BLT Tutorial Guide

BLT Team

Jul 21, 2021

Table of Contents

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.

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
  - Handles differences across CMake versions
  - Multi-platform support (HPC Platforms, OSX, Windows)
- Batteries included
  - Built-in support for HPC Basics: MPI, OpenMP, CUDA, and HIP
  - 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

Questions

Any questions can be sent to blt-dev@llnl.gov. If you are an LLNL employee or collaborator, we have an internal Microsoft Teams group chat named "BLT" as well.

CHAPTER 3

Contributions

We welcome all kinds of contributions: new features, bug fixes, documentation edits.

To contribute, make a pull request, with `develop` as the destination branch. We use CI testing and your branch must pass these tests before being merged.

For more information, see the contributing guide.

CHAPTER 4

Authors

Thanks to all of BLT's contributors.

4.1 User Tutorial

This tutorial provides instructions for:

- Adding BLT to a CMake project
- Building, linking, and installing libraries and executables
- Setting up unit tests with GTest
- Setting up *host-config* files to handle multiple platform configurations
- Using external project dependencies
- Exporting your project's CMake targets for outside projects to use
- Creating documentation with Sphinx and Doxygen

The two example CMake projects used are included in BLT's source tree at:

- `<blt-dir>/cmake/docs/tutorial/bare_bones`
- `<blt-dir>/cmake/docs/tutorial/calc_pi`

Here are direct links to the projects in BLT's GitHub repo:

- https://github.com/LLNL/blt/tree/develop/docs/tutorial/bare_bones
- https://github.com/LLNL/blt/tree/develop/docs/tutorial/calc_pi

`bare_bones` provides a minimum template for starting a new project and `calc_pi` 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 ANL's using MPI examples.

Most of the tutorial focuses on the BLT features used to create the complete `calc_pi` project.

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.

We provide instructions to build and run these projects on several LLNL HPC platforms and ORNL's Summit platform. See *Host-configs*.

4.1.1 Getting Started

BLT is easy to include in your CMake project whether it is an existing project or you are starting from scratch. This tutorial assumes you are using git and the CMake Makefile generator but those commands can easily be changed or ignored.

Include BLT in your Git Repository

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

Add 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 a subdirectory in your repository

This example will clone a copy of 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 CMake 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 "")
```

(continues on next page)

(continued from previous page)

```
    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)
```

This is a robust way of setting up BLT and supports an optional external BLT source directory via the command line option `BLT_SOURCE_DIR`. Using the external BLT source directory allows you to use single BLT instance across multiple independent CMake projects. This also adds helpful error messages if the BLT submodule is missing as well as the commands to solve it.

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: Bare Bones BLT Project

The `bare_bones` example project shows you some of BLT's built-in features. It demonstrates the bare minimum required for testing purposes.

Here is the entire CMakeLists.txt file needed for a bare bones project:

```
cmake_minimum_required(VERSION 3.8)
project( bare_bones )

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

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/bare_bones
cmake .
```

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/bare_bones
mkdir build
cd build
cmake ..
```

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

Note: Smoke tests are designed to show when basic functionality is not working. For example, if you have turned on MPI in your project but the MPI compiler wrapper cannot produce an executable that runs even the most basic MPI code, the `blt_mpi_smoke` test will fail. This helps you know that the problem doesn't lie in your own code but in the building/linking of MPI.

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/bare_bones/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 `bare_bones` 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.

Example files

Files related to setting up the Bare Bones project:

`CMakeLists.txt`

```
#------------------------------------------------------------------------------
# BLT Tutorial Example: Bare Bones Project.
#------------------------------------------------------------------------------

cmake_minimum_required(VERSION 3.8)
project( bare_bones )

# 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
#------------------------------------------------------------------------------

# _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
```

4.1.2 Creating Targets

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 CMake targets 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 then extract that code into a library, which we link into a new executable, `example_2`.

Example 1: Stand-alone Executable

This example is as basic as it gets. After setting up a BLT CMake project, like the Bare Bones project in the previous section, we can start using BLT's macros.

Creating a stand-alone 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. You can create the executable `<build dir>/bin/example_1`, by running the following commands:

```
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 while creating a CMake executable target with the given parameters. It also greatly simplifies the usage of internal and external dependencies. The full list of supported parameters can be found on the *blt_add_executable* API documentation.

Example 2: Executable with a Library

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 extra 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 defaults to building a static library unless you override it with `SHARED` or with the global CMake option `BUILD_SHARED_LIBS`. The full list of supported parameters can be found on the *blt_add_library* API documentation.

4.1.3 Adding Tests

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

Each GoogleTest 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 GoogleTest 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.

GoogleTest (C/C++ Tests)

The contents of a typical GoogleTest 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(...);
}
```

Each unit test is defined by the GoogleTest `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 GoogleTest 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 GoogleTest is initialized before `MPI_Init()` is called.

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

See the GoogleTest Primer for a discussion of GoogleTest 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 GoogleTest 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
!  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
```

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
```

(continues on next page)

(continued from previous page)

```
  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 Targets*), 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)
```

Note: The `COMMAND` parameter can be used to pass arguments into a test executable.

Note: The `NAME` of the test can be different from the test executable, which is passed in through the `COMMAND` parameter.

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

Converting CTest XML to JUnit

It is often useful to convert CTest's XML output to JUnit for various reporting tools such as CI. This is a two step process.

First run your test suite with one of the following commands to output with CTest's XML and to turn off compressed output:

```
make CTEST_OUTPUT_ON_FAILURE=1 test ARGS="--no-compress-output -T Test -VV -j8"
ctest -DCTEST_OUTPUT_ON_FAILURE=1 --no-compress-output -T Test -VV -j8
```

Then convert the CTest XML file to JUnit's format with the XSL file included in BLT. This can be done in many ways, but most Linux or Unix machines come with a program called `xsltproc`

```
cd build-dir
xsltproc -o junit.xml path/to/blt/tests/ctest-to-junit.xsl Testing/*/Test.xml
```

Then point the reporting tool to the outputted `junit.xml` file.

4.1.4 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 specific gcc (version 4.9.3 in this case) on the LLNL Pascal cluster.

```
set(GCC_HOME "/usr/tce")
set(CMAKE_C_COMPILER   "${GCC_HOME}/bin/gcc" CACHE PATH "")
set(CMAKE_CXX_COMPILER "${GCC_HOME}/bin/g++" CACHE PATH "")

# Fortran support
set(ENABLE_FORTRAN ON CACHE BOOL "")
set(CMAKE_Fortran_COMPILER "${GCC_HOME}/bin/gfortran" CACHE PATH "")
```

Building and Testing on Pascal

Since compute nodes on the Pascal cluster have CPUs and GPUs, 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 Pascal:

```
# create build dir
mkdir build
cd build
# configure using host-config
cmake -C ../../host-configs/llnl/toss_3_x86_64_ib/gcc@4.9.3_nvcc.cmake  ..
```

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
```

(continues on next page)

(continued from previous page)

```
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 LLNL BlueOS Ray cluster:

```
# create build dir
mkdir build
cd build
# configure using host-config
cmake -C ../../host-configs/llnl/blueos_3_ppc64le_ib_p9/clang@upstream_nvcc_xlf.cmake␣
↪ ..
```

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
```

Building and Testing on Summit

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 OLCF Summit cluster:

```
# load the cmake module
module load cmake
# create build dir
mkdir build
```

(continues on next page)

(continued from previous page)

```
cd build
# configure using host-config
cmake -C ../../host-configs/olcf/summit/gcc@6.4.0_nvcc.cmake  ..
```

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

```
bash-4.2$ bsub -W 30 -nnodes 1 -P <valid project>  -Is /bin/bash
bash-4.2$ module load gcc cuda
bash-4.2$ make
bash-4.2$ make test

Running tests...
Test project /projects/blt/docs/tutorial/calc_pi/build
      Start  1: test_1
 1/11 Test  #1: test_1 ..........................   Passed    0.00 sec
      Start  2: test_2
 2/11 Test  #2: test_2 ..........................   Passed    1.03 sec
      Start  3: test_3
 3/11 Test  #3: test_3 ..........................   Passed    0.21 sec
      Start  4: blt_gtest_smoke
 4/11 Test  #4: blt_gtest_smoke .................   Passed    0.00 sec
      Start  5: blt_fruit_smoke
 5/11 Test  #5: blt_fruit_smoke .................   Passed    0.00 sec
      Start  6: blt_mpi_smoke
 6/11 Test  #6: blt_mpi_smoke ...................   Passed    0.76 sec
      Start  7: blt_cuda_smoke
 7/11 Test  #7: blt_cuda_smoke ..................   Passed    0.22 sec
      Start  8: blt_cuda_runtime_smoke
 8/11 Test  #8: blt_cuda_runtime_smoke ...........   Passed    0.07 sec
      Start  9: blt_cuda_version_smoke
 9/11 Test  #9: blt_cuda_version_smoke ...........   Passed    0.06 sec
      Start 10: blt_cuda_mpi_smoke
10/11 Test #10: blt_cuda_mpi_smoke ..............   Passed    0.80 sec
      Start 11: blt_cuda_gtest_smoke
11/11 Test #11: blt_cuda_gtest_smoke ............   Passed    0.21 sec

100% tests passed, 0 tests failed out of 11

Total Test time (real) =   3.39 sec
```

Example Host-configs

Basic TOSS3 (for example: Quartz) host-config that has C, C++, and Fortran Compilers along with MPI support:

`gcc@8.3.1 host-config`

```
# Copyright (c) 2017-2021, Lawrence Livermore National Security, LLC and
# other BLT Project Developers. See the top-level LICENSE file for details
#
# SPDX-License-Identifier: (BSD-3-Clause)

#------------------------------------------------------------------------------
# Example host-config file for the quartz cluster at LLNL
#------------------------------------------------------------------------------
#
```

(continues on next page)

(continued from previous page)

```
# This file provides CMake with paths / details for:
#  C,C++, & Fortran compilers + MPI
#
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
# gcc@8.3.1 compilers
#------------------------------------------------------------------------------

set(GCC_VERSION "gcc-8.3.1")
set(GCC_HOME "/usr/tce/packages/gcc/${GCC_VERSION}")

# c compiler
set(CMAKE_C_COMPILER "${GCC_HOME}/bin/gcc" CACHE PATH "")

# cpp compiler
set(CMAKE_CXX_COMPILER "${GCC_HOME}/bin/g++" CACHE PATH "")

# fortran support
set(ENABLE_FORTRAN ON CACHE BOOL "")

# fortran compiler
set(CMAKE_Fortran_COMPILER "${GCC_HOME}/bin/gfortran" CACHE PATH "")

#------------------------------------------------------------------------------
# MPI Support
#------------------------------------------------------------------------------
set(ENABLE_MPI ON CACHE BOOL "")

set(MPI_HOME             "/usr/tce/packages/mvapich2/mvapich2-2.3-${GCC_VERSION}"␣
↪CACHE PATH "")

set(MPI_C_COMPILER       "${MPI_HOME}/bin/mpicc" CACHE PATH "")
set(MPI_CXX_COMPILER     "${MPI_HOME}/bin/mpicxx" CACHE PATH "")
set(MPI_Fortran_COMPILER "${MPI_HOME}/bin/mpif90" CACHE PATH "")

set(MPIEXEC              "/usr/bin/srun" CACHE PATH "")
set(MPIEXEC_NUMPROC_FLAG "-n" CACHE PATH "")
```

Here are the full example host-config files for LLNL's Pascal, Ray, and Quartz Clusters that uses the default compilers on the system:

`gcc@4.9.3 host-config`

```
# Copyright (c) 2017-2021, Lawrence Livermore National Security, LLC and
# other BLT Project Developers. See the top-level LICENSE file for details
#
# SPDX-License-Identifier: (BSD-3-Clause)

#------------------------------------------------------------------------------
# Example host-config file for the Pascal cluster at LLNL
#------------------------------------------------------------------------------
# This file provides CMake with paths / details for:
#  C,C++, & Fortran compilers + MPI & CUDA
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
# gcc@4.9.3 compilers
#------------------------------------------------------------------------------
# _blt_tutorial_compiler_config_start
set(GCC_HOME "/usr/tce")
set(CMAKE_C_COMPILER   "${GCC_HOME}/bin/gcc" CACHE PATH "")
set(CMAKE_CXX_COMPILER "${GCC_HOME}/bin/g++" CACHE PATH "")

# Fortran support
set(ENABLE_FORTRAN ON CACHE BOOL "")
set(CMAKE_Fortran_COMPILER "${GCC_HOME}/bin/gfortran" CACHE PATH "")
# _blt_tutorial_compiler_config_end

#------------------------------------------------------------------------------
# MPI Support
#------------------------------------------------------------------------------
# _blt_tutorial_mpi_config_start
set(ENABLE_MPI ON CACHE BOOL "")

set(MPI_HOME "/usr/tce/packages/mvapich2/mvapich2-2.3-gcc-4.9.3/")
set(MPI_C_COMPILER "${MPI_HOME}/bin/mpicc" CACHE PATH "")

set(MPI_CXX_COMPILER "${MPI_HOME}/bin/mpicxx" CACHE PATH "")

set(MPI_Fortran_COMPILER "${MPI_HOME}/bin/mpif90" CACHE PATH "")
# _blt_tutorial_mpi_config_end

#------------------------------------------------------------------------------
# CUDA support
#------------------------------------------------------------------------------
# _blt_tutorial_cuda_config_start
set(ENABLE_CUDA ON CACHE BOOL "")

set(CUDA_TOOLKIT_ROOT_DIR "/usr/tce/packages/cuda/cuda-10.1.168" CACHE PATH "")
set(CMAKE_CUDA_COMPILER "${CUDA_TOOLKIT_ROOT_DIR}/bin/nvcc" CACHE PATH "")
set(CMAKE_CUDA_HOST_COMPILER "${CMAKE_CXX_COMPILER}" CACHE PATH "")

set(CMAKE_CUDA_ARCHITECTURES "70" CACHE STRING "")
set(_cuda_arch "sm_${CMAKE_CUDA_ARCHITECTURES}")
set(CMAKE_CUDA_FLAGS "-restrict -arch ${_cuda_arch} -std=c++11 --expt-extended-lambda␣
↪-G"
    CACHE STRING "")

set(CUDA_SEPARABLE_COMPILATION ON CACHE BOOL "")

# _blt_tutorial_cuda_config_end
```

More complicated BlueOS host-config that has C, C++, MPI, and CUDA support:

`clang@upstream C++17 host-config`

```
# Copyright (c) 2017-2021, Lawrence Livermore National Security, LLC and
# other BLT Project Developers. See the top-level LICENSE file for details
#
# SPDX-License-Identifier: (BSD-3-Clause)

#------------------------------------------------------------------------------
# Example host-config file for the blue_os cluster at LLNL
```

(continues on next page)

(continued from previous page)

```
#------------------------------------------------------------------------------
#
# This file provides CMake with paths / details for:
#  C/C++:   Clang with GCC 8.3.1 toolchain with C++17 support
#  Cuda
#  MPI
#
#------------------------------------------------------------------------------

#---------------------------------------
# Compilers
#---------------------------------------

set(_CLANG_VERSION "clang-upstream-2019.08.15")
set(_CLANG_DIR "/usr/tce/packages/clang/${_CLANG_VERSION}")
set(_GCC_DIR "/usr/tce/packages/gcc/gcc-8.3.1")

set(CMAKE_C_COMPILER "${_CLANG_DIR}/bin/clang" CACHE PATH "")
set(CMAKE_CXX_COMPILER "${_CLANG_DIR}/bin/clang++" CACHE PATH "")

set(BLT_CXX_STD "c++17" CACHE STRING "")

set(CMAKE_C_FLAGS "--gcc-toolchain=${_GCC_DIR}" CACHE PATH "")
set(CMAKE_CXX_FLAGS "--gcc-toolchain=${_GCC_DIR}" CACHE PATH "")

set(BLT_EXE_LINKER_FLAGS " -Wl,-rpath,${_GCC_DIR}/lib" CACHE PATH "Adds a missing␣
↪libstdc++ rpath")

#---------------------------------------
# MPI
#---------------------------------------
set(ENABLE_MPI ON CACHE BOOL "")

set(_MPI_BASE_DIR "/usr/tce/packages/spectrum-mpi/spectrum-mpi-rolling-release-${_
↪CLANG_VERSION}")

set(MPI_C_COMPILER "${_MPI_BASE_DIR}/bin/mpicc" CACHE PATH "")
set(MPI_CXX_COMPILER "${_MPI_BASE_DIR}/bin/mpicxx" CACHE PATH "")

#------------------------------------------------------------------------------
# Cuda
#------------------------------------------------------------------------------

set(ENABLE_CUDA ON CACHE BOOL "")

set(CUDA_TOOLKIT_ROOT_DIR "/usr/tce/packages/cuda/cuda-11.0.182" CACHE PATH "")

set(CMAKE_CUDA_COMPILER "${CUDA_TOOLKIT_ROOT_DIR}/bin/nvcc" CACHE PATH "")
set(CMAKE_CUDA_HOST_COMPILER "${CMAKE_CXX_COMPILER}" CACHE PATH "")

set(CMAKE_CUDA_ARCHITECTURES "70" CACHE STRING "")
set(_cuda_arch "sm_${CMAKE_CUDA_ARCHITECTURES}")
set(CMAKE_CUDA_FLAGS "-Xcompiler=--gcc-toolchain=${_GCC_DIR} -restrict -arch ${_cuda_
↪arch} -std=c++17 --expt-extended-lambda -G" CACHE STRING "")

# nvcc does not like gtest's 'pthreads' flag
set(gtest_disable_pthreads ON CACHE BOOL "")
```

(continues on next page)

(continued from previous page)

```
set(ENABLE_GTEST_DEATH_TESTS OFF CACHE BOOL "")

# Very specific fix for working around CMake adding implicit link directories␣
↪returned by the BlueOS
# compilers to link CUDA executables
set(BLT_CMAKE_IMPLICIT_LINK_DIRECTORIES_EXCLUDE "/usr/tce/packages/gcc/gcc-4.9.3/
↪lib64/gcc/powerpc64le-unknown-linux-gnu/4.9.3;/usr/tce/packages/gcc/gcc-4.9.3/lib64
↪" CACHE STRING "")
```

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

OSX clang@7.3.0 host-config

```
# Copyright (c) 2017-2021, Lawrence Livermore National Security, LLC and
# other BLT Project Developers. See the top-level LICENSE file for details
#
# SPDX-License-Identifier: (BSD-3-Clause)

##############################################################################
# host-config for naples
##############################################################################

##############################################################################
# Dependencies were built with spack (https://github.com/llnl/spack)
##############################################################################
# spack install  cmake@3.8.2
# spack install  mpich
# spack install  py-sphinx
# spack activate py-sphinx
# spack install  doxygen


##############################################################################
# cmake path
##############################################################################
# /Users/harrison37/Work/blt_tutorial/tpls/spack/opt/spack/darwin-elcapitan-x86_64/
↪clang-7.3.0-apple/cmake-3.8.2-n2i4ijlet37i3jhmjfhzms2wo3b4ybcm/bin/cmake

##############################################################################
# mpi from spack
##############################################################################
set(ENABLE_MPI ON CACHE PATH "")

set(MPI_BASE_DIR "/Users/harrison37/Work/blt_tutorial/tpls/spack/opt/spack/darwin-
↪elcapitan-x86_64/clang-7.3.0-apple/mpich-3.2-yc7ipshe7e3w4ohtgjtms2agecxruavw/bin"␣
↪CACHE PATH "")

set(MPI_C_COMPILER   "${MPI_BASE_DIR}/mpicc" CACHE PATH "")
set(MPI_CXX_COMPILER "${MPI_BASE_DIR}/mpicxx" CACHE PATH "")
set(MPIEXEC          "${MPI_BASE_DIR}/mpiexec" CACHE PATH "")

##############################################################################
# Cuda Support (standard osx cuda toolkit install)
##############################################################################
set(ENABLE_CUDA ON CACHE BOOL "")

set(CUDA_TOOLKIT_ROOT_DIR "/Developer/NVIDIA/CUDA-8.0/" CACHE PATH "")
```

(continues on next page)

(continued from previous page)

```
set(CUDA_BIN_DIR          "/Developer/NVIDIA/CUDA-8.0/bin/" CACHE PATH "")

##############################################################################
# sphinx from spack
##############################################################################
set(SPHINX_EXECUTABLE "/Users/harrison37/Work/blt_tutorial/tpls/spack/opt/spack/
↪darwin-elcapitan-x86_64/clang-7.3.0-apple/python-2.7.13-
↪jmhznopgz2j5zkmuzjygg5oyxnxtc653/bin/sphinx-build" CACHE PATH "")

##############################################################################
# doxygen from spack
##############################################################################
set(DOXYGEN_EXECUTABLE "/Users/harrison37/Work/blt_tutorial/tpls/spack/opt/spack/
↪darwin-elcapitan-x86_64/clang-7.3.0-apple/doxygen-1.8.12-
↪mji43fu4hxuu6js5irshpihkwwucn7rv/bin/doxygen" CACHE PATH "")
```

4.1.5 Importing Targets

One key goal for BLT is to simplify the use of external dependencies and libraries when building your project. To accomplish this, BLT provides a `DEPENDS_ON` option for the *blt_add_library* and *blt_add_executable* macros that supports both your own projects CMake targets and imported targets. We have logically broken this topic into two groups:

Common HPC Dependencies Dependencies such as MPI, CUDA, HIP, and OpenMP, are bundled and ready to use included with BLT as regular named CMake targets. For example, just adding `openmp` to any `DEPENDS_ON` will add the necessary OpenMP compiler and link flags to any target.

Third Party Libraries These are external libraries that your project depend on, such as Lua. They are imported into your project in different ways depending on the level of CMake support provided by that project. BLT provides a macro, *blt_import_library*, which allows you to bundle all necessary information under a single name. Some projects properly export their CMake targets and only need to be imported via a call to `include()`.

Common HPC Dependencies

BLT creates named targets for the common HPC dependencies that most HPC projects need, such as MPI, CUDA, HIP, and OpenMP. Something BLT assists it's users with is getting these dependencies to interoperate within the same library or executable.

As previously mentioned in *Adding Tests*, BLT also provides bundled versions of GoogleTest, GoogleMock, GoogleBenchmark, and FRUIT. Not only are the source for these included, we provide named CMake targets for them as well.

BLT's `mpi`, `cuda`, `cuda_runtime`, `hip`, `hip_runtime`,and `openmp` targets are all defined via the *blt_import_library* macro. This creates a true CMake imported target that is inherited properly through the CMake's dependency graph.

Note: BLT also supports exporting its third-party targets via the `BLT_EXPORT_THIRDPARTY` option. See *Exporting Targets* for more information.

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

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

MPI

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 Pascal Cluster:

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

set(MPI_HOME "/usr/tce/packages/mvapich2/mvapich2-2.3-gcc-4.9.3/")
set(MPI_C_COMPILER "${MPI_HOME}/bin/mpicc" CACHE PATH "")

set(MPI_CXX_COMPILER "${MPI_HOME}/bin/mpicxx" CACHE PATH "")

set(MPI_Fortran_COMPILER "${MPI_HOME}/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 *Adding Tests*, GoogleTest 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 GoogleTest.

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

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

    MPI_Init(&argc, &argv);
```

(continues on next page)

(continued from previous page)

```
    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

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 Pascal Cluster:

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

set(CUDA_TOOLKIT_ROOT_DIR "/usr/tce/packages/cuda/cuda-10.1.168" CACHE PATH "")
set(CMAKE_CUDA_COMPILER "${CUDA_TOOLKIT_ROOT_DIR}/bin/nvcc" CACHE PATH "")
set(CMAKE_CUDA_HOST_COMPILER "${CMAKE_CXX_COMPILER}" CACHE PATH "")

set(CMAKE_CUDA_ARCHITECTURES "70" CACHE STRING "")
set(_cuda_arch "sm_${CMAKE_CUDA_ARCHITECTURES}")
set(CMAKE_CUDA_FLAGS "-restrict -arch ${_cuda_arch} -std=c++11 --expt-extended-lambda␣
↪-G"
    CACHE STRING "")

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

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

```
    blt_add_library( NAME       calc_pi_cuda
                     HEADERS    calc_pi_cuda.hpp calc_pi_cuda_exports.h
                     SOURCES    calc_pi_cuda.cpp
```

(continues on next page)

(continued from previous page)

```
                    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/C++/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:

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

set(CUDA_TOOLKIT_ROOT_DIR "/usr/tce/packages/cuda/cuda-10.1.168" CACHE PATH "")
set(CMAKE_CUDA_COMPILER "${CUDA_TOOLKIT_ROOT_DIR}/bin/nvcc" CACHE PATH "")
set(CMAKE_CUDA_HOST_COMPILER "${MPI_CXX_COMPILER}" CACHE PATH "")

set(CMAKE_CUDA_ARCHITECTURES "70" CACHE STRING "")
set(_cuda_arch "sm_${CMAKE_CUDA_ARCHITECTURES}")
set(CMAKE_CUDA_FLAGS "-restrict -arch ${_cuda_arch} -std=c++11 --expt-extended-lambda␣
↪-G" CACHE STRING "")

set(CUDA_SEPARABLE_COMPILATION ON CACHE BOOL "" )

# nvcc does not like gtest's 'pthreads' flag
set(gtest_disable_pthreads 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)
```

HIP

HIP tutorial coming soon!

BLT also supports AMD's HIP via a mechanism very similar to our CUDA support.

Important Setup Variables

- ENABLE_HIP : Enables HIP support in BLT
- HIP_ROOT_DIR : Root directory for HIP installation

BLT Targets

- hip : Adds include directories, hip runtime libraries, and compiles source with hipcc
- hip_runtime : Adds include directories and hip runtime libraries

Third Party Libraries

Third party libraries come in three flavors based on the CMake support provided by the project and the CMake community as a whole: no Cmake support, CMake's Find* modules, and First Class project support.

No CMake Support

Some libraries have no support for easily importing their CMake targets into external projects, either through properly exporting their targets themselves or the CMake community has not written a Find module that eases this work.

BLT provides a *blt_import_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.

We will use Lua as an example of this because up until recently (CMake version 3.18), there was no Find module. This provides us a great example on how to show two ways of importing the same library's targets.

The following example shows how to find a library, Lua this time, manually. By first, searching for the include directories and then for the library itself. Finally it calls *blt_import_library* to bundle that information under one easy to remember name, lua :

```
# first Check for LUA_DIR
if (NOT EXISTS "${LUA_DIR}")
    message(FATAL_ERROR "Given LUA_DIR does not exist: ${LUA_DIR}")
endif()

if (NOT IS_DIRECTORY "${LUA_DIR}")
    message(FATAL_ERROR "Given LUA_DIR is not a directory: ${LUA_DIR}")
```

(continues on next page)

(continued from previous page)

```
endif()

# Find includes directory
find_path( LUA_INCLUDE_DIR lua.hpp
           PATHS  ${LUA_DIR}/include/
                  ${LUA_DIR}/include/lua
           NO_DEFAULT_PATH
           NO_CMAKE_ENVIRONMENT_PATH
           NO_CMAKE_PATH
           NO_SYSTEM_ENVIRONMENT_PATH
           NO_CMAKE_SYSTEM_PATH)

# Find libraries
find_library( LUA_LIBRARY NAMES lua liblua
              PATHS ${LUA_DIR}/lib
              NO_DEFAULT_PATH
              NO_CMAKE_ENVIRONMENT_PATH
              NO_CMAKE_PATH
              NO_SYSTEM_ENVIRONMENT_PATH
              NO_CMAKE_SYSTEM_PATH)


include(FindPackageHandleStandardArgs)
# handle the QUIETLY and REQUIRED arguments and set LUA_FOUND to TRUE
# if all listed variables are TRUE
find_package_handle_standard_args(LUA  DEFAULT_MSG
                                  LUA_INCLUDE_DIR
                                  LUA_LIBRARY )

if(NOT LUA_FOUND)
    message(FATAL_ERROR "LUA_DIR is not a path to a valid Lua install")
endif()

message(STATUS "Lua Includes: ${LUA_INCLUDE_DIR}")
message(STATUS "Lua Libraries: ${LUA_LIBRARY}")

blt_import_library(NAME       lua
                   TREAT_INCLUDES_AS_SYSTEM ON
                   DEFINES    HAVE_LUA=1
                   INCLUDES   ${LUA_INCLUDE_DIR}
                   LIBRARIES  ${LUA_LIBRARY}
                   EXPORTABLE ON)
```

Then `lua` is available to be used in the `DEPENDS_ON` list in the following *blt_add_executable* or *blt_add_library* calls, or in any CMake command that accepts a target.

Note: CMake targets created by *blt_import_library* are `INTERFACE` libraries that can be installed and exported if the `EXPORTABLE` option is enabled. For example, if the `calc_pi` project depends on Lua, it could export its `lua` target. To avoid introducing target name conflicts for users of the `calc_pi` project who might also create a target called `lua`, `lua` should be exported as `calc_pi\:\:lua` .

Note: Because CMake targets are only accessible from within the directory they were defined (including subdirectories), the `include()` command should be preferred to the `add_subdirectory()` command for adding CMake files that create imported library targets needed in other directories. The `GLOBAL` option to *blt_import_library* can

also be used to manage visibility.

Cmake's Find Modules

This time we will do exactly the same thing but using the new CMake provided `FindLua.cmake` module. Instead of calling having to ensure correctness and calling `find_path` and `find_library`, we only have to call `find_package` and it handles this for us. Each Find module outputs differently named variables so it is important to read the documentation on CMake's website. This is where *blt_import_library* shines because you only have to figure those variables once then use the new imported library's `NAME` in the rest of your project.

```
# FindLua.cmake takes in LUA_DIR as an environment variable, which is the directory
# where Lua was installed to and fills variables: LUA_FOUND, LUA_LIBRARIES, and LUA_
↪INCLUDE_DIR
set(ENV{LUA_DIR} ${LUA_DIR})

find_package(Lua)

if (NOT LUA_FOUND)
    MESSAGE(FATAL_ERROR "Could not find Lua in the provided LUA_DIR: ${LUA_DIR}")
endif()

blt_import_library(NAME       lua
                   TREAT_INCLUDES_AS_SYSTEM ON
                   DEFINES    HAVE_LUA=1
                   INCLUDES   ${LUA_INCLUDE_DIR}
                   LIBRARIES  ${LUA_LIBRARIES}
                   EXPORTABLE ON)
```

First Class Project Support

Some projects provide what we call First Class support. They have gone through the effort of properly exporting all necessary targets to use their project and install the necessary configuration files inside of their install directory, usually something like `<install dir>\lib\cmake\<Project Name>Config.cmake`.

LLNL's Axom project exports all targets that can be easily imported into your project with a single CMake function call:

```
# use the provided PATH directory and create a cmake target named 'axom'
find_package(axom REQUIRED)
```

You can then add the created CMake target, `axom`, to any `DEPENDS_ON` list or use any other regular CMake function to change it.

4.1.6 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
```

(continues on next page)

(continued from previous page)

```
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 your local build directory will contain the following for viewing:

- Sphinx: `build-calc-pi/docs/sphinx/html/index.html`
- Doxygen: `build-calc-pi/docs/doxygen/html/index.html`

4.1.7 Advanced Topics

This section includes subpages on advanced topics such as:

- *Portable Compiler Flags* : Managing compiler flags across compiler families
- *Exporting Targets* : Exporting your project's CMake Targets for others to use
- *Object Libraries* : A collection of object files that are not linked or archived into a library

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: GNU does not have a way to only disable warnings about OpenMP pragmas, so you must disable warnings about all unknown pragmas on this compiler.

Exporting Targets

BLT provides several built-in targets for commonly used libraries:

`mpi` Available when `ENABLE_MPI` is `ON`

`openmp` Available when `ENABLE_OPENMP` is `ON`

`cuda` and `cuda_runtime` Available when `ENABLE_CUDA` is `ON`

`hip` and `hip_runtime` Available when `ENABLE_HIP` is `ON`

These targets can be made exportable in order to make them available to users of your project via CMake's `install()` command. Setting BLT's `BLT_EXPORT_THIRDPARTY` option to `ON` will mark all active targets in the above list as `EXPORTABLE` (see the *blt_import_library* API documentation for more info).

Note: As with other `EXPORTABLE` targets created by *blt_import_library*, these targets should be prefixed with the name of the project. Either the `EXPORT_NAME` target property or the `NAMESPACE` option to CMake's `install` command can be used to modify the name of an installed target.

Note: If a target in your project is added to an export set, any of its dependencies marked `EXPORTABLE` must be added to the same export set. Failure to add them will result in a CMake error in the exporting project.

Typical usage of the `BLT_EXPORT_THIRDPARTY` option is as follows:

```
# BLT configuration - enable MPI
set(ENABLE_MPI ON CACHE BOOL "")
# and mark the subsequently created MPI target as exportable
set(BLT_EXPORT_THIRDPARTY ON CACHE BOOL "")
# Both of the above must happen before SetupBLT.cmake is included
include(/path/to/SetupBLT.cmake)

# Later, a project might mark a target as dependent on MPI
blt_add_executable( NAME    example_1
                    SOURCES example_1.cpp
                    DEPENDS_ON mpi )

# Add the example_1 target to the example-targets export set
install(TARGETS example_1 EXPORT example-targets)

# Add the MPI target to the same export set - this is required
# because the mpi target was marked exportable
install(TARGETS mpi EXPORT example-targets)
```

To avoid collisions with projects that import "example-targets", there are two options for adjusting the exported name of the `mpi` target.

The first is to rename only the `mpi` target's exported name:

```
set_target_properties(mpi PROPERTIES EXPORT_NAME example::mpi)
install(EXPORT example-targets)
```

With this approach the `example_1` target's exported name is unchanged - a project that imports the `example-targets` export set will have `example_1` and `example::mpi` targets made available. The imported `example_1` will depend on `example::mpi`.

Another approach is to install all targets in the export set behind a namespace:

```
install(EXPORT example-targets NAMESPACE example::)
```

With this approach all targets in the export set are prefixed, so an importing project will have `example::example_1` and `example::mpi` targets made available. The imported `example::example_1` will depend on `example::mpi`.

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.

If you are using separable CUDA compilation (relocatable device code) in your object library, users of that library will be required to use NVCC to link their executables - in general, only NVCC can perform the "device link" step. To remove this restriction, you can enable the `CUDA_RESOLVE_DEVICE_SYMBOLS` property on an object library:

```
set_target_properties(myObjectLibrary PROPERTIES CUDA_RESOLVE_DEVICE_SYMBOLS ON)
```

To enable this device linking step for all libraries in your project (including object libraries), you can set the `CMAKE_CUDA_RESOLVE_DEVICE_SYMBOLS` option to `ON`. This defaults the `CUDA_RESOLVE_DEVICE_SYMBOLS` target property to `ON` for all targets created by BLT.

You can read more about this property in the CMake documentation.

Note: These options only apply when an object library in your project is linked later into a shared or static library, in which case a separate object file containing device symbols is created and added to the "final" library. Object libraries provided directly to users of your project will still require a device link step.

The `CUDA_RESOLVE_DEVICE_SYMBOLS` property is also supported for static and shared libraries. By default, it is enabled for shared libraries but disabled for static libraries.

4.1.8 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.

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.

Error at Configure Time for Third-party Dependency Problems

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
)
```

4.2 API Documentation

4.2.1 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**

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

blt_add_executable

```
blt_add_executable( NAME        <name>
                    SOURCES     [source1 [source2 ...]]
                    HEADERS     [header1 [header2 ...]]
                    INCLUDES    [dir1 [dir2 ...]]
                    DEFINES     [define1 [define2 ...]]
                    DEPENDS_ON  [dep1 [dep2 ...]]
                    OUTPUT_DIR  [dir]
                    OUTPUT_NAME [name]
                    FOLDER      [name])
```

Adds an executable target to the 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 target depends on

OUTPUT_DIR Directory that this target will built to, defaults to bin

OUTPUT_NAME Override built file name of the executable (defaults to `<name>`)

FOLDER Name of the IDE folder to ease organization

Adds an executable target, called `<name>`, to be built from the given sources. It also adds the given `INCLUDES` and `DEFINES` from the parameters to this macro and adds all inherited information from the list given by `DEPENDS_ON`. This macro creates a true CMake target that can be altered by other CMake commands like normal, such as `set_target_property()`. It also adds `SOURCES` and `HEADERS` to the library for build system dependency tracking and IDE folder support.

`OUTPUT_NAME` is useful when multiple CMake targets with the same name need to be created by different targets.

Note: 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.

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_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]
```

(continues on next page)

(continued from previous page)

```
                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 the 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 creates a true CMake target that can be altered by other CMake commands like normal, such as `set_target_property()`. It also adds `SOURCES` and `HEADERS` to the library for build system dependency tracking and IDE folder support.

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 instantiate 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 `HEADERS` argument and do not specify `SOURCES` (because there aren't any). Header-only libraries can have dependencies like compiled libraries. These will be propagated to targets that depend on the header-only library.

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: Due to necessary record keeping, BLT Object libraries need to be defined by *blt_add_library* before they are used in any `DEPENDS_ON` list. They also do not follow CMake's normal transitivity rules. This is 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 appropriately 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_patch_target

```
blt_patch_target( 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 ...]] )
```

Modifies the properties of an existing target. `PUBLIC` visibility is used unless the target is an `INTERFACE` library, in which case `INTERFACE` visibility is used.

NAME Name of the CMake target to patch

DEPENDS_ON List of CMake targets that this target depends on

INCLUDES List of include directories to be inherited by dependent targets

TREAT_INCLUDES_AS_SYSTEM Whether to inform the compiler to treat this target's include paths as system headers - this applies to all include paths for the target, not just those specifies in the `INCLUDES` parameter. 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`.

FORTRAN_MODULES Fortran module directories to be inherited by dependent targets

LIBRARIES List of CMake targets and library files (.a/.so/.lib/.dll) that make up this target, used for libraries

COMPILE_FLAGS List of compiler flags to be inherited by dependent targets

LINK_FLAGS List of linker flags to be inherited by dependent targets

DEFINES List of compiler defines to be inherited by dependent targets

This macro does not create a target, it is intended to be used with CMake targets created via another BLT macro or CMake command. Unlike *blt_register_library*, it modifies the specified target, updating the CMake properties of the target that correspond to each of the parameters.

> **Warning:** The `DEPENDS_ON` and `LIBRARIES` parameters cannot be used when patching a target declared in a separate directory unless CMake policy CMP0079 has been set.

blt_import_library

```
blt_import_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 ...]]
                    GLOBAL                     [ON|OFF]
                    EXPORTABLE                 [ON|OFF])
```

Creates a CMake target from build artifacts and system files generated outside of this build system.

NAME Name of the created CMake target

DEPENDS_ON List of CMake targets that this library depends on

INCLUDES List of include directories to be inherited by dependent targets

TREAT_INCLUDES_AS_SYSTEM Whether to inform the compiler to treat this library's include paths as system headers

FORTRAN_MODULES Fortran module directories to be inherited by dependent targets

LIBRARIES List of CMake targets and library files (.a/.so/.lib/.dll) that make up this library

COMPILE_FLAGS List of compiler flags to be inherited by dependent targets

LINK_FLAGS List of linker flags to be inherited by dependent targets

DEFINES List of compiler defines to be inherited by dependent targets

GLOBAL Whether to extend the visibility of the created library to global scope

EXPORTABLE Whether the created target should be exportable and `install`-able

Allows libraries not built with CMake to be imported as native CMake targets in order to take full advantage of CMake's transitive dependency resolution.

For example, a `Find<library>.cmake` may set only the variables `<library>_LIBRARIES` (which might contain the .a/.so/.lib/.dll file for the library itself, and the libraries it depends on) and `<library>_INCLUDES` (which might contain the include directories required to use the library). Instead of using these variables directly every time they are needed, they could instead be built into a CMake target. It also allows for compiler and linker options to be associated with the library.

As with BLT-registered libraries, it can be added to the `DEPENDS_ON` parameter when building another target or to `target_link_libraries()` to transitively add in all includes, libraries, flags, and definitions associated with the imported library.

The `EXPORTABLE` option is intended to be used to simplify the process of exporting a project. Instead of handwriting package location logic in a CMake package configuration file, the `EXPORTABLE` targets can be exported with the targets defined by the project.

Note: Libraries marked `EXPORTABLE` cannot also be marked `GLOBAL`. They also must be added to any export set that includes a target that depends on the `EXPORTABLE` library.

Note: It is highly recommended that `EXPORTABLE` imported targets be installed with a project-specific namespace/prefix, either with the `NAMESPACE` option of CMake's `install()` command, or the `EXPORT_NAME` target

property. This mitigates the risk of conflicting target names.

In CMake terms, the imported libraries will be `INTERFACE` libraries.

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

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.

Note: In general, this macro should be avoided unless absolutely necessary, as it does not create a native CMake target. If the library to register already exists as a CMake target, consider using *blt_patch_target*. Otherwise, consider using *blt_import_library*. These options are insufficient in some circumstances, for example, if it is necessary to add libraries to a CMake library target declared in another directory while keeping the modified target usable with the same name as the original target. In this case *blt_register_library* is the only option.

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

Internal variable names are prefixed with _ to avoid collision with input parameters.

4.2.2 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*, *blt_import_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.

4.2.3 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
5  blt_assert_exists( FILES ${PROJECT_SOURCE_DIR}/cmake/blt/SetupBLT.cmake )
6
7  ## checks can also be bundled in one call
8  blt_assert_exists( DIRECTORIES ${PROJECT_SOURCE_DIR}/cmake/blt
9                     FILES ${PROJECT_SOURCE_DIR}/cmake/blt/SetupBLT.cmake )
10
```

(continues on next page)

(continued from previous page)

```
## 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 )
```

4.2.4 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>
                  [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}" )
```

4.2.5 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>
                     YAPF_CFG_FILE         <Path to Yapf config file>
                     CMAKEFORMAT_CFG_FILE  <Path to CMakeFormat config file>
                     CPPCHECK_FLAGS        <List of flags added to Cppcheck>
                     CLANGQUERY_CHECKER_DIRECTORIES [dir1 [dir2]])
```

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

YAPF_CFG_FILE Path to Yapf config file

CMAKEFORMAT_CFG_FILE Path to CMakeFormat config file

CPPCHECK_FLAGS List of flags added to Cppcheck

CLANGQUERY_CHECKER_DIRECTORIES List of directories where clang-query's checkers are located

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, BLT_Python_FILE_EXTS, BLT_CMAKE_FILE_EXTS, and BLT_Fortran_FILE_EXTS. Currently this macro only has code checks for C/C++ and Python; it simply filters out the Fortran files.

This macro supports C/C++ 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.

This macro also supports Python code formatting with Yapf only if the following requirements are met:

- YAPF_CFG_FILE is given
- YAPF_EXECUTABLE is defined and found prior to calling this macro

This macro also supports CMake code formatting with CMakeFormat only if the following requirements are met:

- CMAKEFORMAT_CFG_FILE is given
- CMAKEFORMAT_EXECUTABLE is defined and found prior to calling this macro

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>`.

If a particular version of a code formatting tool is required, you can configure BLT to enforce that version by setting `BLT_REQUIRED_<CLANGFORMAT|ASTYLE|UNCRUSTIFY|YAPF|CMAKEFORMAT>_VERSION` to as much of the version as you need. For example:

```
# If astyle major version 3 is required (3.0, 3.1, etc are acceptable)
set(BLT_REQUIRED_ASTYLE_VERSION "3")
# Or, if exactly 3.1 is needed
set(BLT_REQUIRED_ASTYLE_VERSION "3.1")
```

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
  - CLANGQUERY_CHECKER_DIRECTORIES parameter given or BLT_CLANGQUERY_CHECKER_DIRECTORIES is defined
- clang-tidy

  - CLANGTIDY_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|clang_query|clang_tidy>_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 ...]]
                            CHECKER_DIRECTORIES [dir1 [dir2]])
```

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

CHECKER_DIRECTORIES List of directories where clang-query's checkers are located

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.

A list of checker directories is required for clang-query, this can be defined either by the parameter CHECKER_DIRECTORIES or the variable BLT_CLANGQUERY_CHECKER_DIRECTORIES.

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_clang_tidy_target

```
blt_add_clang_tidy_target( NAME              <Created Target Name>
                           WORKING_DIRECTORY <Working Directory>
                           COMMENT           <Additional Comment for Target␣
↪Invocation>
                           CHECKS            <If specified, enables a specific set of␣
↪checks>
                           FIX               <TRUE | FALSE (default)>
                           SRC_FILES         [source1 [source2 ...]] )
```

Creates a new build target for running clang-tidy.

NAME Name of created build target

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

COMMENT Comment prepended to the build target output

CHECKS List of checks to be run on the selected source files, available checks are listed here.

FIX Applies fixes for checks (a subset of clang-tidy checks specify how they should be resolved)

SRC_FILES Source list that clang-tidy will be ran on

Clang-tidy is a tool used for diagnosing and fixing typical programming errors. It is useful for enforcing coding standards and rules on your source code. Clang-tidy is documented here.

`CHECKS` are the static analysis "rules" to specifically run on the target. If no checks are specified, clang-tidy will run the default available static analysis checks.

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.

blt_add_yapf_target

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

Creates a new build target for running Yapf

NAME Name of created build target

MODIFY_FILES Modify the files in place. Defaults to FALSE.

CFG_FILE Path to Yapf config file

PREPEND_FLAGS Additional flags added to the front of the Yapf flags

APPEND_FLAGS Additional flags added to the end of the Yapf flags

COMMENT Comment prepended to the build target output

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

SRC_FILES Source list that Yapf will be ran on

Yapf is a Source Code Beautifier for Python code. More information about Yapf 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.

blt_add_cmakeformat_target

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

Creates a new build target for running CMakeFormat

NAME Name of created build target

MODIFY_FILES Modify the files in place. Defaults to FALSE.

CFG_FILE Path to CMakeFormat config file

PREPEND_FLAGS Additional flags added to the front of the CMakeFormat flags

APPEND_FLAGS Additional flags added to the end of the CMakeFormat flags

COMMENT Comment prepended to the build target output

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

SRC_FILES Source list that CMakeFormat will be ran on

CMakeFormat is a Source Code Beautifier for CMake code. More information about CMakeFormat 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.

4.2.6 Code Metric Macros

blt_add_code_coverage_target

```
blt_add_code_coverage_target( NAME                <Created Target Name>
                              RUNNER              <The command to run the tests>
                              SOURCE_DIRECTORIES [dir1 [dir2 ...]] )
```

Creates a new build target for generating a code coverage report.

NAME Name of created build target

RUNNER The command used to run the tests, e.g., `make test`

SOURCE_DIRECTORIES The directories containing the source code whose test coverage is to be evaluated

Code coverage is the degree to which the tests for a piece of software "cover" functions and/or individual lines of code. It can be used to identify gaps in testing, namely, code that is not tested. GCC's `gcov` tool is used to generate the coverage data, and its accompanying `lcov` tool is used to generate an HTML report containing coverage percentage information and highlighted source code that indicates which code was or was not executed as part of the test suite.

Note: Coverage analysis is only supported by GNU/Clang compilers.

This functionality requires that BLT's `ENABLE_COVERAGE` option is enabled and that `gcov`, `lcov`, and `genhtml` are present on your system. To use a specific version of one of these tools, you can set `GCOV_EXECUTABLE`, `LCOV_EXECUTABLE`, and `GENHTML_EXECUTABLE` to point at the desired version(s).

Note: The `ENABLE_COVERAGE` option will add compiler flags that instrument your code (and slow it down). The option should never be enabled by default in a project for performance reasons.

4.2.7 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. Requires that a CMake variable named `SPHINX_EXECUTABLE` contains the path to the `sphinx-build` executable.

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>`.

4.3 Developer Guide

This section contains information for BLT developers.

4.3.1 Release Process

Note: No significant code development is performed on a release branch. In addition to preparing release notes and other documentation, the only code changes that should be done are bug fixes identified during release preparations

Here are the steps to follow when creating a BLT release.

1: Start Release Candidate Branch

Create a release candidate branch off of the develop branch to initiate a release. The name of a release branch must contain the associated release version number. Typically, we use a name like v0.4.0-rc (i.e., version 0.4.0 release candidate).

```
git checkout -b v0.4.0-rc
```

2: Update Versions in Code

Update BLT_VERSION

- `SetupBLT.cmake`: `set(BLT_VERSION "0.4.0" CACHE STRING "")`

Update Release Notes

1. Update `RELEASE-NOTES.md` by changing the `unreleased` section from:

```
## [Unreleased] - Release date yyyy-mm-dd
```

Also add to a versioned section with the current date while leaving the unreleased section:

```
## [Unreleased] - Release date yyyy-mm-dd

## [Version 0.4.0] - Release date 2021-04-09
```

Finally, add a link to the bottom as well:

```
[Unreleased]:    https://github.com/LLNL/blt/compare/v0.3.6...develop
```

to:

```
[Unreleased]:    https://github.com/LLNL/blt/compare/v0.4.0...develop
[Version 0.4.0]:    https://github.com/LLNL/blt/compare/v0.3.6...v0.4.0
```

3: Create Pull Request and push a git *tag* for the release

1. Commit the changes and push them to Github.
2. Create a pull request from release candidate branch to `main` branch.
3. Merge pull request after reviewed and passing tests.
4. Checkout main locally: `git checkout main && git pull`
5. Create release tag: `git tag v0.4.0`
6. Push tag to Github: `git push --tags`

4: Draft a Github Release

Draft a new Release on Github

1. Enter the desired tag version, e.g., *v0.4.0*
2. Select **main** as the target branch to tag a release.
3. Enter a Release title with the same as the tag *v0.4.0*
4. Copy and paste the information for the release from the `RELEASE-NOTES.md` into the release description (omit any sections if empty).
5. Publish the release. This will add a corresponding entry in the Releases section

Note: Github will add a corresponding tarball and zip archives consisting of the source files for each release.

5: Create Release Branch and Mergeback to develop

1. Create a branch off main that is for the release branch.

```
git pull
git checkout main
git checkout -b release-v0.4.0
git push --set-upstream origin release-v0.4.0
```

2. Create a pull request to merge `main` into `develop` through Github. When approved, merge it.

7: Build Release Documentation

Enable the build on readthedocs version page for the version branch created in step 5.