke ...................   Passed    0.01 sec
    Start 5: blt_fruit_smoke
5/8 Test #5: blt_fruit_smoke ...................   Passed    0.01 sec
    Start 6: blt_mpi_smoke
6/8 Test #6: blt_mpi_smoke .....................   Passed    2.82 sec
    Start 7: blt_cuda_smoke
7/8 Test #7: blt_cuda_smoke ....................   Passed    0.48 sec
    Start 8: blt_cuda_runtime_smoke
8/8 Test #8: blt_cuda_runtime_smoke ............   Passed    0.11 sec

100% tests passed, 0 tests failed out of 8

Total Test time (real) =   6.80 sec
```

Building and testing on Ray

Here is how you can use the host-config file to configure a build of the `calc_pi` project with MPI and CUDA enabled on the blue_os Ray cluster:

```
# load new cmake b/c default on ray is too old
ml cmake
# create build dir
mkdir build
cd build
# configure using host-config
```

(continues on next page)

(continued from previous page)

```
cmake -C ../../host-configs/llnl/blueos_3_ppc64le_ib_p9/clang@upstream_nvcc_xlf.cmake␣
↪\
      -DBLT_SOURCE_DIR=../../../../blt  ..
```

And here is how to build and test the code on Ray:

```
bash-4.2$ lalloc 1 -G <valid group>
bash-4.2$ make
bash-4.2$ make test

Running tests...
Test project projects/blt/docs/tutorial/calc_pi/build
    Start 1: test_1
1/7 Test #1: test_1 ............................   Passed    0.01 sec
    Start 2: test_2
2/7 Test #2: test_2 ............................   Passed    1.24 sec
    Start 3: test_3
3/7 Test #3: test_3 ............................   Passed    0.17 sec
    Start 4: blt_gtest_smoke
4/7 Test #4: blt_gtest_smoke ...................   Passed    0.01 sec
    Start 5: blt_mpi_smoke
5/7 Test #5: blt_mpi_smoke .....................   Passed    0.82 sec
    Start 6: blt_cuda_smoke
6/7 Test #6: blt_cuda_smoke ....................   Passed    0.15 sec
    Start 7: blt_cuda_runtime_smoke
7/7 Test #7: blt_cuda_runtime_smoke ............   Passed    0.04 sec

100% tests passed, 0 tests failed out of 7

Total Test time (real) =   2.47 sec
```

3.1.6 Creating Documentation

BLT provides macros to build documentation using Sphinx and Doxygen.

Sphinx is the documentation system used by the Python programming language project (among many others).

Doxygen is a widely used system that generates documentation from annotated source code. Doxygen is heavily used for documenting C++ software.

Sphinx and Doxygen are not built into BLT, so the `sphinx-build` and `doxygen` executables must be available via a user's `PATH` at configuration time, or explicitly specified using the CMake variables `SPHINX_EXECUTABLE` and `DOXYGEN_EXECUTABLE`.

Here is an example of setting `sphinx-build` and `doxygen` paths in a host-config file:

```
set(SPHINX_EXECUTABLE "/usr/bin/sphinx-build" CACHE FILEPATH "")

set(DOXYGEN_EXECUTABLE "/usr/bin/doxygen" CACHE FILEPATH "")
```

The `calc_pi` example provides examples of both Sphinx and Doxygen documentation.

Calc Pi Sphinx Example

Sphinx is a python package that depends on several other packages. It can be installed via spack, pip, anaconda, etc…

`sphinx-build` processes a `config.py` file which includes a tree of *reStructuredText* files. The Sphinx `sphinx-quickstart` utility helps you generate a new sphinx project, including selecting common settings for the `config.py`.

BLT provides a `blt_add_sphinx_target()` macro which, which will look for a `conf.py` file in the current directory and add a command to build the Sphinx docs using this file to the `docs` CMake target.

---

blt_add_sphinx_target

A macro to create a named sphinx target for user documentation. Assumes there is a `conf.py` sphinx configuration file in the current directory. This macro is active when BLT is configured with a valid `SPHINX_EXECUTABLE` path.

Here is an example of using `blt_add_sphinx_target()` in a CMakeLists.txt file:

```
#------------------------------------------------------------------------------
# add a target to generate documentation with sphinx
#------------------------------------------------------------------------------

if(SPHINX_FOUND)
    blt_add_sphinx_target( calc_pi_sphinx )
endif()
```

Here is the example reStructuredText file that contains documentation for the *calc_pi* example.

```
.. Calc Pi documentation master file, created by
   sphinx-quickstart on Sun Sep 10 21:47:20 2017.
   You can adapt this file completely to your liking, but it should at least
   contain the root `toctree` directive.

Welcome to Calc Pi's documentation!
==================================


This is a tutorial example for BLT (https://github.com/llnl/blt) that creates
C++ libraries that calculate :math:`\pi` serially and in parallel using MPI
and CUDA.

These libraries calculate :math:`\pi` by approximating the integral
:math:`f(x) = \int_0^14/(1+x^2)` using numerical integration.
In the MPI implementation, the intervals are distributed across MPI tasks and
a MPI_AllReduce calculates the final result. In the CUDA implementation, the
intervals are distributed across CUDA blocks and threads and a tree reduction
calculates the final result.


The method is adapted from:
https://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples-usingmpi/simplempi/
↪cpi_c.html
```

Calc Pi Doxygen Example

Doxygen is a compiled executable that can be installed via spack, built-by-hand, etc…

`doxygen` processes a `Doxyfile` which specifies options, including where to look for annotated source files.

BLT provides a `blt_add_doxygen_target()` macro which, which will look for a `Doxyfile.in` file in the current directory, configure this file to create a `Doxyfile` in the build directory, and add a command to build the

Doxygen docs using this file to the `docs` CMake target.

---

blt_add_doxygen_target

A macro to create a named doxygen target for API documentation. Assumes there is a `Doxyfile.in` doxygen configuration file in the current directory. This macro is active when BLT is configured with a valid `DOXYGEN_EXECUTABLE` path.

Here is an example of using `blt_add_doxygen_target()` in a CMakeLists.txt file:

```
#------------------------------------------------------------------------------
# add a target to generate documentation with Doxygen
#------------------------------------------------------------------------------

if(DOXYGEN_FOUND)
    blt_add_doxygen_target( calc_pi_doxygen )
endif()
```

Here is the example `Doxyfile.in` file that is configured by CMake and passed to `doxygen`.

```
#---------------------------------------
# Doxygen Config for Calc Pi Example
#---------------------------------------

PROJECT_NAME  = Calc Pi
PROJECT_BRIEF = "Calc Pi"

INPUT = @CMAKE_CURRENT_SOURCE_DIR@/../../

GENERATE_XML   = NO
GENERATE_LATEX = NO

RECURSIVE      = NO
STRIP_CODE_COMMENTS = NO
```

Building the Calc Pi Example Documentation

Here is an example of building both the calc_pi Sphinx and Doxygen docs using the `docs` CMake target:

```
cd build-calc-pi
make docs
...
[ 50%] Building HTML documentation with Sphinx
[ 50%] Built target calc_pi_sphinx
[ 50%] Built target sphinx_docs
[100%] Generating API documentation with Doxygen
Searching for include files...
Searching for example files...
Searching for images...
Searching for dot files...
...
lookup cache used 3/65536 hits=3 misses=3
finished...
[100%] Built target calc_pi_doxygen
[100%] Built target doxygen_docs
[100%] Built target docs
```

After this, you can view the Sphinx docs at:

- `build-calc-pi/docs/sphinx/html/index.html`

and the Doxygen docs at:

- `build-calc-pi/docs/doxygen/html/index.html`

3.1.7 CMake Recommendations

This section includes several recommendations for how to wield CMake. Some of them are embodied in BLT, others are broader suggestions for CMake bliss.

Disable in-source builds

BLT Enforces This

In-source builds clutter source code with temporary build files and prevent other out-of-source builds from being created. Disabling in-source builds avoids clutter and accidental checkins of temporary build files.

Avoid using globs to identify source files

Globs are evaluated at CMake configure time - not build time. This means CMake will not detect new source files when they are added to the file system unless there are other changes that trigger CMake to reconfigure.

The CMake documentation also warns against this: https://cmake.org/cmake/help/v3.10/command/file.html?highlight=glob#file

Use arguments instead of options in CMake Macros and Functions

`CMAKE_PARSE_ARGUMENTS` allows Macros or Functions to support options. Options are enabled by passing them by name when calling a Macro or Function. Because of this, wrapping an existing Macro or Function in a way that passes through options requires if tests and multiple copies of the call. For example:

```
if(OPTION)
    my_function(arg1 arg2 arg3 OPTION)
else()
    my_function(arg1 arg2 arg3)
endif()
```

Adding more options compounds the logic to achieve these type of calls.

To simplify calling logic, we recommend using an argument instead of an option.

```
if(OPTION)
    set(arg4_value ON)
endif()

my_function(arg1 arg2 arg3 ${arg4_value})
```

Prefer explicit paths to locate third-party dependencies

Require passing explicit paths (ex: `ZZZ_DIR`) for third-party dependency locations. This avoids surprises with incompatible installs sprinkled in various system locations. If you are using off-the-shelf *FindZZZ* logic, also consider adding CMake checks to verify that *FindZZZ* logic actually found the dependencies at the location specified.

Emit a configure error if an explicitly identified third-party dependency is not found or an incorrect version is found.

If an explicit path to a dependency is given (ex: `ZZZ_DIR`) it should be valid or result in a CMake configure error.

In contrast, if you only issue a warning and automatically disable a feature when a third-party dependency is bad, the warning often goes unnoticed and may not be caught until folks using your software are surprised. Emitting a configure error stops CMake and draws attention to the fact that something is wrong. Optional dependencies are still supported by including them only if an explicit path to the dependency is provided (ex: `ZZZ_DIR`).

Add headers as source files to targets

BLT Macros Support This

This ensures headers are tracked as dependencies and are included in the projects created by CMake's IDE generators, like Xcode or Eclipse.

Always support *make install*

This allows CMake to do the right thing based on `CMAKE_INSTALL_PREFIX`, and also helps support CPack create release packages. This is especially important for libraries. In addition to targets, header files require an explicit install command.

Here is an example that installs a target and its headers:

```
#------------------------------------------------------
# Install Targets for example lib
#------------------------------------------------------
install(FILES ${example_headers} DESTINATION include)
install(TARGETS example
  EXPORT example
  LIBRARY DESTINATION lib
  ARCHIVE DESTINATION lib
)
```

3.2 API Documentation

3.2.1 Target Macros

blt_add_benchmark

```
blt_add_benchmark( NAME           [name]
                   COMMAND        [command]
                   NUM_MPI_TASKS [n])
```

Adds a benchmark to the project.

NAME Name that CTest reports.

COMMAND Command line that will be used to run the test and can include arguments.

NUM_MPI_TASKS Indicates this is an MPI test and how many MPI tasks to use.

This macro adds a benchmark test to the `Benchmark` CTest configuration which can be run by the `run_benchmarks` build target. These tests are not run when you use the regular `test` build target.

This macro is just a thin wrapper around *blt_add_test* and assists with building up the correct command line for running the benchmark. For more information see *blt_add_test*.

The underlying executable should be previously added to the build system with *blt_add_executable*. It should include the necessary benchmarking library in its DEPENDS_ON list.

Any calls to this macro should be guarded with ENABLE_BENCHMARKS unless this option is always on in your build project.

Note: BLT provides a built-in Google Benchmark that is enabled by default if you set ENABLE_BENCHMARKS=ON and can be turned off with the option ENABLE_GBENCHMARK.

Listing 1: **Example**

```
if(ENABLE_BENCHMARKS)
    blt_add_executable(NAME    component_benchmark
                       SOURCES my_benchmark.cpp
                       DEPENDS gbenchmark)
    blt_add_benchmark(
         NAME    component_benchmark
         COMMAND component_benchmark "--benchmark_min_time=0.0 --v=3 --benchmark_
↪format=json")
endif()
```

blt_add_executable

```
blt_add_executable( NAME       <name>
                    SOURCES    [source1 [source2 ...]]
                    INCLUDES   [dir1 [dir2 ...]]
                    DEFINES    [define1 [define2 ...]]
                    DEPENDS_ON [dep1 [dep2 ...]]
                    OUTPUT_DIR [dir]
                    FOLDER     [name])
```

Adds an executable target, called <name>, to be built from the given sources.

The INCLUDES argument allows you to define what include directories are needed to compile this executable.

The DEFINES argument allows you to add needed compiler definitions that are needed to compile this executable.

If given a DEPENDS_ON argument, it will add the necessary includes and libraries if they are already registered with blt_register_library. If not it will add them as a cmake target dependency.

The OUTPUT_DIR is used to control the build output directory of this executable. This is used to overwrite the default bin directory.

If the first entry in SOURCES is a Fortran source file, the fortran linker is used. (via setting the CMake target property LINKER_LANGUAGE to Fortran) FOLDER is an optional keyword to organize the target into a folder in an IDE.

This is available when ENABLE_FOLDERS is ON and when using a cmake generator that supports this feature and will otherwise be ignored.

blt_add_library

```
blt_add_library( NAME         <libname>
                 SOURCES      [source1 [source2 ...]]
                 HEADERS      [header1 [header2 ...]]
                 INCLUDES     [dir1 [dir2 ...]]
                 DEFINES      [define1 [define2 ...]]
                 DEPENDS_ON   [dep1 ...]
                 OUTPUT_NAME  [name]
                 OUTPUT_DIR   [dir]
                 SHARED       [TRUE | FALSE]
                 OBJECT       [TRUE | FALSE]
                 CLEAR_PREFIX [TRUE | FALSE]
                 FOLDER       [name])
```

Adds a library target to your project.

NAME Name of the created CMake target

SOURCES List of all sources to be added

HEADERS List of all headers to be added

INCLUDES List of include directories both used by this target and inherited by dependent targets

DEFINES List of compiler defines both used by this target and inherited by dependent targets

DEPENDS_ON List of CMake targets and BLT registered libraries that this library depends on

OUTPUT_NAME Override built file name of library (defaults to <NAME>)

OUTPUT_DIR Directory that this target will built to

SHARED Builds library as shared and overrides global BUILD_SHARED_LIBS (defaults to OFF)

OBJECT Create an Object library

CLEAR_PREFIX Removes library prefix (defaults to 'lib' on linux)

FOLDER Name of the IDE folder to ease organization

This macro supports three types of libraries automatically: normal, header-only, or object.

Normal libraries are libraries that have sources that are compiled and linked into a single library and have headers that go along with them (unless it's a Fortran library).

Header-only libraries are useful when you do not want the library separately compiled or are using C++ templates that require the library's user to instatiate them. These libraries have headers but no sources. To create a header-only library (CMake calls them INTERFACE libraries), simply list all headers under the HEADER argument and do not specify SOURCES (because there aren't any).

Object libraries are basically a collection of compiled source files that are not archived or linked. They are sometimes useful when you want to solve compilicated linking problems (like circular dependencies) or when you want to combine smaller libraries into one larger library but don't want the linker to remove unused symbols. Unlike regular CMake object libraries you do not have to use the `$<TARGET_OBJECTS:<libname>>` syntax, you can just use <libname> with BLT macros. Unless you have a good reason don't use Object libraries.

Note: BLT Object libraries do not follow CMake's normal transitivity rules. Due to CMake requiring you install the individual object files if you install the target that uses them. BLT manually adds the INTERFACE target properties to get around this.

This macro uses the BUILD_SHARED_LIBS, which is defaulted to OFF, to determine whether the library will be built as shared or static. The optional boolean SHARED argument can be used to override this choice.

If given a DEPENDS_ON argument, this macro will inherit the necessary information from all targets given in the list. This includes CMake targets as well as any BLT registered libraries already defined via *blt_register_library*. To ease use, all information is used by this library and inherited by anything depending on this library (CMake PUBLIC inheritance).

OUTPUT_NAME is useful when multiple libraries with the same name need to be created by different targets. For example, you might want to build both a shared and static library in the same build instead of building twice, once with BUILD_SHARED_LIBS set to ON and then with OFF. NAME is the CMake target name, OUTPUT_NAME is the created library name.

Note: The FOLDER option is only used when ENABLE_FOLDERS is ON and when the CMake generator supports this feature and will otherwise be ignored.

blt_add_test

```
blt_add_test( NAME            [name]
              COMMAND         [command]
              NUM_MPI_TASKS   [n]
              NUM_OMP_THREADS [n]
              CONFIGURATIONS  [config1 [config2...]])
```

Adds a test to the project.

NAME Name that CTest reports.

COMMAND Command line that will be used to run the test and can include arguments.

NUM_MPI_TASKS Indicates this is an MPI test and how many MPI tasks to use.

NUM_OMP_THREADS Indicates this test requires the defined environment variable OMP_NUM_THREADS set to the given variable.

CONFIGURATIONS Set the CTest configuration for this test. When not specified, the test will be added to the default CTest configuration.

This macro adds the named test to CTest, which is run by the build target `test`. This macro does not build the executable and requires a prior call to *blt_add_executable*.

This macro assists with building up the correct command line. It will prepend the RUNTIME_OUTPUT_DIRECTORY target property to the executable.

If NUM_MPI_TASKS is given or ENABLE_WRAP_ALL_TESTS_WITH_MPIEXEC is set, the macro will appropiately use MPIEXEC, MPIEXEC_NUMPROC_FLAG, and BLT_MPI_COMMAND_APPEND to create the MPI run line.

MPIEXEC and MPIEXEC_NUMPROC_FLAG are filled in by CMake's FindMPI.cmake but can be overwritten in your host-config specific to your platform. BLT_MPI_COMMAND_APPEND is useful on machines that require extra arguments to MPIEXEC.

If NUM_OMP_THREADS is given, this macro will set the environment variable OMP_NUM_THREADS before running this test. This is done by appending to the CMake tests property.

Note: If you do not require this macros command line assistance, you can call CMake's `add_test()` directly. For example, you may have a script checked into your repository you wish to run as a test instead of an executable you built as a part of your build system.

Any calls to this macro should be guarded with ENABLE_TESTS unless this option is always on in your build project.

Listing 2: **Example**

```
if (ENABLE_TESTS)
    blt_add_executable(NAME    my_test
                       SOURCES my_test.cpp)
    blt_add_test(NAME    my_test
                 COMMAND my_test --with-some-argument)
endif()
```

blt_register_library

```
blt_register_library( NAME                     <libname>
                      DEPENDS_ON               [dep1 [dep2 ...]]
                      INCLUDES                 [include1 [include2 ...]]
                      TREAT_INCLUDES_AS_SYSTEM [ON|OFF]
                      FORTRAN_MODULES          [path1 [path2 ..]]
                      LIBRARIES                [lib1 [lib2 ...]]
                      COMPILE_FLAGS            [flag1 [flag2 ..]]
                      LINK_FLAGS               [flag1 [flag2 ..]]
                      DEFINES                  [def1 [def2 ...]] )
```

Registers a library to the project to ease use in other BLT macro calls.

Stores information about a library in a specific way that is easily recalled in other macros. For example, after registering gtest, you can add gtest to the DEPENDS_ON in your blt_add_executable call and it will add the INCLUDES and LIBRARIES to that executable.

TREAT_INCLUDES_AS_SYSTEM informs the compiler to treat this library's include paths as system headers. Only some compilers support this. This is useful if the headers generate warnings you want to not have them reported in your build. This defaults to OFF.

This does not actually build the library. This is strictly to ease use after discovering it on your system or building it yourself inside your project.

Note: The OBJECT parameter is for internal BLT support for object libraries and is not for users. Object libraries are created using blt_add_library().

Internally created variables (NAME = "foo"):

BLT_FOO_IS_REGISTERED_LIBRARY
BLT_FOO_IS_OBJECT_LIBRARY
BLT_FOO_DEPENDS_ON
BLT_FOO_INCLUDES
BLT_FOO_TREAT_INCLUDES_AS_SYSTEM
BLT_FOO_FORTRAN_MODULES
BLT_FOO_LIBRARIES

BLT_FOO_COMPILE_FLAGS
BLT_FOO_LINK_FLAGS
BLT_FOO_DEFINES

3.2.2 Target Property Macros

blt_add_target_compile_flags

```
blt_add_target_compile_flags( TO    <target>
                              SCOPE <PUBLIC (Default)| INTERFACE | PRIVATE>
                              FLAGS [FOO [BAR ...]])
```

Appends compiler flags to a CMake target by appending to the target's existing flags.

TO Name of CMake target

SCOPE Defines the scope of the given flags. Defaults to PUBLIC and is case insensitive.

FLAGS List of compile flags

This macro provides very similar functionality to CMake's native `add_compile_options` and `target_compile_options` commands, but provides more fine-grained scoping for the compile flags on a per target basis.

The given target must be added via CMake's `add_executable` or `add_library` commands or with the corresponding `blt_add_executable` and `blt_add_library` macros.

PRIVATE flags are used for the given target. INTERFACE flags are inherited by any target that depends on this target. PUBLIC flags are both INTERFACE and PRIVATE.

Note: This macro will strip away leading and trailing whitespace from each flag.

---

blt_add_target_definitions

```
blt_add_target_definitions( TO    <target>
                            SCOPE <PUBLIC (Default)| INTERFACE | PRIVATE>
                            TARGET_DEFINITIONS [FOO [BAR ...]])
```

Appends pre-processor definitions to the given target's existing flags.

TO Name of CMake target

SCOPE Defines the scope of the given definitions. Defaults to PUBLIC and is case insensitive.

FLAGS List of definitions flags

This macro provides very similar functionality to CMake's native `add_definitions` and `target_add_defintions` commands, but provides more fine-grained scoping for the compile definitions on a per target basis. Given a list of definitions, e.g., FOO and BAR, this macro adds compiler definitions to the compiler command for the given target, i.e., it will pass -DFOO and -DBAR.

The given target must be added via CMake's `add_executable` or `add_library` commands or with the corresponding `blt_add_executable` and `blt_add_library` macros.

PRIVATE flags are used for the given target. INTERFACE flags are inherited by any target that depends on this target. PUBLIC flags are both INTERFACE and PRIVATE.

Note: The target definitions can either include or omit the "-D" characters. E.g. the following are all valid ways to add two compile definitions (A=1 and B) to target 'foo'.

Note: This macro will strip away leading and trailing whitespace from each definition.

Listing 3: **Example**

```
blt_add_target_definitions(TO foo TARGET_DEFINITIONS A=1 B)
blt_add_target_definitions(TO foo TARGET_DEFINITIONS -DA=1 -DB)
blt_add_target_definitions(TO foo TARGET_DEFINITIONS "A=1;-DB")
blt_add_target_definitions(TO foo TARGET_DEFINITIONS " " -DA=1;B)
```

blt_add_target_link_flags

```
blt_add_target_link_flags( TO    <target>
                           SCOPE <PUBLIC (Default)| INTERFACE | PRIVATE>
                           FLAGS [FOO [BAR ...]])
```

Appends linker flags to a the given target's existing flags.

TO Name of CMake target

SCOPE Defines the scope of the given flags. Defaults to PUBLIC and is case insensitive.

FLAGS List of linker flags

This macro provides very similar functionality to CMake's native `add_link_options` and `target_link_options` commands, but provides more fine-grained scoping for the compile definitions on a per target basis.

The given target must be added via CMake's `add_executable` or `add_library` commands or with the corresponding `blt_add_executable` and `blt_add_library` macros.

PRIVATE flags are used for the given target. INTERFACE flags are inherited by any target that depends on this target. PUBLIC flags are both INTERFACE and PRIVATE.

If *CUDA_LINK_WITH_NVCC* is set to ON, this macro will automatically convert "-Wl,-rpath," to "-Xlinker -rpath -Xlinker ".

Note: This macro also handles the various changes that CMake made in 3.13. For example, the target property LINK_FLAGS was changes to LINK_OPTIONS and was changed from a string to a list. New versions now support Generator Expressions. Also pre-3.13, there were no macros to add link flags to targets so we do this by setting the properties directly.

Note: In CMake versions prior to 3.13, this list is converted to a string internally and any ; characters will be removed.

Note: In CMake versions 3.13 and above, this list is prepended with "SHELL:" which stops CMake from de-duplicating flags. This is especially bad when linking with NVCC when you have groups of flags like "-Xlinker -rpath -Xlinker <directory>".

blt_print_target_properties

```
blt_print_target_properties(TARGET <target>)
```

Prints out all properties of the given target.

TARGET Name of CMake target

The given target must be added via `add_executable` or `add_library` or with the corresponding `blt_add_executable`, `blt_add_library`, or `blt_register_library` macros.

Output is of the form for each property:

[<target> property] <property>: <value>

blt_set_target_folder

```
blt_set_target_folder( TARGET <target>
                       FOLDER <folder>)
```

Sets the FOLDER property of the given CMake target.

TARGET Name of CMake target

FOLDER Name of the folder

This is used to organize properties in an IDE.

This feature is only available when BLT's *ENABLE_FOLDERS* option is ON and in CMake generators that support folders (but is safe to call regardless of the generator or value of ENABLE_FOLDERS).

Note: Do not use this macro on header-only (INTERFACE) library targets, since this will generate a CMake configuration error.

3.2.3 Utility Macros

blt_assert_exists

```
blt_assert_exists(
  [DIRECTORIES <dir1> [<dir2> ...] ]
  [FILES <file1> [<file2> ...] ]
  [TARGETS <target1> [<target2> ...] ] )
```

Checks if the specified directory, file and/or cmake target exists and throws an error message.

Note: The behavior for checking if a given file or directory exists is well-defined only for absolute paths.

Listing 4: **Example**

```
1 ## check if the directory 'blt' exists in the project
2 blt_assert_exists( DIRECTORIES ${PROJECT_SOURCE_DIR}/cmake/blt )
3
4 ## check if the file 'SetupBLT.cmake' file exists
```

(continues on next page)

(continued from previous page)

```
blt_assert_exists( FILES ${PROJECT_SOURCE_DIR}/cmake/blt/SetupBLT.cmake )

## checks can also be bundled in one call
blt_assert_exists( DIRECTORIES ${PROJECT_SOURCE_DIR}/cmake/blt
                   FILES ${PROJECT_SOURCE_DIR}/cmake/blt/SetupBLT.cmake )

## check if the CMake targets `foo` and `bar` exist
blt_assert_exists( TARGETS foo bar )
```

blt_append_custom_compiler_flag

```
blt_append_custom_compiler_flag(
                    FLAGS_VAR  flagsVar       (required)
                    DEFAULT    defaultFlag    (optional)
                    GNU        gnuFlag        (optional)
                    CLANG      clangFlag      (optional)
                    HCC        hccFlag        (optional)
                    INTEL      intelFlag      (optional)
                    XL         xlFlag         (optional)
                    MSVC       msvcFlag       (optional)
                    MSVC_INTEL msvcIntelFlag  (optional)
                    PGI        pgiFlag        (optional)
                    CRAY       crayFlag       (optional))
```

Appends compiler-specific flags to a given variable of flags

If a custom flag is given for the current compiler, we use that. Otherwise, we will use the DEFAULT flag (if present).

If ENABLE_FORTRAN is On, any flagsVar with "fortran" (any capitalization) in its name will pick the compiler family (GNU,CLANG, INTEL, etc) based on the fortran compiler family type. This allows mixing C and Fortran compiler families, e.g. using Intel fortran compilers with clang C compilers.

When using the Intel toolchain within visual studio, we use the MSVC_INTEL flag, when provided, with a fallback to the MSVC flag.

blt_find_libraries

```
blt_find_libraries( FOUND_LIBS <FOUND_LIBS variable name>
                    NAMES      [libname1 [libname2 ...]]
                    REQUIRED   [TRUE (default) | FALSE ]
                    PATHS      [path1 [path2 ...]])
```

This command is used to find a list of libraries.

If the libraries are found the results are appended to the given FOUND_LIBS variable name. NAMES lists the names of the libraries that will be searched for in the given PATHS.

If REQUIRED is set to TRUE, BLT will produce an error message if any of the given libraries are not found. The default value is TRUE.

PATH lists the paths in which to search for NAMES. No system paths will be searched.

blt_list_append

```
blt_list_append(TO       <list>
                ELEMENTS [<element>...]
                IF       <bool>)
```

Appends elements to a list if the specified bool evaluates to true.

This macro is essentially a wrapper around CMake's `list(APPEND ...)` command which allows inlining a conditional check within the same call for clarity and convenience.

This macro requires specifying:

- The target list to append to by passing `TO <list>`
- A condition to check by passing `IF <bool>`
- The list of elements to append by passing `ELEMENTS [<element>...]`

Note, the argument passed to the IF option has to be a single boolean value and cannot be a boolean expression since CMake cannot evaluate those inline.

Listing 5: **Example**

```
set(mylist A B)

set(ENABLE_C TRUE)
blt_list_append( TO mylist ELEMENTS C IF ${ENABLE_C} ) # Appends 'C'

set(ENABLE_D TRUE)
blt_list_append( TO mylist ELEMENTS D IF ENABLE_D ) # Appends 'D'

set(ENABLE_E FALSE)
blt_list_append( TO mylist ELEMENTS E IF ENABLE_E ) # Does not append 'E'

unset(_undefined)
blt_list_append( TO mylist ELEMENTS F IF _undefined ) # Does not append 'F'
```

blt_list_remove_duplicates

```
blt_list_remove_duplicates(TO <list>)
```

Removes duplicate elements from the given TO list.

This macro is essentially a wrapper around CMake's `list(REMOVE_DUPLICATES ...)` command but doesn't throw an error if the list is empty or not defined.

Listing 6: **Example**

```
set(mylist A B A)
blt_list_remove_duplicates( TO mylist )
```

3.2.4 Git Macros

blt_git

```
blt_git(SOURCE_DIR      <dir>
        GIT_COMMAND     <command>
        OUTPUT_VARIABLE <out>
        RETURN_CODE     <rc>
        [QUIET] )
```

Runs the supplied git command on the given Git repository.

This macro runs the user-supplied Git command, given by GIT_COMMAND, on the given Git repository corresponding to SOURCE_DIR. The supplied GIT_COMMAND is just a string consisting of the Git command and its arguments. The resulting output is returned to the supplied CMake variable provided by the OUTPUT_VARIABLE argument.

A return code for the Git command is returned to the caller via the CMake variable provided with the RETURN_CODE argument. A non-zero return code indicates that an error has occured.

Note, this macro assumes FindGit() was invoked and was successful. It relies on the following variables set by FindGit():

- Git_FOUND flag that indicates if git is found
- GIT_EXECUTABLE points to the Git binary

If Git_FOUND is "false" this macro will throw a FATAL_ERROR message.

Listing 7: **Example**

```
blt_git( SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}
         GIT_COMMAND describe --tags master
         OUTPUT_VARIABLE axom_tag
         RETURN_CODE rc )
if (NOT ${rc} EQUAL 0)
    message( FATAL_ERROR "blt_git failed!" )
endif()
```

blt_is_git_repo

```
blt_is_git_repo(OUTPUT_STATE <state>
                [SOURCE_DIR  <dir>] )
```

Checks if we are working with a valid Git repository.

This macro checks if the corresponding source directory is a valid Git repo. Nominally, the corresponding source directory that is used is set to ${CMAKE_CURRENT_SOURCE_DIR}. A different source directory may be optionally specified using the SOURCE_DIR argument.

The resulting state is stored in the CMake variable specified by the caller using the OUTPUT_STATE parameter.

Listing 8: **Example**

```
blt_is_git_repo( OUTTPUT_STATE is_git_repo )
if ( ${is_git_repo} )
    message(STATUS "Pointing to a valid Git repo!")
else()
    message(STATUS "Not a Git repo!")
endif()
```

blt_git_tag

```
blt_git_tag( OUTPUT_TAG  <tag>
             RETURN_CODE <rc>
             [SOURCE_DIR <dir>]
             [ON_BRANCH  <branch>] )
```

Returns the latest tag on a corresponding Git repository.

This macro gets the latest tag from a Git repository that can be specified via the SOURCE_DIR argument. If SOURCE_DIR is not supplied, the macro will use ${CMAKE_CURRENT_SOURCE_DIR}. By default the macro will return the latest tag on the branch that is currently checked out. A particular branch may be specified using the ON_BRANCH option.

The tag is stored in the CMake variable specified by the caller using the the OUTPUT_TAG parameter.

A return code for the Git command is returned to the caller via the CMake variable provided with the RETURN_CODE argument. A non-zero return code indicates that an error has occured.

Listing 9: **Example**

```
blt_git_tag( OUTPUT_TAG tag RETURN_CODE rc ON_BRANCH master )
if ( NOT ${rc} EQUAL 0 )
    message( FATAL_ERROR "blt_git_tag failed!" )
endif()
message( STATUS "tag=${tag}" )
```

blt_git_branch

```
blt_git_branch( BRANCH_NAME <branch>
                RETURN_CODE <rc>
                [SOURCE_DIR <dir>] )
```

Returns the name of the active branch in the checkout space.

This macro gets the name of the current active branch in the checkout space that can be specified using the SOURCE_DIR argument. If SOURCE_DIR is not supplied by the caller, this macro will point to the checkout space corresponding to ${CMAKE_CURRENT_SOURCE_DIR}.

A return code for the Git command is returned to the caller via the CMake variable provided with the RETURN_CODE argument. A non-zero return code indicates that an error has occured.

Listing 10: **Example**

```
blt_git_branch( BRANCH_NAME active_branch RETURN_CODE rc )
if ( NOT ${rc} EQUAL 0 )
    message( FATAL_ERROR "blt_git_tag failed!" )
endif()
message( STATUS "active_branch=${active_branch}" )
```

blt_git_hashcode

```
blt_git_hashcode( HASHCODE    <hc>
                  RETURN_CODE <rc>
```

(continues on next page)

(continued from previous page)

```
                  [SOURCE_DIR <dir>]
                  [ON_BRANCH  <branch>])
```

Returns the SHA-1 hashcode at the tip of a branch.

This macro returns the SHA-1 hashcode at the tip of a branch that may be specified with the ON_BRANCH argument. If the ON_BRANCH argument is not supplied, the macro will return the SHA-1 hash at the tip of the current branch. In addition, the caller may specify the target Git repository using the SOURCE_DIR argument. Otherwise, if SOURCE_DIR is not specified, the macro will use ${CMAKE_CURRENT_SOURCE_DIR}.

A return code for the Git command is returned to the caller via the CMake variable provided with the RETURN_CODE argument. A non-zero return code indicates that an error has occured.

Listing 11: **Example**

```
blt_git_hashcode( HASHCODE sha1 RETURN_CODE rc )
if ( NOT ${rc} EQUAL 0 )
    message( FATAL_ERROR "blt_git_hashcode failed!" )
endif()
message( STATUS "sha1=${sha1}" )
```

3.2.5 Code Check Macros

blt_add_code_checks

```
blt_add_code_checks( PREFIX               <Base name used for created targets>
                     SOURCES              [source1 [source2 ...]]
                     ASTYLE_CFG_FILE      <Path to AStyle config file>
                     CLANGFORMAT_CFG_FILE <Path to ClangFormat config file>
                     UNCRUSTIFY_CFG_FILE  <Path to Uncrustify config file>
                     CPPCHECK_FLAGS       <List of flags added to Cppcheck>)
```

This macro adds all enabled code check targets for the given SOURCES.

PREFIX Prefix used for the created code check build targets. For example: <PREFIX>_uncrustify_check

SOURCES Source list that the code checks will be ran on

ASTYLE_CFG_FILE Path to AStyle config file

CLANGFORMAT_CFG_FILE Path to ClangFormat config file

UNCRUSTIFY_CFG_FILE Path to Uncrustify config file

CPPCHECK_FLAGS List of flags added to Cppcheck

The purpose of this macro is to enable all code checks in the default manner. It runs all code checks from the working directory *CMAKE_BINARY_DIR*. If you need more specific functionality you will need to call the individual code check macros yourself.

Note: For library projects that may be included as a subproject of another code via CMake's add_subproject(), we recommend guarding "code check" targets against being included in other codes. The following check *if ("${PROJECT_SOURCE_DIR}" STREQUAL "${CMAKE_SOURCE_DIR}")* will stop your code checks from running unless you are the main CMake project.

Sources are filtered based on file extensions for use in these code checks. If you need additional file extensions defined add them to BLT_C_FILE_EXTS and BLT_Fortran_FILE_EXTS. Currently this macro only has code checks for C/C++ and simply filters out the Fortran files.

This macro supports code formatting with either AStyle, ClangFormat, or Uncrustify (but not all at the same time) only if the following requirements are met:

- AStyle
  - ASTYLE_CFG_FILE is given
  - ASTYLE_EXECUTABLE is defined and found prior to calling this macro
- ClangFormat
  - CLANGFORMAT_CFG_FILE is given
  - CLANGFORMAT_EXECUTABLE is defined and found prior to calling this macro
- Uncrustify
  - UNCRUSTIFY_CFG_FILE is given
  - UNCRUSTIFY_EXECUTABLE is defined and found prior to calling this macro

---

Note: ClangFormat does not support a command line option for config files. To work around this, we copy the given config file to the build directory where this macro runs from.

Enabled code formatting checks produce a *check* build target that will test to see if you are out of compliance with your code formatting and a *style* build target that will actually modify your source files. It also creates smaller child build targets that follow the pattern <*PREFIX*>_<*astyle*|*clangformat*|*uncrustify*>_<*check*|*style*>.

This macro supports the following static analysis tools with their requirements:

- CppCheck
  - CPPCHECK_EXECUTABLE is defined and found prior to calling this macro
  - <optional> CPPCHECK_FLAGS added to the cppcheck command line before the sources
- Clang-Query
  - CLANGQUERY_EXECUTABLE is defined and found prior to calling this macro

These are added as children to the *check* build target and produce child build targets that follow the pattern <*PREFIX*>_<*cppcheck*|*clangquery*>_*check*.

blt_add_clang_query_target

```
blt_add_clang_query_target( NAME              <Created Target Name>
                            WORKING_DIRECTORY <Working Directory>
                            COMMENT           <Additional Comment for Target␣
↪Invocation>
                            CHECKERS          <specifies a subset of checkers>
                            DIE_ON_MATCH      <TRUE | FALSE (default)>
                            SRC_FILES         [source1 [source2 ...]])
```

Creates a new build target for running clang-query.

NAME Name of created build target

WORKING_DIRECTORY Directory in which the clang-query command is run. Defaults to where macro is called.

COMMENT Comment prepended to the build target output

CHECKERS list of checkers to be run by created build target

DIE_ON_MATCH Causes build failure on first clang-query match. Defaults to FALSE.S

SRC_FILES Source list that clang-query will be ran on

Clang-query is a tool used for examining and matching the Clang AST. It is useful for enforcing coding standards and rules on your source code. A good primer on how to use clang-query can be found here.

Turning on DIE_ON_MATCH is useful if you're using this in CI to enforce rules about your code.

CHECKERS are the static analysis passes to specifically run on the target. The following checker options can be given:

- (no value) : run all available static analysis checks found
- (checker1:checker2) : run checker1 and checker2
- (interpreter) : run the clang-query interpeter to interactively develop queries

blt_add_cppcheck_target

```
blt_add_cppcheck_target( NAME                <Created Target Name>
                         WORKING_DIRECTORY   <Working Directory>
                         PREPEND_FLAGS       <Additional flags for cppcheck>
                         APPEND_FLAGS        <Additional flags for cppcheck>
                         COMMENT             <Additional Comment for Target␣
↪Invocation>
                         SRC_FILES           [source1 [source2 ...]] )
```

Creates a new build target for running cppcheck

NAME Name of created build target

WORKING_DIRECTORY Directory in which the clang-query command is run. Defaults to where macro is called.

PREPEND_FLAGS Additional flags added to the front of the cppcheck flags

APPEND_FLAGS Additional flags added to the end of the cppcheck flags

COMMENT Comment prepended to the build target output

SRC_FILES Source list that cppcheck will be ran on

Cppcheck is a static analysis tool for C/C++ code. More information about Cppcheck can be found here.

blt_add_astyle_target

```
blt_add_astyle_target( NAME              <Created Target Name>
                       MODIFY_FILES      [TRUE | FALSE (default)]
                       CFG_FILE          <AStyle Configuration File>
                       PREPEND_FLAGS     <Additional Flags to AStyle>
                       APPEND_FLAGS      <Additional Flags to AStyle>
                       COMMENT           <Additional Comment for Target Invocation>
                       WORKING_DIRECTORY <Working Directory>
                       SRC_FILES         [FILE1 [FILE2 ...]] )
```

Creates a new build target for running AStyle

NAME Name of created build target

MODIFY_FILES Modify the files in place. Defaults to FALSE.

CFG_FILE Path to AStyle config file

PREPEND_FLAGS Additional flags added to the front of the AStyle flags

APPEND_FLAGS Additional flags added to the end of the AStyle flags

COMMENT Comment prepended to the build target output

WORKING_DIRECTORY Directory in which the AStyle command is run. Defaults to where macro is called.

SRC_FILES Source list that AStyle will be ran on

AStyle is a Source Code Beautifier for C/C++ code. More information about AStyle can be found here.

When MODIFY_FILES is set to TRUE, modifies the files in place and adds the created build target to the parent *style* build target. Otherwise the files are not modified and the created target is added to the parent *check* build target. This target will notify you which files do not conform to your style guide.

Note: Setting MODIFY_FILES to FALSE is only supported in AStyle v2.05 or greater.

---

blt_add_clangformat_target

```
blt_add_clangformat_target( NAME              <Created Target Name>
                            MODIFY_FILES      [TRUE | FALSE (default)]
                            CFG_FILE          <ClangFormat Configuration File>
                            PREPEND_FLAGS     <Additional Flags to ClangFormat>
                            APPEND_FLAGS      <Additional Flags to ClangFormat>
                            COMMENT           <Additional Comment for Target␣
↪Invocation>
                            WORKING_DIRECTORY <Working Directory>
                            SRC_FILES         [FILE1 [FILE2 ...]] )
```

Creates a new build target for running ClangFormat

NAME Name of created build target

MODIFY_FILES Modify the files in place. Defaults to FALSE.

CFG_FILE Path to ClangFormat config file

PREPEND_FLAGS Additional flags added to the front of the ClangFormat flags

APPEND_FLAGS Additional flags added to the end of the ClangFormat flags

COMMENT Comment prepended to the build target output

WORKING_DIRECTORY Directory in which the ClangFormat command is run. Defaults to where macro is called.

SRC_FILES Source list that ClangFormat will be ran on

ClangFormat is a Source Code Beautifier for C/C++ code. More information about ClangFormat can be found here.

When MODIFY_FILES is set to TRUE, modifies the files in place and adds the created build target to the parent *style* build target. Otherwise the files are not modified and the created target is added to the parent *check* build target. This target will notify you which files do not conform to your style guide.

Note: ClangFormat does not support a command line option for config files. To work around this, we copy the given config file to the given working directory. We recommend using the build directory *${PROJECT_BINARY_DIR}*. Also if someone is directly including your CMake project in theirs, you may conflict with theirs. We recommend

guarding your code checks against this with the following check *if ("${PROJECT_SOURCE_DIR}" STREQUAL "${CMAKE_SOURCE_DIR}")*.

Note: ClangFormat does not support a command line option for check (–dry-run) until version 10. This version is not widely used or available at this time. To work around this, we use an included script called run-clang-format.py that does not use PREPEND_FLAGS or APPEND_FLAGS in the *check* build target because the script does not support command line flags passed to *clang-format*. This script is not used in the *style* build target.

blt_add_uncrustify_target

```
blt_add_uncrustify_target( NAME              <Created Target Name>
                           MODIFY_FILES      [TRUE | FALSE (default)]
                           CFG_FILE          <Uncrustify Configuration File>
                           PREPEND_FLAGS     <Additional Flags to Uncrustify>
                           APPEND_FLAGS      <Additional Flags to Uncrustify>
                           COMMENT           <Additional Comment for Target␣
↪Invocation>
                           WORKING_DIRECTORY <Working Directory>
                           SRC_FILES         [source1 [source2 ...]] )
```

Creates a new build target for running Uncrustify

NAME Name of created build target

MODIFY_FILES Modify the files in place. Defaults to FALSE.

CFG_FILE Path to Uncrustify config file

PREPEND_FLAGS Additional flags added to the front of the Uncrustify flags

APPEND_FLAGS Additional flags added to the end of the Uncrustify flags

COMMENT Comment prepended to the build target output

WORKING_DIRECTORY Directory in which the Uncrustify command is run. Defaults to where macro is called.

SRC_FILES Source list that Uncrustify will be ran on

Uncrustify is a Source Code Beautifier for C/C++ code. More information about Uncrustify can be found here.

When MODIFY_FILES is set to TRUE, modifies the files in place and adds the created build target to the parent *style* build target. Otherwise the files are not modified and the created target is added to the parent *check* build target. This target will notify you which files do not conform to your style guide. .. Note:

```
Setting MODIFY_FILES to FALSE is only supported in Uncrustify v0.61 or greater.
```

3.2.6 Documenation Macros

blt_add_doxygen_target

```
blt_add_doxygen_target(doxygen_target_name)
```

Creates a build target for invoking doxygen to generate docs. Expects to find a Doxyfile.in in the directory the macro is called in.

This macro sets up the doxygen paths so that the doc builds happen out of source. For `make install`, this will place the resulting docs in docs/doxygen/<doxygen_target_name>.

blt_add_sphinx_target

```
blt_add_sphinx_target(sphinx_target_name)
```

Creates a build target for invoking sphinx to generate docs. Expects to find a conf.py or conf.py.in in the directory the macro is called in.

If conf.py is found, it is directly used as input to sphinx.

If conf.py.in is found, this macro uses CMake's configure_file() command to generate a conf.py, which is then used as input to sphinx.

This macro sets up the sphinx paths so that the doc builds happen out of source. For `make install`, this will place the resulting docs in docs/sphinx/<sphinx_target_name>.