Generic TriBITS Project, Build, Test, and Install Reference Guide

Author: Roscoe A. Bartlett
Contact: bartlett.roscoe@gmail.com
Date: 2018-09-18
Version: tribits_start-2138-g6df1d30

Abstract: This document is generated from the generic template body document `TribitsBuildReferenceBody.rst` and provides a general project-independent reference on how to configure, build, test, and install a project that uses the TriBITS CMake build system. The primary audience of this particular build of this document are TriBITS project developers themselves. A project-specific version of this document should be created and accessed by users of a particular TriBITS-based project.

Contents

1 Introduction

This document is created using the script `create-build-ref.sh` in this directory which just runs:

```
$ ./create-project-build-ref.py \
  --project-name="<Project>" \
  --project-template-file=TribitsBuildReferenceTemplate.rst \
  --file-base=TribitsBuildReference
```

In a project-specific version, `<Project>` is replaced with the actual project name (e.g. `Trilinos`). This version of the generated document is referred to by the general TribitsDeveloperGuide.[rst,html,pdf] document.

Below are given genetic versions of the sections that show up in every project-specific build of this document.

2 Getting set up to use CMake

Before one can configure <Project> to be built, one must first obtain a version of CMake on the system newer than 2.8.11 This guide assumes that once CMake is installed that it will be in the default path with the name `cmake`.

2.1 Installing a binary release of CMake [casual users]

Download and install the binary (version 2.8.11 or greater is recommended) from:

http://www.cmake.org/cmake/resources/software.html

2.2 Installing CMake from source [developers and experienced users]

If you have access to the <Project> git repositories (which which includes a snapshot of TriBITS), then install CMake with:

```
$ cd <some-scratch-space>/
$ export TRIBITS_BASE_DIR=<project-base-dir>/cmake/tribits
$ $TRIBITS_BASE_DIR/devtools_install/install-cmake.py \
   --install-dir=<INSTALL_BASE_DIR> \
   --do-all
```

This will result in cmake and related CMake tools being installed in `<INSTALL_BASE_DIR>/bin/` (see the instructions printed at the end on how to update your `PATH` env var).

To get help for installing CMake with this script use:

```
$ $TRIBITS_BASE_DIR/devtools_install/install-cmake.py --help
```

NOTE: you will want to read the help message about how to use sudo to install in a privileged location (like the default `/usr/local/bin`).

2.3 Installing Ninja from Source

The Ninja tool allows for much faster parallel builds for some large CMake projects and performs much faster dependency analysis than the Makefiles backend build system. It also provides some other nice features like `ninja -n -d explain` to show why the build system decides to (re)build the targets that it decides to build.

The Kitware fork of Ninja at:

https://github.com/Kitware/ninja/releases

provides releases of Ninja that allows CMake 3.7.0+ to build Fortran code with Ninja. For example, the Kitware Ninja release `1.7.2.git.kitware.dyndep-1` works with Fortran.

Ninja is easy to install from source. It is a simple `configure --prefix=<dir>`, `make` and `make install`.

3 Getting CMake Help

3.1 Finding CMake help at the website

http://www.cmake.org

3.2 Building CMake help locally

To get help on CMake input options, run:

```
$ cmake --help
```

To get help on a single CMake function, run:

```
$ cmake --help-command <command>
```

To generate the entire documentation at once, run:

```
$ cmake --help-full cmake.help.html
```

(Open your web browser to the file cmake.help.html)

4 Configuring (Makefile, Ninja and other Generators)

CMake supports a number of different build generators (e.g. Ninja, Eclipse, XCode, MS Visual Studio, etc.) but the primary generator most people use on Unix/Linux system is `make` (using the default cmake option `-G"Unix Makefiles"`) and CMake generated Makefiles. Another (increasingly) popular generator is Ninja (using cmake option `-GNinja`). Most of the material in this section applies to all generators but most experience is for the Makefiles and Ninja generators.

4.1 Setting up a build directory

In order to configure, one must set up a build directory. <Project> does **not** support in-source builds so the build tree must be separate from the source tree. The build tree can be created under the source tree such as with:

```
$ cd <src-dir>/
$ mkdir <build-dir>
$ cd <build-dir>/
```

but it is generally recommended to create a build directory parallel from the source tree such as with:

```
<some-base-dir>/
  <src-dir>/
  <build-dir>/
```

NOTE: If you mistakenly try to configure for an in-source build (e.g. with 'cmake .') you will get an error message and instructions on how to resolve the problem by deleting the generated CMakeCache.txt file (and other generated files) and then follow directions on how to create a different build directory as shown above.

4.2 Basic configuration

A few different approaches for configuring are given below but likely the most recommended one for complex environments is to use `*.cmake` fragment files passed in through the <Project>_CONFIGURE_OPTIONS_FILE option.

a) Create a 'do-configure' script such as [Recommended]:

```
EXTRA_ARGS=$@

cmake \
  -D CMAKE_BUILD_TYPE=DEBUG \
  -D <Project>_ENABLE_TESTS=ON \
  $EXTRA_ARGS \
  ${SOURCE_BASE}
```

and then run it with:

```
./do-configure [OTHER OPTIONS] -D<Project>_ENABLE_<TRIBITS_PACKAGE>=ON
```

where `<TRIBITS_PACKAGE>` is a valid SE Package name (see above), etc. and `SOURCE_BASE` is set to the <Project> source base directory (or your can just give it explicitly in the script).

See `<Project>/sampleScripts/*` for examples of real `do-configure` scripts for different platforms.

NOTE: If one has already configured once and one needs to configure from scratch (needs to wipe clean defaults for cache variables, updates compilers, other types of changes) then one will want to delete the local CMakeCache.txt and other CMake-generated files before configuring again (see Reconfiguring completely from scratch).

b) Create a CMake file fragment and point to it [Recommended].

Create a do-configure script like:

```
EXTRA_ARGS=$@

cmake \
  -D <Project>_CONFIGURE_OPTIONS_FILE=MyConfigureOptions.cmake \
  -D <Project>_ENABLE_TESTS=ON \
  $EXTRA_ARGS \
  ${SOURCE_BASE}
```

where MyConfigureOptions.cmake (in the current working directory) might look like:

```
SET(CMAKE_BUILD_TYPE DEBUG CACHE STRING "Set in MyConfigureOptions.cmake")
SET(<Project>_ENABLE_CHECKED_STL ON CACHE BOOL "Set in MyConfigureOptions.cmake")
SET(BUILD_SHARED_LIBS ON CACHE BOOL "Set in MyConfigureOptions.cmake")
...
```

Using a configuration fragment `*.cmake` file allows for better reuse of configure options across different configure scripts and better version control of configure options. Using the comment `"Set in MyConfigureOptions.cmake"` makes it easy see where that variable got set when looking an the generated `CMakeCache.txt` file. Also, when this `*.cmake` fragment file changes, CMake will automatically trigger a reconfgure during a make (because it knows about the file and will check its time stamp, unlike when using `-C <file-name>.cmake`, see below).

One can use the `FORCE` option in the `SET()` commands shown above and that will override any value of the options that might already be set. However, that will not allow the user to override the options on the CMake command-line using `-D<VAR>=<value>` so it is generally **not** desired to use `FORCE`.

One can also pass in a list of configuration fragment files separated by commas `','` which will be read in the order they are given as:

```
-D <Project>_CONFIGURE_OPTIONS_FILE=<file0>.cmake,<file1>.cmake,...
```

One can read in configure option files under the project source directory by using the type `STRING` such as with:

```
-D <Project>_CONFIGURE_OPTIONS_FILE:STRING=cmake/MpiConfig1.cmake
```

In this case, the relative paths will be with respect to the project base source directory, not the current working directory (unlike when using `-C <file-name>.cmake`, see below). (By specifying the type `STRING`, one turns off CMake interpretation as a `FILEPATH`. Otherwise, the type `FILEPATH` causes CMake to always interpret relative paths with respect to the current working directory and set the absolute path).

Note that CMake options files can also be read in using the built-in CMake argument `-C <file>.cmake` as:

```
cmake -C <file0>.cmake -C <file1>.cmake ... [other options] \
  ${SOURCE_BASE}
```

However, there are some differences to using `<Project>_CONFIGURE_OPTIONS_FILE` vs. `-C` to read in `*.cmake` files to be aware of as described below:

1) One can use `-D<Project>_CONFIGURE_OPTIONS_FILE:STRING=<rel-path>/<file-name>.cmake` with a relative path w.r.t. to the source tree to make it easier

to point to options files in the project source. Using `cmake -C <abs-path>/<file-name>.cmake` would require having to give the absolute path `<abs-path>` or a longer relative path from the build directory back to the source directory. Having to give the absolute path to files in the source tree complicates configure scripts in some cases (i.e. where the project source directory location may not be known or easy to get).

2) When configuration files are read in using `<Project>_CONFIGURE_OPTIONS_FILE`, the will get reprocessed on every reconfigure (such as when reconfigure happens automatically when running `make`). That means that if options change in those included `*.cmake` files from the initial configure, then those updated options will get automatically picked up in a reconfigure. But when processing `*.cmake` files using the built-in `-C <file-name>.cmake` argument, updated options will not get set. Therefore, if one wants to have the `*.cmake` files automatically be reprocessed, then one should use `<Project>_CONFIGURE_OPTIONS_FILE`. But if one does not want to have the contents of the `*.cmake` file reread on reconfigures, then one would want to use `-C`.

3) One can create and use parametrized `*.cmake` files that can be used with multiple TriBITS projects. For example, one can have set statements like `SET(${PROJECT_NAME}_ENABLE_Fortran OFF ...)` since `PROJECT_NAME` is known before the file is included. One can't do that with `cmake -C` and instead would have to the full variables names specific for a given project.

4) However, the `*.cmake` files specified by `<Project>_CONFIGURE_OPTIONS_FILE` will only get read in **after** the project's `ProjectName.cmake` and other `SET()` statements are called at the top of the project's top-level `CMakeLists.txt.‘ file. So any CMake cache variables that are set in this early CMake code will override cache defaults set in the included ‘‘*.cmake` file. (This is why TriBITS projects must be careful **not** to set default values for cache variables directly like this but instead should set indirect `<Project>_<VarName>_DEFAULT` non-cache variables.) But when a `*.cmake` file is read in using `-C`, then the `SET()` statements in those files will get processed before any in the project's `CMakeLists.txt` file. So be careful about this difference in behavior and carefully watch cache variable values actually set in the generated `CMakeCache.txt` file.

c) Using the QT CMake configuration GUI:

On systems where the QT CMake GUI is installed (e.g. Windows) the CMake GUI can be a nice way to configure <Project> (or just explore options) if you are a user. To make your configuration easily repeatable, you might want to create a fragment file and just load it by setting <Project>_CONFIGURE_OPTIONS_FILE (see above) in the GUI.

4.3 Selecting the list of packages to enable

The <Project> project is broken up into a set of packages that can be enabled (or disabled). For details and generic examples, see Package Dependencies and Enable/Disable Logic and TriBITS Dependency Handling Behaviors.

See the following use cases:

- Determine the list of packages that can be enabled
- Print package dependencies
- Enable a set of packages
- Enable or disable tests for specific packages
- Enable to test all effects of changing a given package(s)
- Enable all packages with tests and examples
- Disable a package and all its dependencies
- Remove all package enables in the cache

4.3.1 Determine the list of packages that can be enabled

In order to see the list of available <Project> SE Packages to enable, just run a basic CMake configure, enabling nothing, and then grep the output to see what packages are available to enable. The full set of defined packages is contained the lines starting with `'Final set of enabled SE packages'` and `'Final set of non-enabled SE packages'`. If no SE packages are enabled by default (which is base behavior), the full list of packages will be listed on the line `'Final set of non-enabled SE packages'`. Therefore, to see the full list of defined packages, run:

```
./do-configure 2>&1 | grep "Final set of .*enabled SE packages"
```

Any of the packages shown on those lines can potentially be enabled using `-D <Project>_ENABLE_<TRIBITS_PACKAGE>=ON` (unless they are set to disabled for some reason, see the CMake output for package disable warnings).

Another way to see the full list of SE packages that can be enabled is to configure with <Project>_DUMP_PACKAGE_DEPENDENCIES = `ON` and then grep for `<Project>_SE_PACKAGES` using, for example:

```
./do-configure 2>&1 | grep "<Project>_SE_PACKAGES: "
```

4.3.2 Print package dependencies

The set of package dependencies can be printed in the `cmake` STDOUT by setting the configure option:

```
-D <Project>_DUMP_PACKAGE_DEPENDENCIES=ON
```

This will print the basic forward/upstream dependencies for each SE package. To find this output, look for the line:

```
Printing package dependencies ...
```

and the dependencies are listed below this for each SE package in the form:

```
-- <PKG>_LIB_REQUIRED_DEP_TPLS: <TPL0> <TPL1> ...
-- <PKG>_LIB_OPTIONAL_DEP_TPLS: <TPL2> <TPL3> ...
-- <PKG>_LIB_REQUIRED_DEP_PACKAGES: <PKG0> <[PKG1> ...
-- <PKG>_LIB_OPTIONAL_DEP_PACKAGES: <PKG2> <PKG3> ...
-- <PKG>_TEST_REQUIRED_DEP_TPLS: <TPL4> <TPL5> ...
-- <PKG>_TEST_OPTIONAL_DEP_TPLS: <TPL6> <TPL7> ...
-- <PKG>_TEST_REQUIRED_DEP_PACKAGES: <PKG4> <[PKG5> ...
-- <PKG>_TEST_OPTIONAL_DEP_PACKAGES: <PKG6> <PKG7> ...
```

(Dependencies that don't exist are left out of the output. For example, if there are no `<PKG>_LIB_OPTIONAL_DEP_PACKAGES` dependencies, then that line is not printed.)

To also see the direct forward/downstream dependencies for each SE package, also include:

```
-D <Project>_DUMP_FORWARD_PACKAGE_DEPENDENCIES=ON
```

These dependencies are printed along with the backward/upstsream dependencies as described above.

Both of these variables are automatically enabled when <Project>_VERBOSE_CONFIGURE = `ON`.

4.3.3 Enable a set of packages

To enable an SE package `<TRIBITS_PACKAGE>` (and optionally also its tests and examples), configure with:

```
-D <Project>_ENABLE_<TRIBITS_PACKAGE>=ON \
-D <Project>_ENABLE_ALL_OPTIONAL_PACKAGES=ON \
-D <Project>_ENABLE_TESTS=ON \
```

This set of arguments allows a user to turn on `<TRIBITS_PACKAGE>` as well as all packages that `<TRIBITS_PACKAGE>` can use. All of the package's optional "can use" upstream dependent packages are enabled with `-D<Project>_ENABLE_ALL_OPTIONAL_PACKAGES=ON`. However, `-D<Project>_ENABLE_TESTS=ON` will only enable tests and examples for `<TRIBITS_PACKAGE>` (or any other packages specifically enabled).

If a TriBITS package `<TRIBITS_PACKAGE>` has subpackages (e.g. `<A>`, `<B>`, etc.), then enabling the package is equivalent to setting:

```
-D <Project>_ENABLE_<TRIBITS_PACKAGE><A>=ON \
-D <Project>_ENABLE_<TRIBITS_PACKAGE><B>=ON \
 ...
```

However, a TriBITS subpackage will only be enabled if it is not already disabled either explicitly or implicitly.

NOTE: The CMake cache variable type for all `XXX_ENABLE_YYY` variables is actually `STRING` and not `BOOL`. That is because these enable variables take on the string enum values of `"ON"`, `"OFF"`, end empty `""`. An empty enable means that the TriBITS dependency system is allowed to decide if an enable should be turned on or off based on various logic. The CMake GUI will enforce the values of `"ON"`, `"OFF"`, and empty `""` but it will not enforce this if you set the value

on the command line or in a SET() statement in an input `*.cmake` options files. However, setting `-DXXX_ENABLE_YYY=TRUE` and `-DXXX_ENABLE_YYY=FALSE` is allowed and will be interpreted correctly..

4.3.4 Enable or disable tests for specific packages

The enable tests for explicitly enabled packages, configure with:

```
-D <Project>_ENABLE_<TRIBITS_PACKAGE_1>=ON \
-D <Project>_ENABLE_<TRIBITS_PACKAGE_2>=ON \
-D <Project>_ENABLE_TESTS=ON \
```

This wil result in the enable of the test suites for any package that explicitly enabled with `-D <Project>_ENABLE_<TRIBITS_PACKAGE>=ON`. Note that his will **not** result in the enable of the test suites for any packages that may only be implicitly enabled in order to build the explicitly enabled packages.

If one wants to enable a package along with the enable of other packages, but not the test suite for that package, then when can disable the tests for that package by configuring with:

```
-D <Project>_ENABLE_<TRIBITS_PACKAGE_1>=ON \
-D <Project>_ENABLE_<TRIBITS_PACKAGE_2>=ON \
-D <Project>_ENABLE_<TRIBITS_PACKAGE_3>=ON \
-D <Project>_ENABLE_TESTS=ON \
-D <TRIBITS_PACKAGE_2>_ENABLE_TESTS=OFF \
```

The above will enable the package test suites for `<TRIBITS_PACKGE_1>` and `<TRIBITS_PACKGE_3>` but **not** for `<TRIBITS_PACKAGE_2>` (or any other packages that might get implicitly enabled). One might use this if one wants to build and install package `<TRIBITS_PACKAGE_2>` but does not want to build and run the test suite for that package.

These and other options give the user complete control of what packages get enabled or disabled and what package test suites are enabled or disabled.

4.3.5 Enable to test all effects of changing a given package(s)

To enable an SE package `<TRIBITS_PACKAGE>` to test it and all of its downstream packages, configure with:

```
-D <Project>_ENABLE_<TRIBITS_PACKAGE>=ON \
-D <Project>_ENABLE_ALL_FORWARD_DEP_PACKAGES=ON \
-D <Project>_ENABLE_TESTS=ON \
```

The above set of arguments will result in package `<TRIBITS_PACKAGE>` and all packages that depend on `<TRIBITS_PACKAGE>` to be enabled and have all of their tests turned on. Tests will not be enabled in packages that do not depend on `<TRIBITS_PACKAGE>` in this case. This speeds up and robustifies pre-push testing.

4.3.6 Enable all packages with tests and examples

To enable all SE packages (and optionally also their tests and examples), add the configure options:

```
-D <Project>_ENABLE_ALL_PACKAGES=ON \
-D <Project>_ENABLE_TESTS=ON \
```

Specific packages can be disabled with `<Project>_ENABLE_<TRIBITS_PACKAGE>=OFF`. This will also disable all packages that depend on `<TRIBITS_PACKAGE>`.

All examples are also enabled by default when setting `<Project>_ENABLE_TESTS=ON`.

By default, setting `<Project>_ENABLE_ALL_PACKAGES=ON` only enables primary tested (PT) code. To have this also enable all secondary tested (ST) code, one must also set `<Project>_ENABLE_SECONDARY_TESTED_CODE=ON`.

NOTE: If the project is a "meta-project", then `<Project>_ENABLE_ALL_PACKAGES=ON` may not enable *all* the SE packages but only the project's primary meta-project packages. See Package Dependencies and Enable/Disable Logic and TriBITS Dependency Handling Behaviors for details.

4.3.7 Disable a package and all its dependencies

To disable an SE package and all of the packages that depend on it, add the configure options:

```
-D <Project>_ENABLE_<TRIBITS_PACKAGE>=OFF
```

For example:

```
-D <Project>_ENABLE_<PACKAGE_A>=ON \
-D <Project>_ENABLE_ALL_OPTIONAL_PACKAGES=ON \
-D <Project>_ENABLE_<PACKAGE_B>=ON \
```

will enable `<PACKAGE_A>` and all of the packages that it depends on except for `<PACKAGE_B>` and all of its forward dependencies.

If a TriBITS package `<TRIBITS_PACKAGE>` has subpackages (e.g. `<A>`, `<B>`, etc.), then disabling the package is equivalent to setting:

```
-D <Project>_ENABLE_<TRIBITS_PACKAGE><A>=OFF \
-D <Project>_ENABLE_<TRIBITS_PACKAGE><B>=OFF \
...
```

The disable of the subpackage is this case will override any enables.

If a disabled package is a required dependency of some explicitly enabled downstream package, then the configure will error out if `<Project>_DISABLE_ENABLED_FORWARD_DEP_PACKAG` Otherwise, a WARNING will be printed and the downstream package will be disabled and configuration will continue.

4.3.8 Remove all package enables in the cache

To wipe the set of package enables in the `CMakeCache.txt` file so they can be reset again from scratch, configure with:

```
$ ./-do-confiugre -D <Project>_UNENABLE_ENABLED_PACKAGES=TRUE
```

This option will set to empty " all package enables, leaving all other cache variables as they are. You can then reconfigure with a new set of package enables for a different set of packages. This allows you to avoid more expensive configure time checks and to preserve other cache variables that you have set and don't want to loose. For example, one would want to do this to avoid compiler and TPL checks.

4.4 Selecting compiler and linker options

The compilers for C, C++, and Fortran will be found by default by CMake if they are not otherwise specified as described below (see standard CMake documentation for how default compilers are found). The most direct way to set the compilers are to set the CMake cache variables:

```
-D CMAKE_<LANG>_COMPILER=<path-to-compiler>
```

The path to the compiler can be just a name of the compiler (e.g. `-DCMAKE_C_COMPILER=gcc`) or can be an absolute path (e.g. `-DCMAKE_C_COMPILER=/usr/local/bin/cc`). The safest and more direct approach to determine the compilers is to set the absolute paths using, for example, the cache variables:

```
-D CMAKE_C_COMPILER=/opt/my_install/bin/gcc \
-D CMAKE_CXX_COMPILER=/opt/my_install/bin/g++ \
-D CMAKE_Fortran_COMPILER=/opt/my_install/bin/gfortran
```

or if `TPL_ENABLE_MPI=ON` (see Configuring with MPI support) something like:

```
-D CMAKE_C_COMPILER=/opt/my_install/bin/mpicc \
-D CMAKE_CXX_COMPILER=/opt/my_install/bin/mpicxx \
-D CMAKE_Fortran_COMPILER=/opt/my_install/bin/mpif90
```

If these the CMake cache variables are not set, then CMake will use the compilers specified in the environment variables `CC`, `CXX`, and `FC` for C, C++ and Fortran, respectively. If one needs to drill down through different layers of scripts, then it can be useful to set the compilers using these environment variables. But in general is it recommended to be explicit and use the above CMake cache variables to set the absolute path to the compilers to remove all ambiguity.

If absolute paths to the compilers are not specified using the CMake cache variables or the environment variables as described above, then in MPI mode (i.e. `TPL_ENABLE_MPI=ON`) TriBITS performs its own search for the MPI compiler wrappers that will find the correct compilers for most MPI distributions (see Configuring with MPI support). However, if in serial mode (i.e. `TPL_ENABLE_MPI=OFF`), then CMake will do its own default compiler search. The algorithm by which raw CMake finds these compilers is not precisely documented (and seems to change based on the platform). However, on Linux systems, the observed algorithm appears to be:

1. Search for the C compiler first by looking in `PATH` (or the equivalent on Windows), starting with a compiler with the name `cc` and then moving on to other names like `gcc`, etc. This first compiler found is set to `CMAKE_C_COMPILER`.

2. Search for the C++ compiler with names like `c++`, `g++`, etc., but restrict the search to the same directory specified by base path to the C compiler given in the variable `CMAKE_C_COMPILER`. The first compiler that is found is set to `CMAKE_CXX_COMPILER`.

3. Search for the Fortran compiler with names like `f90`, `gfortran`, etc., but restrict the search to the same directory specified by base path

to the C compiler given in the variable CMAKE_C_COMPILER. The first compiler that is found is set to CMAKE_CXX_COMPILER.

WARNING: While this build-in CMake compiler search algorithm may seems reasonable, it fails to find the correct compilers in many cases for a non-MPI serial build. For example, if a newer version of GCC is installed and is put first in PATH, then CMake will fail to find the updated gcc compiler and will instead find the default system cc compiler (usually under /usr/bin/cc on Linux may systems) and will then only look for the C++ and Fortran compilers under that directory. This will fail to find the correct updated compilers because GCC does not install a C compiler named cc! Therefore, if you want to use the default CMake compiler search to find the updated GCC compilers, you can set the CMake cache variable:

```
-D CMAKE_C_COMPILER=gcc
```

or can set the environment variable CC=gcc. Either one of these will result in CMake finding the updated GCC compilers found first in PATH.

Once one has specified the compilers, one can also set the compiler flags, but the way that CMake does this is a little surprising to many people. But the <Project> TriBITS CMake build system offers the ability to tweak the built-in CMake approach for setting compiler flags. First some background is in order. When CMake creates the object file build command for a given source file, it passes in flags to the compiler in the order:

```
${CMAKE_<LANG>_FLAGS}  ${CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>}
```

where <LANG> = C, CXX, or Fortran and <CMAKE_BUILD_TYPE> = DEBUG or RELEASE. Note that the options in CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> come after and override those in CMAKE_<LANG>_FLAGS! The flags in CMAKE_<LANG>_FLAGS apply to all build types. Optimization, debug, and other build-type-specific flags are set in CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>. CMake automatically provides a default set of debug and release optimization flags for CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> (e.g. CMAKE_CXX_FLAGS_DEBUG is typically "-g -O0" while CMAKE_CXX_FLAGS_RELEASE is typically "-O3"). This means that if you try to set the optimization level with -DCMAKE_CXX_FLAGS="-O4", then this level gets overridden by the flags specified in CMAKE_<LANG>_FLAGS_BUILD or CMAKE_<LANG>_FLAGS_RELEASE.

Note that TriBITS will set defaults for CMAKE_<LANG>_FLAGS and CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TY which may be different that what raw CMake would set. TriBITS provides a means for project and package developers and users to set and override these compiler flag variables globally and on a package-by-package basis. Below, the facilities for manipulating compiler flags is described.

Also, to see that the full set of compiler flags one has to actually build a target with, for example make VERBOSE=1 (see Building with verbose output without reconfiguring). One can not just look at the cache variables for CMAKE_<LANG>_FLAGS and CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> in the file CMakeCache.txt. These get overwritten and redefined by TriBITS in development as described below (see Overriding CMAKE_BUILD_TYPE debug/release compiler options).

The <Project> TriBITS CMake build system will set up default compile flags for GCC ('GNU') in development mode (i.e. <Project>_ENABLE_DEVELOPMENT_MODE=ON)

on order to help produce portable code. These flags set up strong warning options and enforce language standards. In release mode (i.e. <Project>_ENABLE_DEVELOPMENT_MODE=ON), these flags are not set. These flags get set internally into the variables CMAKE_<LANG>_FLAGS (when processing packages, not at the global cache variable level) but the user can append flags that override these as described below.

4.4.1 Configuring to build with default debug or release compiler flags

To build a debug version, pass into 'cmake':

```
-D CMAKE_BUILD_TYPE=DEBUG
```

This will result in debug flags getting passed to the compiler according to what is set in CMAKE_<LANG>_FLAGS_DEBUG.

To build a release (optimized) version, pass into 'cmake':

```
-D CMAKE_BUILD_TYPE=RELEASE
```

This will result in optimized flags getting passed to the compiler according to what is in CMAKE_<LANG>_FLAGS_RELEASE.

The default build type is typically CMAKE_BUILD_TYPE=RELEASE unless -D USE_XSDK_DEFAULTS=TRUE is set in which case the default build type is CMAKE_BUILD_TYPE=DEBUG as per the xSDK configure standard.

4.4.2 Adding arbitrary compiler flags but keeping default build-type flags

To append arbitrary compiler flags to CMAKE_<LANG>_FLAGS (which may be set internally by TriBITS) that apply to all build types, configure with:

```
-D CMAKE_<LANG>_FLAGS="<EXTRA_COMPILER_OPTIONS>"
```

where <EXTRA_COMPILER_OPTIONS> are your extra compiler options like "-DSOME_MACRO_TO_DEFINE -funroll-loops". These options will get appended to (i.e. come after) other internally defined compiler option and therefore override them. The options are then pass to the compiler in the order:

```
<DEFAULT_TRIBITS_LANG_FLAGS> <EXTRA_COMPILER_OPTIONS> \
  ${CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>}
```

This that setting CMAKE_<LANG>_FLAGS can override the default flags that TriBITS will set for CMAKE_<LANG>_FLAGS but will **not** override flags specified in CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>.

Instead of directly setting the CMake cache variables CMAKE_<LANG>_FLAGS one can instead set environment variables CFLAGS, CXXFLAGS and FFLAGS for CMAKE_C_FLAGS, CMAKE_CXX_FLAGS and CMAKE_Fortran_FLAGS, respectively.

In addition, if -DUSE_XSDK_DEFAULTS=TRUE is set, then one can also pass in Fortran flags using the environment variable FCFLAGS (raw CMake does not recognize FCFLAGS). But if FFLAGS and FCFLAGS are both set, then they must be the same or a configure error will occur.

Options can also be targeted to a specific TriBITS package using:

```
-D <TRIBITS_PACKAGE>_<LANG>_FLAGS="<EXTRA_COMPILER_OPTIONS>"
```

The package-specific options get appended to those already in CMAKE_<LANG>_FLAGS and therefore override (but not replace) those set globally in CMAKE_<LANG>_FLAGS (either internally or by the user in the cache).

NOTES:

1) Setting CMAKE_<LANG>_FLAGS will override but will not replace any other internally set flags in CMAKE_<LANG>_FLAGS defined by the <Project> CMake system because these flags will come after those set internally. To get rid of these project/TriBITS default flags, see below.

2) Given that CMake passes in flags in CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> after those in CMAKE_<LANG>_FLAGS means that users setting the CMAKE_<LANG>_FLAGS and <TRIBITS_PACKAGE>_<LANG>_FLAGS will **not** override the flags in CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_T which come after on the compile line. Therefore, setting CMAKE_<LANG>_FLAGS and <TRIBITS_PACKAGE>_<LANG>_FLAGS should only be used for options that will not get overridden by the debug or release compiler flags in CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>. However, setting CMAKE_<LANG>_FLAGS will work well for adding extra compiler defines (e.g. -DSOMETHING) for example.

WARNING: Any options that you set through the cache variable CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TY will get overridden in the <Project> CMake system for GNU compilers in development mode so don't try to manually set CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> directly! To override those options, see CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>_OVERRIDE below.

4.4.3 Overriding CMAKE_BUILD_TYPE debug/release compiler options

To override the default CMake-set options in CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>, use:

```
-D CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>_OVERRIDE="<OPTIONS_TO_OVERRIDE>"
```

For example, to default debug options use:

```
-D CMAKE_C_FLAGS_DEBUG_OVERRIDE="-g -O1" \
-D CMAKE_CXX_FLAGS_DEBUG_OVERRIDE="-g -O1"
-D CMAKE_Fortran_FLAGS_DEBUG_OVERRIDE="-g -O1"
```

and to override default release options use:

```
-D CMAKE_C_FLAGS_RELEASE_OVERRIDE="-O3 -funroll-loops" \
-D CMAKE_CXX_FLAGS_RELEASE_OVERRIDE="-O3 -funroll-loops"
-D CMAKE_Fortran_FLAGS_RELEASE_OVERRIDE="-O3 -funroll-loops"
```

NOTES: The TriBITS CMake cache variable CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>_OVERRIDE is used and not CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> because is given a default internally by CMake and the new varaible is needed to make the override explicit.

4.4.4 Appending arbitrary libraries and link flags every executable

In order to append any set of arbitrary libraries and link flags to your executables use:

```
-D<Project>_EXTRA_LINK_FLAGS="<EXTRA_LINK_LIBRARIES>" \
-DCMAKE_EXE_LINKER_FLAGS="<EXTRA_LINK_FLAGG>"
```

Above, you can pass any type of library and they will always be the last libraries listed, even after all of the TPLs.

NOTE: This is how you must set extra libraries like Fortran libraries and MPI libraries (when using raw compilers). Please only use this variable as a last resort.

NOTE: You must only pass in libraries in `<Project>_EXTRA_LINK_FLAGS` and *not* arbitrary linker flags. To pass in extra linker flags that are not libraries, use the built-in CMake variable `CMAKE_EXE_LINKER_FLAGS` instead. The TriBITS variable `<Project>_EXTRA_LINK_FLAGS` is badly named in this respect but the name remains due to backward compatibility requirements.

4.4.5 Turning off strong warnings for individual packages

To turn off strong warnings (for all languages) for a given TriBITS package, set:

```
-D <TRIBITS_PACKAGE>_DISABLE_STRONG_WARNINGS=ON
```

This will only affect the compilation of the sources for `<TRIBITS_PACKAGES>`, not warnings generated from the header files in downstream packages or client code.

Note that strong warnings are only enabled by default in development mode (`<Project>_ENABLE_DEVELOPMENT_MODE==ON`) but not release mode (`<Project>_ENABLE_DEVELOPMENT_MODE`
A release of <Project> should therefore not have strong warning options enabled.

4.4.6 Overriding all (strong warnings and debug/release) compiler options

To override all compiler options, including both strong warning options and debug/release options, configure with:

```
-D CMAKE_C_FLAGS="-O3 -funroll-loops" \
-D CMAKE_CXX_FLAGS="-O3 -fexceptions" \
-D CMAKE_BUILD_TYPE=NONE \
-D <Project>_ENABLE_STRONG_C_COMPILE_WARNINGS=OFF \
-D <Project>_ENABLE_STRONG_CXX_COMPILE_WARNINGS=OFF \
-D <Project>_ENABLE_SHADOW_WARNINGS=OFF \
-D <Project>_ENABLE_COVERAGE_TESTING=OFF \
-D <Project>_ENABLE_CHECKED_STL=OFF \
```

NOTE: Options like `<Project>_ENABLE_SHADOW_WARNINGS`, `<Project>_ENABLE_COVERAGE_TESTING`, and `<Project>_ENABLE_CHECKED_STL` do not need to be turned off by default but they are shown above to make it clear what other CMake cache variables can add compiler and link arguments.

NOTE: By setting `CMAKE_BUILD_TYPE=NONE`, then `CMAKE_<LANG>_FLAGS_NONE` will be empty and therefore the options set in `CMAKE_<LANG>_FLAGS` will be all that is passed in.

4.4.7 Enable and disable shadowing warnings for all <Project> packages

To enable shadowing warnings for all <Project> packages (that don't already have them turned on) then use:

```
-D <Project>_ENABLE_SHADOW_WARNINGS=ON
```

To disable shadowing warnings for all <Project> packages (even those that have them turned on by default) then use:

```
-D <Project>_ENABLE_SHADOW_WARNINGS=OFF
```

NOTE: The default value is empty " which lets each <Project> package decide for itself if shadowing warnings will be turned on or off for that package.

4.4.8 Removing warnings as errors for CLEANED packages

To remove the `-Werror` flag (or some other flag that is set) from being applied to compile CLEANED packages (like the Trilinos package Teuchos), set the following when configuring:

```
-D <Project>_WARNINGS_AS_ERRORS_FLAGS=""
```

4.4.9 Adding debug symbols to the build

To get the compiler to add debug symbols to the build, configure with:

```
-D <Project>_ENABLE_DEBUG_SYMBOLS=ON
```

This will add `-g` on most compilers. NOTE: One does **not** generally need to create a fully debug build to get debug symbols on most compilers.

4.5 Enabling support for Ninja

The Ninja build tool can be used as the back-end build tool instead of Makefiles by adding:

```
-GNinja
```

to the CMake configure line (the default on most Linux and OSX platforms is `-G"Unix Makefiles"`). This instructs CMake to create the back-end `ninja` build files instead of back-end Makefiles (see Building (Ninja generator)).

In addition, for versions of CMake 3.7.0+, the TriBITS build system will, by default, generate Makefiles in every binary directory where there is a CMakeLists.txt file in the source tree. These Makefiles have targets scoped to that subdirectory that use `ninja` to build targets in that subdirectory just like with the native CMake recursive `-G"Unix Makefiles"` generator. This allows one to `cd` into any binary directory and type `make` to build just the targets in that directory. These TriBITS-generated Ninja makefiles also support `help` and `help-objects` targets making it easy to build individual executables, libraries and object files in any binary subdirectory.

WARNING: Using `make -j<N>` with these TriBITS-generated Ninja Makefiles will **not** result in using `<N>` processes to build in parallel and will instead use **all** of the free cores to build on the machine! To control the number of processes used, run `make NP=<N>` instead! See Building in parallel with Ninja.

The generation of these Ninja makefiles can be disabled by setting:

```
-D<Project>_WRITE_NINJA_MAKEFILES=OFF
```

(But these Ninja Makefiles get created very quickly even for a very large CMake project so there is usually little reason to not generate them.) Trying to set `-D<Project>_WRITE_NINJA_MAKEFILES=ON` for versions of CMake older than 3.7.0 will not work since features were added to CMake 3.7.0+ that allow for the generation of these makefiles.

4.6 Enabling support for C++11

To enable support for C++11 in packages that support C++11 (either optionally or required), configure with:

```
-D <Project>_ENABLE_CXX11=ON
```

By default, the system will try to automatically find compiler flags that will enable C++11 features. If it finds flags that allow a test C++11 program to compile, then it will an additional set of configure-time tests to see if several C++11 features are actually supported by the configured C++ compiler and support will be disabled if all of these features are not supported.

In order to pre-set and/or override the C++11 compiler flags used, set the cache variable:

```
-D <Project>_CXX11_FLAGS="<compiler flags>"
```

In order to enable C++11 but not have the default system set any flags for C++11, use:

```
-D <Project>_ENABLE_CXX11=ON
-D <Project>_CXX11_FLAGS=" "
```

The empty space " " will result in the system assuming that no flags needs to be set.

4.7 Enabling explicit template instantiation for C++

To enable explicit template instantiation for C++ code for packages that support it, configure with:

```
-D <Project>_ENABLE_EXPLICIT_INSTANTIATION=ON
```

When `OFF`, all packages that have templated C++ code will use implicit template instantiation.

Explicit template instantiation can be enabled (`ON`) or disabled (`OFF`) for individual packages with:

```
-D <TRIBITS_PACKAGE>_ENABLE_EXPLICIT_INSTANTIATION=[ON|OFF]
```

The default value for `<TRIBITS_PACKAGE>_ENABLE_EXPLICIT_INSTANTIATION` is set by `<Project>_ENABLE_EXPLICIT_INSTANTIATION`.

For packages that support it, explicit template instantation can massively reduce the compile times for the C++ code involved. To see what packages support explicit instantation just search the CMakeCache.txt file for variables with `ENABLE_EXPLICIT_INSTANTIATION` in the name.

4.8 Disabling the Fortran compiler and all Fortran code

To disable the Fortran compiler and all <Project> code that depends on Fortran set:

```
-D <Project>_ENABLE_Fortran=OFF
```

NOTE: The Fortran compiler may be disabled automatically by default on systems like MS Windows.

NOTE: Most Apple Macs do not come with a compatible Fortran compiler by default so you must turn off Fortran if you don't have a compatible Fortran compiler.

4.9 Enabling runtime debug checking

a) Enabling <Project> ifdefed runtime debug checking:

To turn on optional ifdefed runtime debug checking, configure with:

```
-D <Project>_ENABLE_DEBUG=ON
```

This will result in a number of ifdefs to be enabled that will perform a number of runtime checks. Nearly all of the debug checks in <Project> will get turned on by default by setting this option. This option can be set independent of `CMAKE_BUILD_TYPE` (which sets the compiler debug/release options).

NOTES:

- The variable `CMAKE_BUILD_TYPE` controls what compiler options are passed to the compiler by default while `<Project>_ENABLE_DEBUG` controls what defines are set in config.h files that control ifdefed debug checks.
- Setting `-DCMAKE_BUILD_TYPE=DEBUG` will automatically set the default `<Project>_ENABLE_DEBUG=ON`.

b) Enabling checked STL implementation:

To turn on the checked STL implementation set:

```
-D <Project>_ENABLE_CHECKED_STL=ON
```

NOTES:

- By default, this will set -D_GLIBCXX_DEBUG as a compile option for all C++ code. This only works with GCC currently.
- This option is disabled by default because to enable it by default can cause runtime segfaults when linked against C++ code that was compiled without -D_GLIBCXX_DEBUG.

4.10 Configuring with MPI support

To enable MPI support you must minimally set:

```
-D TPL_ENABLE_MPI=ON
```

There is built-in logic to try to find the various MPI components on your system but you can override (or make suggestions) with:

```
-D MPI_BASE_DIR="path"
```

(Base path of a standard MPI installation which has the subdirs 'bin', 'libs', 'include' etc.)
or:

```
-D MPI_BIN_DIR="path1;path2;...;pathn"
```

which sets the paths where the MPI executables (e.g. mpiCC, mpicc, mpirun, mpiexec) can be found. By default this is set to `${MPI_BASE_DIR}/bin` if `MPI_BASE_DIR` is set.

NOTE: TriBITS uses the MPI compiler wrappers (e.g. mpiCC, mpicc, mpic++, mpif90, etc.) which is more standard with other builds systems for HPC computing using MPI (and the way that MPI implementations were meant to be used). But directly using the MPI compiler wrappers as the direct compilers is inconsistent with the way that the standard CMake module `FindMPI.cmake` which tries to "unwrap" the compiler wrappers and grab out the raw underlying compilers and the raw compiler and linker command-line arguments. In this way, TriBITS is more consistent with standard usage in the HPC community but is less consistent with CMake (see "HISTORICAL NOTE" below).

There are several different different variations for configuring with MPI support:

a) **Configuring build using MPI compiler wrappers:**

The MPI compiler wrappers are turned on by default. There is built-in logic in TriBITS that will try to find the right MPI compiler wrappers. However, you can specifically select them by setting, for example:

```
-D MPI_C_COMPILER:FILEPATH=mpicc \
-D MPI_CXX_COMPILER:FILEPATH=mpic++ \
-D MPI_Fortan_COMPILER:FILEPATH=mpif77
```

which gives the name of the MPI C/C++/Fortran compiler wrapper executable. In this case, just the names of the programs are given and absolute path of the executables will be searched for under `${MPI_BIN_DIR}/` if the cache variable `MPI_BIN_DIR` is set, or in the default path otherwise. The found programs will then be used to set the cache variables `CMAKE_[C,CXX,Fortran]_COMPILER`.

One can avoid the search and just use the absolute paths with, for example:

```
-D MPI_C_COMPILER:FILEPATH=/opt/mpich/bin/mpicc \
-D MPI_CXX_COMPILER:FILEPATH=/opt/mpich/bin/mpic++ \
-D MPI_Fortan_COMPILER:FILEPATH=/opt/mpich/bin/mpif77
```

However, you can also directly set the variables `CMAKE_[C,CXX,Fortran]_COMPILER` with, for example:

```
-D CMAKE_C_COMPILER:FILEPATH=/opt/mpich/bin/mpicc \
-D CMAKE_CXX_COMPILER:FILEPATH=/opt/mpich/bin/mpic++ \
-D CMAKE_Fortan_COMPILER:FILEPATH=/opt/mpich/bin/mpif77
```

WARNING: If you set just the compiler names and not the absolute paths with `CMAKE_<LANG>_COMPILER` in MPI mode, then a search will not be done and these will be expected to be in the path at build time. (Note that his is inconsistent the behavior of raw CMake in non-MPI mode described in Selecting compiler and linker options). If both `CMAKE_<LANG>_COMPILER` and `MPI_<LANG>_COMPILER` are set, however, then `CMAKE_<LANG>_COMPILER` will be used and `MPI_<LANG>_COMPILER` will be ignored.

Note that when `USE_XSDK_DEFAULTS=FALSE` (see xSDK Configuration Options), then the environment variables `CC`, `CXX` and `FC` are ignored. But when `USE_XSDK_DEFAULTS=TRUE` and the CMake cache variables `CMAKE_[C,CXX,Fortran]_COMPILER` are not set, then the environment variables `CC`, `CXX` and `FC` will be used for `CMAKE_[C,CXX,Fortran]_COMPILER`, even if the CMake cache variables `MPI_[C,CXX,Fortran]_COMPILER` are set! So if one wants to make sure and set the MPI compilers irrespective of the xSDK mode, then one should set cmake cache variables `CMAKE_[C,CXX,Fortran]_COMPILER` to the absolute path of the MPI compiler wrappers.

HISTORICAL NOTE: The TriBITS system has its own custom MPI integration support and does not (currently) use the standard CMake module `FindMPI.cmake`. This custom support for MPI was added to TriBITS in 2008 when it was found the built-in `FindMPI.cmake` module was not sufficient for the needs of Trilinos and the approach taken by the module (still in use as of CMake 3.4.x) which tries to unwrap the raw compilers and grab the list of include directories, link libraries, etc, was not sufficiently portable for the systems where Trilinos needed to be used. But earlier versions of TriBITS used the `FindMPI.cmake` module and that is why the CMake cache variables `MPI_[C,CXX,Fortran]_COMPILER` are defined and still supported.

b) **Configuring to build using raw compilers and flags/libraries:**

While using the MPI compiler wrappers as described above is the preferred way to enable support for MPI, you can also just use the raw compilers and then pass in all of the other information that will be used to compile and link your code.

To turn off the MPI compiler wrappers, set:

```
-D MPI_USE_COMPILER_WRAPPERS=OFF
```

You will then need to manually pass in the compile and link lines needed to compile and link MPI programs. The compile flags can be set through:

```
-D CMAKE_[C,CXX,Fortran]_FLAGS="$EXTRA_COMPILE_FLAGS"
```

The link and library flags must be set through:

```
-D <Project>_EXTRA_LINK_FLAGS="$EXTRA_LINK_FLAGS"
```

Above, you can pass any type of library or other linker flags in and they will always be the last libraries listed, even after all of the TPLs.

NOTE: A good way to determine the extra compile and link flags for MPI is to use:

```
export EXTRA_COMPILE_FLAGS="`$MPI_BIN_DIR/mpiCC --showme:compile`"

export EXTRA_LINK_FLAGS="`$MPI_BIN_DIR/mpiCC --showme:link`"
```

where `MPI_BIN_DIR` is set to your MPI installations binary directory.

c) **Setting up to run MPI programs:**

In order to use the ctest program to run MPI tests, you must set the mpi run command and the options it takes. The built-in logic will try to find the right program and options but you will have to override them in many cases.

MPI test and example executables are passed to CTest `ADD_TEST()` as:

```
ADD_TEST(
  ${MPI_EXEC} ${MPI_EXEC_PRE_NUMPROCS_FLAGS}
  ${MPI_EXEC_NUMPROCS_FLAG} <NP>
  ${MPI_EXEC_POST_NUMPROCS_FLAGS}
  <TEST_EXECUTABLE_PATH> <TEST_ARGS> )
```

where `<TEST_EXECUTABLE_PATH>`, `<TEST_ARGS>`, and `<NP>` are specific to the test being run.

The test-independent MPI arguments are:

```
-D MPI_EXEC:FILEPATH="exec_name"
```

(The name of the MPI run command (e.g. mpirun, mpiexec) that is used to run the MPI program. This can be just the name of the program in which case the full path will be looked for in `${MPI_BIN_DIR}` as described above. If it is an absolute path, it will be used without modification.)

```
-D MPI_EXEC_DEFAULT_NUMPROCS=4
```

(The default number of processes to use when setting up and running MPI test and example executables. The default is set to '4' and only needs to be changed when needed or desired.)

```
-D MPI_EXEC_MAX_NUMPROCS=4
```

(The maximum number of processes to allow when setting up and running MPI test and example executables. The default is set to '4' but should be set to the largest number that can be tolerated for the given machine. Tests with more processes than this are excluded from the test suite at configure time.)

```
-D MPI_EXEC_NUMPROCS_FLAG=-np
```

(The command-line option just before the number of processes to use `<NP>`. The default value is based on the name of `${MPI_EXEC}`, for example, which is `-np` for OpenMPI.)

```
-D MPI_EXEC_PRE_NUMPROCS_FLAGS="arg1;arg2;...;argn"
```

(Other command-line arguments that must come *before* the numprocs argument. The default is empty "".)

```
-D MPI_EXEC_POST_NUMPROCS_FLAGS="arg1;arg2;...;argn"
```

(Other command-line arguments that must come *after* the numprocs argument. The default is empty "".)

NOTE: Multiple arguments listed in `MPI_EXEC_PRE_NUMPROCS_FLAGS` and `MPI_EXEC_POST_NUMPROCS_FLAGS` must be quoted and separated by `';'` as these variables are interpreted as CMake arrays.

4.11 Configuring for OpenMP support

To enable OpenMP support, one must set:

```
-D <Project>_ENABLE_OpenMP=ON
```

Note that if you enable OpenMP directly through a compiler option (e.g., `-fopenmp`), you will NOT enable OpenMP inside <Project> source code.

To skip adding flags for OpenMP for `<LANG>` = `C`, `CXX`, or `Fortran`, use:

```
-D OpenMP_<LANG>_FLAGS_OVERRIDE=" "
```

The single space " " will result in no flags getting added. This is needed since one can't set the flags `OpenMP_<LANG>_FLAGS` to an empty string or the `FIND_PACKAGE(OpenMP)` command will fail. Setting the variable `-DOpenMP_<LANG>_FLAGS_OVERRIDE=" "` is the only way to enable OpenMP but skip adding the OpenMP flags provided by `FIND_PACKAGE(OpenMP)`.

4.12 Building shared libraries

To configure to build shared libraries, set:

```
-D BUILD_SHARED_LIBS=ON
```

The above option will result in all shared libraries to be build on all systems (i.e., `.so` on Unix/Linux systems, `.dylib` on Mac OS X, and `.dll` on Windows systems).

NOTE: If the project has `USE_XSDK_DEFAULTS=ON` set, then this will set `BUILD_SHARED_LIBS=TRUE` by default. Otherwise, the default is `BUILD_SHARED_LIBS=FALSE`

Many systems support a feature called `RPATH` when shared libraries are used that embeds the default locations to look for shared libraries when an executable is run. By default on most systems, CMake will automatically add RPATH directories to shared libraries and executables inside of the build directories. This allows running CMake-built executables from inside the build directory without needing to set `LD_LIBRARY_PATH` on any other environment variables. However, this can be disabled by setting:

```
-D CMAKE_SKIP_BUILD_RPATH=TRUE
```

but it is hard to find a use case where that would be useful.

4.13 Building static libraries and executables

To build static libraries, turn off the shared library support:

```
-D BUILD_SHARED_LIBS=OFF
```

Some machines, such as the Cray XT5, require static executables. To build <Project> executables as static objects, a number of flags must be set:

```
-D BUILD_SHARED_LIBS=OFF \
-D TPL_FIND_SHARED_LIBS=OFF \
-D <Project>_LINK_SEARCH_START_STATIC=ON
```

The first flag tells cmake to build static versions of the <Project> libraries. The second flag tells cmake to locate static library versions of any required TPLs. The third flag tells the auto-detection routines that search for extra required libraries (such as the mpi library and the gfortran library for gnu compilers) to locate static versions.

NOTE: The flag `<Project>_LINK_SEARCH_START_STATIC` is only supported in cmake version 2.8.5 or higher. The variable will be ignored in prior releases of cmake.

4.14 Enabling the usage of resource files to reduce length of build lines

When using the `Unix Makefile` generator and the `Ninja` generator, CMake supports some very useful (undocumented) options for reducing the length of the command-lines used to build object files, create libraries, and link executables. Using these options can avoid troublesome "command-line too long" errors, "Error 127" library creation errors, and other similar errors related to excessively long command-lines to build various targets.

When using the `Unix Makefile` generator, CMake responds to the three cache variables `CMAKE_CXX_USE_RESPONSE_FILE_FOR_INCLUDES`, `CMAKE_CXX_USE_RESPONSE_FILE_FOR_OBJEC` and `CMAKE_CXX_USE_RESPONSE_FILE_FOR_LIBRARIES` described below.

To aggregate the list of all of the include directories (e.g. `'-I <full_path>'`) into a single `*.rsp` file for compiling object files, set:

```
-D CMAKE_CXX_USE_RESPONSE_FILE_FOR_INCLUDES=ON
```

To aggregate the list of all of the object files (e.g. `'<path>/<name>.o'`) into a single `*.rsp` file for creating libraries or linking executables, set:

```
-D CMAKE_CXX_USE_RESPONSE_FILE_FOR_OBJECTS=ON
```

To aggregate the list of all of the libraries (e.g. `'<path>/<libname>.a'`) into a single `*.rsp` file for creating shared libraries or linking executables, set:

```
-D CMAKE_CXX_USE_RESPONSE_FILE_FOR_LIBRARIES=ON
```

When using the `Ninja` generator, CMake only responds to the single option:

```
-D CMAKE_NINJA_FORCE_RESPONSE_FILE=ON
```

which turns on the usage of `*.rsp` response files for include directories, object files, and libraries (and therefore is equivalent to setting the above three `Unix Makefiles` generator options to `ON`).

This feature works well on most standard systems but there are problems in some situations and therefore these options can only be safely enabled on case-by-case basis -- experimenting to ensure they are working correctly. Some examples of some known problematic cases (as of CMake 3.11.2) are:

- CMake will only use resource files with static libraries created with GNU `ar` (e.g. on Linux) but not BSD `ar` (e.g. on MacOS). With BSD `ar`, CMake may break up long command-lines (i.e. lots of object files) with multiple calls to `ar` but that may only work with the `Unix Makefiles` generator, not the `Ninja` generator.
- Some versions of `gfortran` do not accept `*.rsp` files.
- Some versions of `nvcc` (e.g. with CUDA 8.044) do not accept `*.rsp` files for compilation or linking.

Because of problems like these, TriBITS cannot robustly automatically turn on these options. Therefore, it is up to the user to try these options out to see if they work with their specific version of CMake, compilers, and OS.

NOTE: When using the `Unix Makefiles` generator, one can decide to set any combination of these three options based on need and preference and what actually works with a given OS, version of CMake, and provided compilers. For example, on one system `CMAKE_CXX_USE_RESPONSE_FILE_FOR_OBJECTS=ON` may work but `CMAKE_CXX_USE_RESPONSE_FILE_FOR_INCLUDES=ON` may not (which is the case for `gfortran` mentioned above). Therefore, one should experiment carefully and inspect the build lines using `make VERBOSE=1 <target>` as described in Building with verbose output without reconfiguring when deciding which of these options to enable.

NOTE: Newer versions of CMake may automatically determine when these options need to be turned on so watch for that in looking at the build lines.

4.15 Enabling support for an optional Third-Party Library (TPL)

To enable a given TPL, set:

```
-D TPL_ENABLE_<TPLNAME>=ON
```

where <TPLNAME> = BLAS, LAPACK Boost, Netcdf, etc.

The full list of TPLs that is defined and can be enabled is shown by doing a configure with CMake and then grepping the configure output for `Final set of .* TPLs`. The set of TPL names listed in `'Final set of enabled TPLs'` and `'Final set of non-enabled TPLs'` gives the full list of TPLs that can be enabled (or disabled).

Some TPLs require only libraries (e.g. Fortran libraries like BLAS or LAPACK), some TPL require only include directories, and some TPLs require both.

Each TPL specification is defined in a `FindTPL<TPLNAME>.cmake` module file. The job of each of these of these module files is to set the CMake cache variables:

- `TPL_<TPLNAME>_INCLUDE_DIRS:PATH`: List of paths to header files for the TPL (if the TPL supplies header files).
- `TPL_<TPLNAME>_LIBRARIES:PATH`: List of (absolute) paths to libraries, ordered as they will be on the link line (of the TPL supplies libraries).

These variables are the only variables that are actually used in the CMake build system. Therefore, one can set these two variables as CMake cache variables, for `SomeTPL` for example, with:

```
-D TPL_SomeTPL_INCLUDE_DIRS="${LIB_BASE}/include/a;${LIB_BASE}/include/b" \
-D TPL_SomeTPL_LIBRARIES="${LIB_BASE}/lib/liblib1.so;${LIB_BASE}/lib/liblib2.so" \
```

Using this approach, one can be guaranteed that these libraries and these include directories and will used in the compile and link lines for the packages that depend on this TPL `SomeTPL`.

WARNING: When specifying `TPL_<TPLNAME>_INCLUDE_DIRS` and/or `TPL_<TPLNAME>_LIBRARIES`, the build system will use these without question. It will **not** check for the existence of these directories or files so make sure that these files and directories exist before these are used in the compiles and links. (This can actually be a feature in rare cases the libraries and header files don't actually get created until after the configure step is complete but before the build step.)

WARNING: Do **not** try to hack the system and set, for example:

```
TPL_BLAS_LIBRARIES="-L/some/dir -llib1 -llib2 ..."
```

This is not compatible with proper CMake usage and it not guaranteed to be supported for all use cases or all platforms! You should instead always use the full library paths when setting `TPL_<TPLNAME>_LIBRARIES`.

When the variables `TPL_<TPLNAME>_INCLUDE_DIRS` and `TPL_<TPLNAME>_LIBRARIES` are not specified, then most `FindTPL<TPLNAME>.cmake` modules use a default find operation. Some will call `FIND_PACKAGE(<TPLNAME>)` internally by default and some may implement the default find in some other way. To know

for sure, see the documentation for the specific TPL (e.g. looking in the `FindTPL<TPLNAME>.cmake` file to be sure).

Most TPLs, however, use a standard system for finding include directories and/or libraries based on the function TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES(). These simple standard `FindTPL<TPLNAME>.cmake` modules specify a set of header files and/or libraries that must be found. The directories where these header files and library files are looked for are specified using the CMake cache variables:

- `<TPLNAME>_INCLUDE_DIRS:PATH`: List of paths to search for header files using `FIND_FILE()` for each header file, in order.
- `<TPLNAME>_LIBRARY_NAMES:STRING`: List of unadorned library names, in the order of the link line. The platform-specific prefixes (e.g.. 'lib') and postfixes (e.g. '.a', '.lib', or '.dll') will be added automatically by CMake. For example, the library `libblas.so`, `libblas.a`, `blas.lib` or `blas.dll` will all be found on the proper platform using the name `blas`.
- `<TPLNAME>_LIBRARY_DIRS:PATH`: The list of directories where the library files will be searched for using `FIND_LIBRARY()`, for each library, in order.

Most `FindTPL<TPLNAME>.cmake` modules will define a default set of libraries to look for and therefore `<TPLNAME>_LIBRARY_NAMES` can typically be left off.

In order to allow a TPL that normally requires one or more libraries to ignore the libraries, one can set `<TPLNAME>_LIBRARY_NAMES` to empty, for example:

```
-D <TPLNAME>_LIBRARY_NAMES=""
```

Optional package-specific support for a TPL can be turned off by setting:

```
-D <TRIBITS_PACKAGE>_ENABLE_<TPLNAME>=OFF
```

This gives the user full control over what TPLs are supported by which package independently.

Support for an optional TPL can also be turned on implicitly by setting:

```
-D <TRIBITS_PACKAGE>_ENABLE_<TPLNAME>=ON
```

where `<TRIBITS_PACKAGE>` is a TriBITS package that has an optional dependency on `<TPLNAME>`. That will result in setting `TPL_ENABLE_<TPLNAME>=ON` internally (but not set in the cache) if `TPL_ENABLE_<TPLNAME>=OFF` is not already set.

If all the parts of a TPL are not found on an initial configure the configure will error out with a helpful error message. In that case, one can change the variables `<TPLNAME>_INCLUDE_DIRS`, `<TPLNAME>_LIBRARY_NAMES`, and/or `<TPLNAME>_LIBRARY_DIRS` in order to help fund the parts of the TPL. One can do this over and over until the TPL is found. By reconfiguring, one avoid a complete configure from scratch which saves time. Or, one can avoid the find operations by directly setting `TPL_<TPLNAME>_INCLUDE_DIRS` and `TPL_<TPLNAME>_LIBRARIES`.

WARNING: The cmake cache variable `TPL_<TPLNAME>_LIBRARY_DIRS` does **not** control where libraries are found. Instead, this variable is set during the find processes and is not actually used in the CMake build system at all.

In summary, this gives the user complete and direct control in specifying exactly what is used in the build process.

TPL Example 1: Standard BLAS Library

Suppose one wants to find the standard BLAS library `blas` in the directory:

```
/usr/lib/
  libblas.so
  libblas.a
  ...
```

The `FindTPLBLAS.cmake` module should be set up to automatically find the BLAS TPL by simply enabling BLAS with:

```
-D TPL_ENABLE_BLAS=ON
```

This will result in setting the CMake cache variable `TPL_BLAS_LIBRARIES` as shown in the CMake output:

```
-- TPL_BLAS_LIBRARIES='/user/lib/libblas.so'
```

(NOTE: The CMake `FIND_LIBRARY()` command that is used internally will always select the shared library by default if both shared and static libraries are specified, unless told otherwise. See Building static libraries and executables for more details about the handling of shared and static libraries.)

However, suppose one wants to find the `blas` library in a non-default location, such as in:

```
/projects/something/tpls/lib/libblas.so
```

In this case, one could simply configure with:

```
-D TPL_ENABLE_BLAS=ON \
-D BLAS_LIBRARY_DIRS=/projects/something/tpls/lib \
```

That will result in finding the library shown in the CMake output:

```
-- TPL_BLAS_LIBRARIES='/projects/something/tpls/libblas.so'
```

And if one wants to make sure that this BLAS library is used, then one can just directly set:

```
-D TPL_BLAS_LIBRARIES=/projects/something/tpls/libblas.so
```

TPL Example 2: Intel Math Kernel Library (MKL) for BLAS

There are many cases where the list of libraries specified in the `FindTPL<TPLNAME>.cmake` module is not correct for the TPL that one wants to use or is present on the system. In this case, one will need to set the CMake cache variable `<TPLNAME>_LIBRARY_NAMES` to tell the TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES() function what libraries to search for, and in what order.

For example, the Intel Math Kernel Library (MKL) implementation for the BLAS is usually given in several libraries. The exact set of libraries needed depends on the version of MKL, whether 32bit or 64bit libraries are needed, etc. Figuring out the correct set and ordering of these libraries for a given platform may not be trivial. But once the set and the order of the libraries is known, then one can provide the correct list at configure time.

For example, suppose one wants to use the threaded MKL libraries listed in the directories:

```
/usr/local/intel/Compiler/11.1/064/mkl/lib/em64t/
/usr/local/intel/Compiler/11.1/064/lib/intel64/
```

and the list of libraries being searched for is `mkl_intel_lp64`, `mkl_intel_thread`, `mkl_core` and `iomp5`.

In this case, one could specify this with the following do-configure script:

```
#!/bin/bash

INTEL_DIR=/usr/local/intel/Compiler/11.1/064

cmake \
  -D TPL_ENABLE_BLAS=ON \
  -D BLAS_LIBRARY_DIRS="${INTEL_DIR}/em64t;${INTEL_DIR}/intel64" \
  -D BLAS_LIBRARY_NAMES="mkl_intel_lp64;mkl_intel_thread;mkl_core;iomp5" \
  ...
  ${PROJECT_SOURCE_DIR}
```

This would call `FIND_LIBRARY()` on each of the listed library names in these directories and would find them and list them in:

```
-- TPL_BLAS_LIBRARIES='/usr/local/intel/Compiler/11.1/064/em64t/libmkl_intel_lp64.so;
```

(where `...` are the rest of the found libraries.)

NOTE: When shared libraries are used, one typically only needs to list the direct libraries, not the indirect libraries, as the shared libraries are linked to each other.

In this example, one could also play it super safe and manually list out the libraries in the right order by configuring with:

```
-D TPL_BLAS_LIBRARIES="${INTEL_DIR}/em64t/libmkl_intel_lp64.so;..."
```

(where `...` are the rest of the libraries found in order).

4.16 Disabling support for a Third-Party Library (TPL)

Disabling a TPL explicitly can be done using:

```
-D TPL_ENABLE_<TPLNAME>=OFF
```

NOTE: If a disabled TPL is a required dependency of some explicitly enabled downstream package, then the configure will error out if <Project>_DISABLE_ENABLED_FORWARD_D
Otherwise, a WARNING will be printed and the downstream package will be disabled and configuration will continue.

4.17 Disabling tentatively enabled TPLs

To disable a tentatively enabled TPL, set:

```
-D TPL_ENABLE_<TPLNAME>=OFF
```

where `<TPLNAME>` = `BinUtils`, `Boost`, etc.

NOTE: Some TPLs in <Project> are always tentatively enabled (e.g. BinUtils for C++ stacktracing) and if all of the components for the TPL are found (e.g. headers and libraries) then support for the TPL will be enabled, otherwise it will be disabled. This is to allow as much functionality as possible to get automatically enabled without the user having to learn about the TPL,

explicitly enable the TPL, and then see if it is supported or not on the given system. However, if the TPL is not supported on a given platform, then it may be better to explicitly disable the TPL (as shown above) so as to avoid the output from the CMake configure process that shows the tentatively enabled TPL being processes and then failing to be enabled. Also, it is possible that the enable process for the TPL may pass, but the TPL may not work correctly on the given platform. In this case, one would also want to explicitly disable the TPL as shown above.

4.18 Require all TPL libraries be found

By default, some TPLs don't require that all of the libraries listed in `<tplName>_LIBRARY_NAMES` be found. To change this behavior so that all libraries for all enabled TPLs be found, one can set:

```
-D <Project>_MUST_FIND_ALL_TPL_LIBS=TRUE
```

This makes the configure process catch more mistakes with the env.

4.19 Disable warnings from TPL header files

To disable warnings coming from included TPL header files for C and C++ code, set:

```
-D<Project>_TPL_SYSTEM_INCLUDE_DIRS=TRUE
```

On some systems and compilers (e.g. GNU), that will result is include directories for all TPLs to be passed in to the compiler using `-isystem` instead of `-I`.

WARNING: On some systems this will result in build failures involving gfortran and module files. Therefore, don't enable this if Fortran code in your project is pulling in module files from TPLs.

4.20 xSDK Configuration Options

The configure of <Project> will adhere to the xSDK configuration standard (todo: put in reference to final document) simply by setting the CMake cache variable:

```
-D USE_XSDK_DEFAULTS=TRUE
```

Setting this will have the following impact:

- `BUILD_SHARED_LIBS` will be set to `TRUE` by default instead of `FALSE`, which is the default for raw CMake projects (see Building shared libraries).
- `CMAKE_BUILD_TYPE` will be set to `DEBUG` by default instead of `RELEASE` which is the standard TriBITS default (see CMAKE_BUILD_TYPE).
- The compilers in MPI mode `TPL_ENABLE_MPI=ON` or serial mode `TPL_ENABLE_MPI=OFF` will be read from the environment variables `CC`, `CXX` and `FC` if they are set but the cmake cache variables `CMAKE_C_COMPILER`, `CMAKE_C_COMPILER` and `CMAKE_C_COMPILER` are not set. Otherwise, the TriBITS default behavior is to ignore these environment variables in MPI mode.

- The Fortran flags will be read from environment variable `FCFLAGS` if the environment variable `FFLAGS` and the CMake cache variable `CMAKE_Fortran_FLAGS` are empty. Otherwise, raw CMake ignores `FCFLAGS` (see Adding arbitrary compiler flags but keeping default build-type flags).

The rest of the required xSDK configure standard is automatically satisfied by every TriBITS CMake project, including the <Project> project.

4.21 Generating verbose output

There are several different ways to generate verbose output to debug problems when they occur:

a) **Trace file processing during configure:**

```
-D <Project>_TRACE_FILE_PROCESSING=ON
```

This will cause TriBITS to print out a trace for all of the project's, repository's, and package's files get processed on lines using the prefix `File Trace:`. This shows what files get processed and in what order they get processed. To get a clean listing of all the files processed by TriBITS just grep out the lines starting with `-- File Trace:`. This can be helpful in debugging configure problems without generating too much extra output.

Note that <Project>_TRACE_FILE_PROCESSING is set to `ON` automatically when <Project>_VERBOSE_CONFIGURE = `ON`.

b) **Getting verbose output from TriBITS configure:**

To do a complete debug dump for the TriBITS configure process, use:

```
-D <Project>_VERBOSE_CONFIGURE=ON
```

However, this produces a *lot* of output so don't enable this unless you are very desperate. But this level of details can be very useful when debugging configuration problems.

To just view the package and TPL dependencies, it is recommended to use `-D` <Project>_DUMP_PACKAGE_DEPENDENCIES `= ON`.

To just print the link libraries for each library and executable created, use:

```
-D <Project>_DUMP_LINK_LIBS=ON
```

Of course `<Project>_DUMP_PACKAGE_DEPENDENCIES` and `<Project>_DUMP_LINK_LIBS` can be used together. Also, note that `<Project>_DUMP_PACKAGE_DEPENDENCIES` and `<Project>_DUMP_LINK_LIBS` both default t `ON` when `<Project>_VERBOSE_CONFIGURE=ON` on the first configure.

c) **Getting verbose output from the makefile:**

```
-D CMAKE_VERBOSE_MAKEFILE=TRUE
```

NOTE: It is generally better to just pass in `VERBOSE=` when directly calling `make` after configuration is finished. See Building with verbose output without reconfiguring.

d) **Getting very verbose output from configure:**

```
-D <Project>_VERBOSE_CONFIGURE=ON --debug-output --trace
```

NOTE: This will print a complete stack trace to show exactly where you are.

4.22 Enabling/disabling deprecated warnings

To turn off all deprecated warnings, set:

```
-D <Project>_SHOW_DEPRECATED_WARNINGS=OFF
```

This will disable, by default, all deprecated warnings in packages in <Project>. By default, deprecated warnings are enabled.

To enable/disable deprecated warnings for a single <Project> package, set:

```
-D <TRIBITS_PACKAGE>_SHOW_DEPRECATED_WARNINGS=OFF
```

This will override the global behavior set by `<Project>_SHOW_DEPRECATED_WARNINGS` for individual package `<TRIBITS_PACKAGE>`.

4.23 Disabling deprecated code

To actually disable and remove deprecated code from being included in compilation, set:

```
-D <Project>_HIDE_DEPRECATED_CODE=ON
```

and a subset of deprecated code will actually be removed from the build. This is to allow testing of downstream client code that might otherwise ignore deprecated warnings. This allows one to certify that a downstream client code is free of calling deprecated code.

To hide deprecated code for a single <Project> package set:

```
-D <TRIBITS_PACKAGE>_HIDE_DEPRECATED_CODE=ON
```

This will override the global behavior set by `<Project>_HIDE_DEPRECATED_CODE` for individual package `<TRIBITS_PACKAGE>`.

4.24 Outputting package dependency information

To generate the various XML and HTML package dependency files, one can set the output directory when configuring using:

```
-D <Project>_DEPS_DEFAULT_OUTPUT_DIR:FILEPATH=<SOME_PATH>
```

This will generate, by default, the output files `<Project>PackageDependencies.xml`, `<Project>PackageDependenciesTable.html`, and `CDashSubprojectDependencies.xml`. If `<Project>_DEPS_DEFAULT_OUTPUT_DIR` is not set, then the individual output files can be specified as described below.

The filepath for <Project>PackageDependencies.xml can be overridden (or set independently) using:

```
-D <Project>_DEPS_XML_OUTPUT_FILE:FILEPATH=<SOME_FILE_PATH>
```

The filepath for `<Project>PackageDependenciesTable.html` can be overridden (or set independently) using:

```
-D <Project>_DEPS_HTML_OUTPUT_FILE:FILEPATH=<SOME_FILE_PATH>
```

The filepath for CDashSubprojectDependencies.xml can be overridden (or set independently) using:

```
-D <Project>_CDASH_DEPS_XML_OUTPUT_FILE:FILEPATH=<SOME_FILE_PATH>
```

NOTES:

- One must start with a clean CMake cache for all of these defaults to work.
- The files `<Project>PackageDependenciesTable.html` and `CDashSubprojectDependencies.xml` will only get generated if support for Python is enabled.

4.25 Enabling different test categories

To turn on a set a given set of tests by test category, set:

```
-D <Project>_TEST_CATEGORIES="<CATEGORY0>;<CATEGORY1>;..."
```

Valid categories include `BASIC`, `CONTINUOUS`, `NIGHTLY`, `HEAVY` and `PERFORMANCE`. `BASIC` tests get built and run for pre-push testing, CI testing, and nightly testing. `CONTINUOUS` tests are for post-push testing and nightly testing. `NIGHTLY` tests are for nightly testing only. `HEAVY` tests are for more expensive tests that require larger number of MPI processes and longer run times. These test categories are nested (e.g. `HEAVY` contains all `NIGHTLY`, `NIGHTLY` contains all `CONTINUOUS` and `CONTINUOUS` contains all `BASIC` tests). However, `PERFORMANCE` tests are special category used only for performance testing and don't nest with the other categories.

4.26 Disabling specific tests

Any TriBITS-added ctest test (i.e. listed in `ctest -N`) can be disabled at configure time by setting:

```
-D <fullTestName>_DISABLE=ON
```

where `<fulltestName>` must exactly match the test listed out by `ctest -N`. Of course specific tests can also be excluded from `ctest` using the `-E` argument. This will result in the printing of a line for the excluded test when Trace test addition or exclusion is enabled.

4.27 Disabling specific test executable builds

Any TriBITS-added executable (i.e. listed in `make help`) can be disabled from being built by setting:

```
-D <exeTargetName>_EXE_DISABLE=ON
```

where `<exeTargetName>` is the name of the target in the build system.

Note that one should also disable any ctest tests that might use this executable as well with `-D<fullTestName>_DISABLE=ON` (see above). This will result in the printing of a line for the executable target being disabled.

4.28 Trace test addition or exclusion

To see what tests get added and see those that don't get added for various reasons, configure with:

```
-D <Project>_TRACE_ADD_TEST=ON
```

That will print one line per test and will show if the test got added or not. If the test is added, it shows some of the key test properties. If the test did not get added, then this line will show why the test was not added (i.e. due to criteria related to the test's `COMM`, `NUM_MPI_PROCS`, `CATEGORIES`, `HOST`, `XHOST`, `HOSTTYPE`, or `XHOSTTYPE` arguments).

4.29 Setting test timeouts at configure time

A maximum default time limit (timeout) for all the tests can be set at configure time using the cache variable:

```
-D DART_TESTING_TIMEOUT=<maxSeconds>
```

where `<maxSeconds>` is the number of wall-clock seconds. The default for most projects is 1500 seconds (see the default value set in the CMake cache). This value gets scaled by <Project>_SCALE_TEST_TIMEOUT and then set as the field `TimeOut` in the CMake-generated file `DartConfiguration.tcl`. The value `TimeOut` from this file is what is directly read by the `ctest` executable. Timeouts for tests are important. For example, when an MPI program has a defect, it can easily hang (forever) until it is manually killed. If killed by a timeout, CTest will kill the test process and all of its child processes correctly.

NOTES:

- If `DART_TESTING_TIMEOUT` is not explicitly set by the user, then the projects gives it a default value (typically 1500 seconds but see the value in the CMakeCache.txt file).
- If `DART_TESTING_TIMEOUT` is explicitly set to empty (i.e. `-DDART_TESTING_TIMEOUT=`), then by default tests have no timeout and can run forever until manually killed.
- Individual tests may have their timeout limit set on a test-by-test basis internally in the project's `CMakeLists.txt` files (see the `TIMEOUT` argument for `TRIBITS_ADD_TEST()` and `TRIBITS_ADD_ADVANCED_TEST()`). When this is the case, the global timeout set with `DART_TESTING_TIMEOUT` has no impact on these individually set test timeouts.

- Be careful not set the global test timeout too low since if a machine becomes loaded tests can take longer to run and may result in timeouts that would not otherwise occur.
- The value of `DART_TESTING_TIMEOUT` and the timeouts for individual tests can be scaled up or down using the cache varaible <Project>_SCALE_TEST_TIMEOUT.
- To set or override the default global test timeout limit at runtime, see Overriding test timeouts.

4.30 Scaling test timeouts at configure time

The global default test timeout DART_TESTING_TIMEOUT as well as all of the timeouts for the individual tests that have their own timeout set (through the `TIMEOUT` argument for each individual test) can be scaled by a constant factor `<testTimeoutScaleFactor>` by configuring with:

```
-D <Project>_SCALE_TEST_TIMEOUT=<testTimeoutScaleFactor>
```

Here, `<testTimeoutScaleFactor>` can be an integral number like `5` or can be fractional number like `1.5`.

This feature is generally used to compensate for slower machines or overloaded test machines and therefore only scaling factors greater than 1 are to be used. The primary use case for this feature is to add large scale factors (e.g. `40` to `100`) to compensate for running tests using valgrind (see Running memory checking) but this can also be used for debug-mode builds that create tests which run more slowly than for full release-mode optimized builds.

NOTES:

- If `<Project>_SCALE_TEST_TIMEOUT` is not set, the the default value is set to `1.0` (i.e. no scaling of test timeouts).
- When scaling the timeouts, the timeout is first truncated to integral seconds so an original timeout like `200.5` will be truncated to `200` before it gets scaled.
- Only the first fractional digit of `<Project>_SCALE_TEST_TIMEOUT` is used so `1.57` is truncated to `1.5`, for example, before scaling the test timeouts.
- The value of the variable DART_TESTING_TIMEOUT is not changed in the `CMakeCache.txt` file. Only the value of `TimeOut` written into the `DartConfiguration.tcl` file (which is directly read by `ctest`) will be scaled. (This ensures that running configure over and over again will not increase `DART_TESTING_TIMEOUT` or `TimeOut` with each new configure.)

4.31 Enabling support for coverage testing

To turn on support for coverage testing set:

```
-D <Project>_ENABLE_COVERAGE_TESTING=ON
```

This will set compile and link options -fprofile-arcs -ftest-coverage for GCC. Use 'make dashboard' (see below) to submit coverage results to CDash

4.32 Viewing configure options and documentation

a) Viewing available configure-time options with documentation:

```
$ cd $BUILD_DIR
$ rm -rf CMakeCache.txt CMakeFiles/
$ cmake -LAH -D <Project>_ENABLE_ALL_PACKAGES=ON \
  $SOURCE_BASE
```

You can also just look at the text file CMakeCache.txt after configure which gets created in the build directory and has all of the cache variables and documentation.

b) Viewing available configure-time options without documentation:

```
$ cd $BUILD_DIR
$ rm -rf CMakeCache.txt CMakeFiles/
$ cmake -LA <SAME_AS_ABOVE> $SOURCE_BASE
```

c) Viewing current values of cache variables:

```
$ cmake -LA $SOURCE_BASE
```

or just examine and grep the file CMakeCache.txt.

4.33 Enabling extra repositories with add-on packages:

To configure <Project> with an post extra set of packages in extra TriBITS repositories, configure with:

```
-D<Project>_EXTRA_REPOSITORIES="<REPO0>,<REPO1>,..."
```

Here, `<REPOi>` is the name of an extra repository that typically has been cloned under the main <Project> source directory as:

```
<Project>/<REPOi>/
```

For example, to add the packages from SomeExtraRepo one would configure as:

```
$ cd $SOURCE_BASE_DIR
$ git clone some_url.com/some/dir/SomeExtraRepo
$ cd $BUILD_DIR
$ ./do-configure -D<Project>_EXTRA_REPOSITORIES=SomeExtraRepo \
  [Other Options]
```

After that, all of the extra packages defined in `SomeExtraRepo` will appear in the list of official <Project> packages (after all of the native packages) and one is free to enable any of the defined add-on packages just like any other native <Project> package.

NOTE: If `<Project>_EXTRAREPOS_FILE` and `<Project>_ENABLE_KNOWN_EXTERNAL_REPOS_TYPE` are specified, then the list of extra repositories in `<Project>_EXTRA_REPOSITORIES` must be a subset and in the same order as the list extra repos read in from the file specified by <Project>_EXTRAREPOS_FILE. (Also see the variable <Project>_PRE_REPOSITORIES as well.)

4.34 Enabling extra repositories through a file

In order to provide the list of extra TriBITS repositories containing add-on packages from a file, configure with:

```
-D<Project>_EXTRAREPOS_FILE:FILEPATH=<EXTRAREPOSFILE> \
-D<Project>_ENABLE_KNOWN_EXTERNAL_REPOS_TYPE=Continuous
```

Specifying extra repositories through an extra repos file allows greater flexibility in the specification of extra repos. This is not helpful for a basic configure of the project but is useful in automated testing using the `TribitsCTestDriverCore.cmake` script and the `checkin-test.py` script.

The valid values of `<Project>_ENABLE_KNOWN_EXTERNAL_REPOS_TYPE` include `Continuous`, `Nightly`, and `Experimental`. Only repositories listed in the file `<EXTRAREPOSFILE>` that match this type will be included. Note that `Nightly` matches `Continuous` and `Experimental` matches `Nightly` and `Continuous` and therefore includes all repos by default.

If `<Project>_IGNORE_MISSING_EXTRA_REPOSITORIES` is set to `TRUE`, then any extra repositories selected who's directory is missing will be ignored. This is useful when the list of extra repos that a given developer develops or tests is variable and one just wants TriBITS to pick up the list of existing repos automatically.

If the file `<projectDir>/cmake/ExtraRepositoriesList.cmake` exists, then it is used as the default value for `<Project>_EXTRAREPOS_FILE`. However, the default value for `<Project>_ENABLE_KNOWN_EXTERNAL_REPOS_TYPE` is empty so no extra repositories are defined by default unless `<Project>_ENABLE_KNOWN_EXTERNAL_REPOS_TYPE` is specifically set to one of the allowed values.

NOTE: The set of extra repositories listed in the file `<Project>_EXTRAREPOS_FILE` can be filtered down by setting the variables `<Project>_PRE_REPOSITORIES` if PRE extra repos are listed and/or `<Project>_EXTRA_REPOSITORIES` if POST extra repos are listed.

4.35 Selecting a different source location for a package

The source location for any package can be changed by configuring with:

```
-D<TRIBITS_PACKAGE>_SOURCE_DIR_OVERRIDE:STRING=<path>
```

Here, `<path>` can be a relative path or an absolute path, but in both cases must be under the project source directory (otherwise, an error will occur). The relative path will then become the relative path for the package under the binary tree as well.

This can be used, for example, to use a different repository for the implementation of a package that is otherwise snapshotted into the base project source repository (e.g. Kokkos in Trilinos).

4.36 Reconfiguring completely from scratch

To reconfigure from scratch, one needs to delete the the `CMakeCache.txt` and base-level `CMakeFiles/` directory, for example, as:

```
$ rm -rf CMakeCache.txt CMakeFiles/
$ ./do-configure [options]
```

Removing the `CMakeCache.txt` file is often needed when removing variables from the configure line since they are already in the cache. Removing the `CMakeFiles/` directories is needed if there are changes in some CMake modules or the CMake version itself. However, usually removing just the top-level `CMakeCache.txt` and `CMakeFiles/` directory is enough to guarantee a clean reconfigure from a dirty build directory.

If one really wants a clean slate, then try:

```
$ rm -rf `ls | grep -v do-configure`
$ ./do-configure [options]
```

WARNING: Later versions of CMake (2.8.10.2+) require that you remove the top-level `CMakeFiles/` directory whenever you remove the `CMakeCache.txt` file.

4.37 Viewing configure errors

To view various configure errors, read the file:

```
$BUILD_BASE_DIR/CMakeFiles/CMakeError.log
```

This file contains detailed output from try-compile commands, Fortran/C name mangling determination, and other CMake-specific information.

4.38 Adding configure timers

To add timers to various configure steps, configure with:

```
-D <Project>_ENABLE_CONFIGURE_TIMING=ON
```

This will do baulk timing for the major configure steps which is independent of the number of packages in the project.

To additionally add timing for the configure of individual packages, configure with:

```
-D <Project>_ENABLE_CONFIGURE_TIMING=ON \
-D <Project>_ENABLE_PACKAGE_CONFIGURE_TIMING=ON
```

If you are configuring a large number of packages (perhaps by including a lot of add-on packages in extra repos) then you might not want to enable package-by-package timing since it can add some significant overhead to the configure times.

If you just want to time individual packages instead, you can enable that with:

```
-D <Project>_ENABLE_CONFIGURE_TIMING=ON \
-D <TRIBITS_PACKAGE_0>_PACKAGE_CONFIGURE_TIMING=ON \
-D <TRIBITS_PACKAGE_1>_PACKAGE_CONFIGURE_TIMING=ON \
...
```

NOTES:

- This requires that you are running on a Linux/Unix system that has the standard shell command `date`. CMake does not have built-in timing functions so this system command needs to be used instead. This will report timings to 0.001 seconds but note that the overall configure time will go up due to the increased overhead of calling `date` as a process shell command.
- "'WARNING:"' Because this feature has to call the `data` using CMake's `EXECUTE_PROCESS()` command, it can be expensive. Therefore, this should really only be turned on for large projects (where the extra overhead is small) or for smaller projects for extra informational purposes.

4.39 Generating export files

The project <Project> can generate export files for external CMake projects or external Makefile projects. These export files provide the lists of libraries, include directories, compilers and compiler options, etc.

To configure to generate CMake export files for the project, configure with:

```
-D <Project>_ENABLE_INSTALL_CMAKE_CONFIG_FILES=ON
```

This will generate the file `<Project>Config.cmake` for the project and the files `<Package>Config.cmake` for each enabled package in the build tree. In addition, this will install versions of these files into the install tree.

To confiugre Makefile export files, configure with:

```
-D <Project>_ENABLE_EXPORT_MAKEFILES=ON
```

which will generate the file `Makefile.export.<Project>` for the project and the files `Makefile.export.<Package>` for each enabled package in the build tree. In addition, this will install versions of these files into the install tree.

The list of export files generated can be reduced by specifying the exact list of packages the files are requested for with:

```
-D <Project>_GENERATE_EXPORT_FILES_FOR_ONLY_LISTED_SE_PACKAGES="<pkg0>;<pkg1>"
```

NOTES:

- Only enabled packages will have their export files generated.
- One would only want to limit the export files generated for very large projects where the cost my be high for doing so.

4.40 Generating a project repo version file

In development mode working with local git repos for the project sources, on can generate a <Project>RepoVersion.txt file which lists all of the repos and their current versions using:

```
-D <Project>_GENERATE_REPO_VERSION_FILE=ON
```

This will cause a <Project>RepoVersion.txt file to get created in the binary directory, get installed in the install directory, and get included in the source distribution tarball.

4.41 CMake configure-time development mode and debug checking

To turn off CMake configure-time development-mode checking, set:

```
-D <Project>_ENABLE_DEVELOPMENT_MODE=OFF
```

This turns off a number of CMake configure-time checks for the <Project> TriBITS/CMake files including checking the package dependencies. These checks can be expensive and may also not be appropriate for a tarball release of the software. However, this also turns off strong compiler warnings so this is not recommended by default (see <TRIBITS_PACKAGE>_DISABLE_STRONG_WARNINGS). For a release of <Project> this option is set OFF by default.

One of the CMake configure-time debug-mode checks performed as part of `<Project>_ENABLE_DEVELOPMENT_MODE=ON` is to assert the existence of TriBITS package directories. In development mode, the failure to find a package directory is usually a programming error (i.e. a miss-spelled package directory name). But in a tarball release of the project, package directories may be purposefully missing (see Creating a tarball of the source tree) and must be ignored. When building from a reduced tarball created from the development sources, set:

```
-D <Project>_ASSERT_MISSING_PACKAGES=OFF
```

Setting this off will cause the TriBITS CMake configure to simply ignore any missing packages and turn off all dependencies on these missing packages.

Another type of checking is for optional inserted/external packages (e.g. packages who's source can optionally be included in and is flagged with `TRIBITS_ALLOW_MISSING_EXTERNAL_P`. Any of these package directories that are missing result in the packages being silently ignored by default. However, notes on what missing packages are being ignored can printed by configuring with:

```
-D <Project>_WARN_ABOUT_MISSING_EXTERNAL_PACKAGES=TRUE
```

These warnings (starting with 'NOTE', not 'WARNING' that would otherwise trigger warnings in CDash) about missing inserted/external packages will print regardless of the setting for `<Project>_ASSERT_MISSING_PACKAGES`.

5 Building (Makefile generator)

This section described building using the default CMake Makefile generator. Building with the Ninja is described in section Building (Ninja generator). But every other CMake generator is also supported such as Visual Studio on Windows, XCode on Macs, and Eclipse project files but using those build systems are not documented here (consult standard CMake and concrete build tool documentation).

5.1 Building all targets

To build all targets use:

```
$ make [-jN]
```

where `N` is the number of processes to use (i.e. 2, 4, 16, etc.) .

5.2 Discovering what targets are available to build

CMake generates Makefiles with a 'help' target! To see the targets at the current directory level type:

```
$ make help
```

NOTE: In general, the `help` target only prints targets in the current directory, not targets in subdirectories. These targets can include object files and all, anything that CMake defines a target for in the current directory. However, running `make help` it from the base build directory will print all major targets in the project (i.e. libraries, executables, etc.) but not minor targets like object files. Any of the printed targets can be used as a target for `make <some-target>`. This is super useful for just building a single object file, for example.

5.3 Building all of the targets for a package

To build only the targets for a given TriBITS package, one can use:

```
$ make <TRIBITS_PACKAGE>_all
```

or:

```
$ cd packages/<TRIBITS_PACKAGE>
$ make
```

This will build only the targets for TriBITS package `<TRIBITS_PACKAGE>` and its required upstream targets.

5.4 Building all of the libraries for a package

To build only the libraries for given TriBITS package, use:

```
$ make <TRIBITS_PACKAGE>_libs
```

5.5 Building all of the libraries for all enabled packages

To build only the libraries for all enabled TriBITS packages, use:

```
$ make libs
```

NOTE: This target depends on the `<PACKAGE>_libs` targets for all of the enabled `<Project>` packages. You can also use the target name `'<Project>_libs`.

5.6 Building a single object file

To build just a single object file (i.e. to debug a compile problem), first, look for the target name for the object file build based on the source file, for example for the source file `SomeSourceFile.cpp`, use:

```
$ make help | grep SomeSourceFile
```

The above will return a target name like:

```
... SomeSourceFile.o
```

To find the name of the actual object file, do:

```
$ find . -name "*SomeSourceFile*.o"
```

that will return something like:

```
./CMakeFiles/<source-dir-path>.dir/SomeSourceFile.cpp.o
```

(but this file location and name depends on the source directory structure, the version of CMake, and other factors). Use the returned name (exactly) for the object file returned in the above find operation to remove the object file first, for example, as:

```
$ rm ./CMakeFiles/<source-dir-path>.dir/SomeSourceFile.cpp.o
```

and then build it again, for example, with:

```
$ make SomeSourceFile.o
```

Again, the names of the target and the object file name an location depend on the CMake version, the structure of your source directories and other factors but the general process of using `make help | grep <some-file-base-name>` to find the target name and then doing a find `find . -name "*<some-file-base-name>*"` to find the actual object file path always works.

For this process to work correctly, you must be in the subdirectory where the `TRIBITS_ADD_LIBRARY()` or `TRIBITS_ADD_EXECUTABLE()` command is called from its `CMakeLists.txt` file, otherwise the object file targets will not be listed by `make help`.

NOTE: CMake does not seem to not check on dependencies when explicitly building object files as shown above so you need to always delete the object file first to make sure that it gets rebuilt correctly.

5.7 Building with verbose output without reconfiguring

One can get CMake to generate verbose make output at build time by just setting the Makefile variable `VERBOSE=1`, for example, as:

```
$ make  VERBOSE=1 [<SOME_TARGET>]
```

Any number of compile or linking problem can be quickly debugged by seeing the raw compile and link lines. See Building a single object file for more details.

NOTE: The libraries listed on the link line are often in the form `-L<lib-dir> -l<lib1> -l<lib2>` even if one passed in full library paths for TPLs through `TPL_<TPLNAME>_LIBRARIES` (see Enabling support for an optional Third-Party Library (TPL)). That is because CMake tries to keep the link lines as short as possible and therefore it often does this translation automatically (whether you want it to or not).

5.8 Relink a target without considering dependencies

CMake provides a way to rebuild a target without considering its dependencies using:

```
$ make <SOME_TARGET>/fast
```

6 Building (Ninja generator)

When using the Ninja back-end (see Enabling support for Ninja), one can build with simply:

```
ninja -j<N>
```

or use any options and workflows that the raw `ninja` executable supports (see `ninja --help`). In general, the `ninja` command can only be run from the base project binary directory and running it from the subdirectory will not work without having to use the `-C <dir>` option pointing to the base dir and one will need to pass in specific target names or the entire project targets will get built with the default `all` target.

But if the TriBITS-created Ninja makefiles are also generated (see <Project>_WRITE_NINJA_MAKEF then `make` can be run from any subdirectory to build targets in that subdirectory. Because of this and other advantages of these makefiles, the majority of the instructions below will be for running with these makefiles, not the raw `ninja` command. These makefiles define many of the standard targets that are provided by the default CMake-generated makefiles like `all`, `clean`, `install`, and `package_source` (run `make help` to see all of the targets).

6.1 Building in parallel with Ninja

By default, running the raw `ninja` command:

```
ninja
```

will use **all** of the free cores on the node to build targets in parallel on the machine! This will not overload the machine but it will not leave any unused cores either (see Ninja documentation).

To run the raw `ninja` command to build with a specific number of build processes (regardless of machine load), e.g. `16` build processes, use:

```
ninja -j16
```

When using the TriBITS-generated Ninja makefiles, running with:

```
make
```

will also use all of the free cores, and **not** just one process like with the default CMake-generated makefiles.

But with the TriBITS-generated Ninja makefiles, to build with a specific number of build processes (regardless of machine load), e.g. `16` build processes, one can **not** use `-j<N>` but instead must use the `NP=<N>` argument with:

```
make NP=16
```

which will call `ninja -j16` internally.

That reason that `-j16` cannot be used with these TriBITS-generated Ninja Makefiles is that the `make` program does not inform the executed `Makefile` the value of this option and therefore this can't be passed on to the underlying `ninja` command. Therefore the `make` option `-j<N>` is essentially ignored. Therefore, running `make -j16` will result in calling raw `ninja` which will use all of the free cores on the machine. Arguably that is better than using only one core and will not overload the machine but still this is behavior the user must be aware.

6.2 Building in a subdirectory with Ninja

To build from a binary subdirectory in the build tree with the TriBITS-generated Ninja makefiles, just `cd` into that directory and build with:

```
cd <some-subdir>/
make NP=16
```

and this will only build targets that are defined in that subdirectory. (See the raw `ninja` command that gets called in this case which is echoed at the top.)

6.3 Building verbose without reconfiguring with Ninja

To build targets and see the full build lines for each with the Ninja makefiles, build with:

```
make NP=10 VERBOSE=1 <target_name>
```

But note that `ninja` will automatically provide the full build command for a build target when that target fails so the `VERBOSE=1` option is not needed in the case were a build target is failing but is useful in other cases none the less.

6.4 Discovering what targets are available to build with Ninja

To determine the target names for library, executable (or any other general target except for object files) that can be built in any binary directory with the TriBITS-generated Ninja Makefiles, use:

```
make help
```

which will return:

```
This Makefile supports the following standard targets:

  all (default)
  clean
  help
  install
  test
  package
  package_source
  edit_cache
  rebuild_cache

and the following project targets:

  <target0>
  <target1>
  ...

Run 'make help-objects' to list object files.
```

To determine the target names for building any object files that can be run in any directory with the TriBITS-generated Ninja Makefiles, use:

```
make help-objects
```

which will return:

```
This Makefile supports the following object files:

  <object-target-0>
  <object-target-1>
  ...
```

NOTE: The raw `ninja` command does not provide a compact way to list all of the targets that can be built in any given directory.

6.5 Building specific targets with Ninja

To build with any specific target, use:

```
make NP=16 <target>
```

See Discovering what targets are available to build with Ninja for how to get a list of targets.

6.6 Building single object files with Ninja

To build any object file, use:

```
make NP=16 <object-target>
```

See Discovering what targets are available to build with Ninja for how to get a list of the object file targets.

Note that unlike the native CMake-generated Makefiles, when an object target like this gets built, Ninja will build all of the upstream targets as well. For example, if you change an upstream header file and just want to see the impact of building a single `*.o` file, this target will build **all** of the targets for the library where the object fill will gets used. But this is not generally what one wants to do to iteratively develop the compilation of a single object file.

To avoid that behavior and instead just build a single `*.o` file, first one must instead use:

```
make VERBOSE=1 <object-target>
```

to print the command-line for building the one object file, and then `cd` to the base project binary directory and manually run that command to build only that object file. (This can be considered a regression w.r.t. the native CMake-generated Makefiles.)

NOTE: The raw `ninja` command does not provide a compact way to list all of the object files that can be built and does not make it easy to build a single object file.

6.7 Cleaning build targets with Ninja

With the TriBITS-generated Ninja Makefiles, when one runs:

```
make clean
```

in a subdirectory to clean out the targets in that subdirectory, the underlying `ninja` command will actually delete not only the targets in that subdirectory but instead will clean **all** the targets upstream from the targets in the current subdirectory as well! This is **not** the behavior of the default CMake-generated Makefiles where only the generated files in that subdirectory will be removed and files for upstream dependencies.

Therefore, if one then wants to clean only the object files, libraries, and executables in a subdirectory, one should just manually delete them with:

```
cd <some-subdir>/
find . -name "*.o" -exec rm {} \;
find . -name "lib*.a" -exec rm {} \;
find . -name "lib*.so*" -exec rm {} \;
find . -name "*.exe" -exec rm {} \;
```

then one can rebuild just the targets in that subdirectory with:

```
make NP=10
```

7 Testing with CTest

This section assumes one is using the CMake Makefile generator described above. Also, the `ctest` does not consider make dependencies when running so the software must be completely built before running `ctest` as described here.

7.1 Running all tests

To run all of the defined tests (i.e. created using `TRIBITS_ADD_TEST()` or `TRIBITS_ADD_ADVANCED_TEST()`) use:

```
$ ctest -j<N>
```

(where `<N>` is an integer for the number of processes to try to run tests in parallel). A summary of what tests are run and their pass/fail status will be printed to the screen. Detailed output about each of the tests is archived in the generate file:

```
Testing/Temporary/LastTest.log
```

where CTest creates the `Testing` directory in the local directory where you run it from.

NOTE: The `-j<N>` argument allows CTest to use more processes to run tests. This will intelligently load balance the defined tests with multiple processes (i.e. MPI tests) and will try not exceed the number of processes `<N>`. However, if tests are defined that use more that `<N>` processes, then CTest will still run the test but will not run any other tests while the limit of `<N>` processes is exceeded. To exclude tests that require more than `<N>` processes, set the cache variable `MPI_EXEC_MAX_NUMPROCS` (see Configuring with MPI support).

7.2 Only running tests for a single package

Tests for just a single TriBITS package can be run with:

```
$ ctest -j4 -L <TRIBITS_PACKAGE>
```

or:

```
$ cd packages/<TRIBITS_PACKAGE>
$ ctest -j4
```

This will run tests for packages and subpackages inside of the parent package `<TRIBITS_PACKAGE>`.

NOTE: CTest has a number of ways to filter what tests get run. You can use the test name using `-E`, you can exclude tests using `-I`, and there are other approaches as well. See `ctest --help` and on-line documentation, and experiment for more details.

7.3 Running a single test with full output to the console

To run just a single test and send detailed output directly to the console, one can run:

```
$ ctest -R ^<FULL_TEST_NAME>$ -VV
```

However, when running just a single test, it is usually better to just run the test command manually to allow passing in more options. To see what the actual test command is, use:

```
$ ctest -R ^<FULL_TEST_NAME>$ -VV -N
```

This will only print out the test command that `ctest` runs and show the working directory. To run the test exactly as `ctest` would, cd into the shown working directory and run the shown command.

7.4 Overriding test timeouts

The configured global test timeout described in `Setting test timeouts at configure time` can be overridden on the CTest command-line as:

```
$ ctest --timeout <maxSeconds>
```

This will override the configured cache variable DART_TESTING_TIMEOUT (actually, the scaled value set as `TimeOut` in the file `DartConfiguration.tcl`). However, this will **not** override the test time-outs set on individual tests on a test-by-test basis!

WARNING: Do not try to use `--timeout=<maxSeconds>` or CTest will just ignore the argument!

7.5 Running memory checking

To configure for running memory testing with `valgrind`, use:

```
-D MEMORYCHECK_COMMAND=<abs-path-to-valgrind>/valgrind \
-D MEMORYCHECK_SUPPRESSIONS_FILE=<abs-path-to-supp-file0> \
-D MEMORYCHECK_COMMAND_OPTIONS="-q --trace-children=yes --tool=memcheck \
  --leak-check=yes --workaround-gcc296-bugs=yes \
  --num-callers=50 --suppressions=<abs-path-to-supp-file1> \
  ... --suppressions=<abs-path-to-supp-fileN>"
```

Above, you have to set the absolute path to the valgrind executable to run using `MEMORYCHECK_COMMAND` as CMake will not find this for you by default. To use a single valgrind suppression file, just set `MEMORYCHECK_SUPPRESSIONS_FILE` to the path of that suppression file as shown above. To add other suppression files, they have to be added as other general valgrind arguments in `MEMORYCHECK_COMMAND_OPTIONS` as shown.

After configuring with the above options, to run the memory tests for all enabled tests, from the **base** project build directory, do:

```
$ ctest -T memcheck
```

This will run valgrind on **every** test command that is run by ctest.

To run valgrind on the tests for a single package, from the **base** project directory, do:

```
$ ctest -T memcheck -L <TRIBITS_PACKAGE>
```

To run valgrind on a specific test, from the **base** project directory, do:

```
$ ctest -T memcheck -R ^<FULL_TEST_NAME>$
```

Detailed output from valgrind is printed in the file:

```
Testing/Temporary/LastDynamicAnalysis_<DATE_TIME>.log
```

NOTE: If you try to run memory tests from any subdirectories, it will not work. You have to run them from the ***base** project build directory as shown above. A nice way to view valgrind results is to submit to CDash using the `dashboard` target (see Dashboard submissions).

NOTE: You have to use the valgrind option `--trace-children=yes` to trace through child processes. This is needed if you have tests that are given as CMake -P scripts (such as advanced tests) or tests driven in bash, Perl, Python, or other languages.

8 Installing

After a build and test of the software is complete, the software can be installed. Actually, to get ready for the install, the install directory must be specified at configure time by setting the variable `CMAKE_INSTALL_PREFIX` in addition to other variables that affect the installation (see the following sections). The other commands described below can all be run after the build and testing is complete.

For the most typical case where the software is build and installed on the same machine in the same location where it will be used, one just needs to configure with:

```
$ cmake -DCMAKE_INSTALL_PREFIX=<install-base-dir> [other options] \
  ${SOURCE_DIR}
$ make -j<N> install
```

For more details, see the following subsections:

- Setting the install prefix
- Setting install RPATH
- Avoiding installing libraries and headers
- Installing the software

8.1 Setting the install prefix

In order to set up for the install, the install prefix should be set up at configure time by setting, for example:

```
-D CMAKE_INSTALL_PREFIX=$HOME/install/<Project>/mpi/opt
```

The default location for the installation of libraries, headers, and executables is given by the variables (with defaults):

```
-D <Project>_INSTALL_INCLUDE_DIR="include" \
-D <Project>_INSTALL_LIB_DIR="lib" \
-D <Project>_INSTALL_RUNTIME_DIR="bin" \
-D <Project>_INSTALL_EXAMPLE_DIR="example"
```

If these paths are relative (i.e. don't start with "/" and use type `STRING`) then they are relative to `${CMAKE_INSTALL_PREFIX}`. Otherwise the paths can be absolute (use type `PATH`) and don't have to be under `${CMAKE_INSTALL_PREFIX}`. For example, to install each part in any arbitrary location use:

```
-D <Project>_INSTALL_INCLUDE_DIR="/usr/<Project>_include" \
-D <Project>_INSTALL_LIB_DIR="/usr/<Project>_lib" \
-D <Project>_INSTALL_RUNTIME_DIR="/usr/<Project>_bin" \
-D <Project>_INSTALL_EXAMPLE_DIR="/usr/share/<Project>/examples"
```

NOTE: The defaults for the above include paths will be set by the standard CMake module `GNUInstallDirs` if `<Project>_USE_GNUINSTALLDIRS=TRUE` is set. Some projects have this set by default (see the `CMakeCache.txt` after configuring to see default being used by this project).

WARNING: To overwrite default relative paths, you must use the data type `STRING` for the cache variables. If you don't, then CMake will use the current binary directory for the base path. Otherwise, if you want to specify absolute paths, use the data type `PATH` as shown above.

8.2 Setting install RPATH

Setting RPATH for installed shared libraries and executables (i.e. `BUILD_SHARED_LIBS=ON`) can be a little tricky. Some discussion of how raw CMake handles RPATH and installations can be found at:

https://cmake.org/Wiki/CMake_RPATH_handling

The TriBITS/CMake build system being used for this <Project> CMake project defines the following default behavior for installed RPATH (which is not the same as the raw CMake default behavior):

- `CMAKE_INSTALL_RPATH` for all libraries and executables built and installed by this CMake project is set to `${<Project>_INSTALL_LIB_DIR}`. (This default is controlled by the variable <Project>_SET_INSTALL_RPATH.)
- The path for all shared external libraries (i.e. TPLs) is set to the location of the external libraries passed in (or automatically discovered) at configure time. (This is controlled by the built-in CMake cache variable `CMAKE_INSTALL_RPATH_USE_LINK_PATH` which is set to `TRUE` by default for most TriBITS projects but is empty "" for raw CMake.)

The above default behavior allows the installed executables and libraries to be run without needing to set `LD_LIBRARY_PATH` or any other system environment variables. However, this setting does not allow the installed libraries and executables to be easily moved or relocated. There are several built-in CMake variables that control how RPATH is handled related to installations. The build-in CMake variables that control RPATH handling include `CMAKE_INSTALL_RPATH`, `CMAKE_SKIP_BUILD_RPATH`, `CMAKE_SKIP_INSTALL_RPATH`, `CMAKE_SKIP_RPATH`, `CMAKE_BUILD_WITH_INSTALL_RPATH`, `CMAKE_INSTALL_RPATH_USE_LINK_PATH`. The TriBITS/CMake build system for <Project> respects all of these raw CMake variables and their documented effect on the build and install.

In addition, this TriBITS/CMake project defines the cache variable:

<Project>_SET_INSTALL_RPATH: If `TRUE`, then the global CMake variable `CMAKE_INSTALL_RPATH` is set to `<Project>_INSTALL_LIB_DIR`. If `CMAKE_INSTALL_RPATH` is set by the user, then that is used instead. This avoids having to manually set `CMAKE_INSTALL_RPATH` to the correct default install directory.

Rather than re-documenting all of the native CMake RPATH variables mentioned above, instead, we describe how these variables should be set for different installation and distribution scenarios:

0. Use default CMake behavior
1. Libraries and executables are built, installed and used on same machine (TriBITS default)
2. Targets will move after installation
3. Targets and TPLs will move after installation
4. Explicitly set RPATH for the final target system

5. Define all shared library paths at runtime using environment variables

These scenarios in detail are:

0. *Use default CMake behavior:* If one just wants the default raw CMake behavior with respect to RPATH, then configure with:

   ```
   -D<Project>_SET_INSTALL_RPATH=FALSE \
   -DCMAKE_INSTALL_RPATH_USE_LINK_PATH=FALSE \
   ```

   This will not put any directories into RPATH for the installed libraries or executables. This is the same behavior as setting `CMAKE_SKIP_INSTALL_RPATH=TRUE` (see Define all shared library paths at runtime using environment variables).

1. *Libraries and executables are built, installed and used on same machine (TriBITS default):* One needs no options for this behavior but to make this explicit then configure with:

   ```
   -D<Project>_SET_INSTALL_RPATH=TRUE \
   -DCMAKE_INSTALL_RPATH_USE_LINK_PATH=TRUE \
   ```

   As described above, this allows libraries and executables to be used right away once installed without needing to set any environment variables.

   Note that this also allows the installed libraries and executables to be moved to the same location on an different but identical machine as well.

2. *Targets will move after installation:* In this scenario, the final location of built libraries and executables will be different on the same machine or an otherwise identical machine. In this case, we assume that all of the external library references and directories would be the same. In this case, one would generally configure with:

   ```
   -D<Project>_SET_INSTALL_RPATH=FALSE \
   -DCMAKE_INSTALL_RPATH_USE_LINK_PATH=TRUE \
   ```

   Then, to run any executables using these shared libraries, one must update LD_LIBRARY_PATH as:

   ```
   $ export LD_LIBRARY_PATH=<final-install-dir>/lib:$LD_LIBRARY_PATH
   ```

   Or, if the final directory location is known, then one can directly `CMAKE_INSTALL_RPATH` at configure time to match the final target system and then one does not need to mess with `LD_LIBRARY_PATH` or any other env variables (see Explicitly set RPATH for the final target system).

3. *Targets and TPLs will move after installation*: In this scenario, the final location of the installed libraries and executables will not be the same as the initial install location and the external library

(i.e. TPL) locations may not be the same on the final target machine. This can be handled in one of two ways. First, if one knows the final target machine structure, then one can directly set `CMAKE_INSTALL_RPATH` to the locations on the final target machine (see Explicitly set RPATH for the final target system). Second, if one does not know the final machine directory structure (or the same distribution needs to support several different systems with different directory structures), then one can set `CMAKE_SKIP_INSTALL_RPATH=TRUE` and then require setting the paths in the env (see Define all shared library paths at runtime using environment variables).

4. *Explicitly set RPATH for the final target system:* If one knows the directory structure of the final target machine where the installed libraries and executables will be used, then one can set those paths at configure time with:

   ```
   -D<Project>_SET_INSTALL_RPATH=FALSE \
   -DCMAKE_INSTALL_RPATH_USE_LINK_PATH=FALSE \
   -DCMAKE_INSTALL_RPATH="<path0>;<path1>;..." \
   ```

   In this case `CMAKE_INSTALL_RPATH` is explicitly set. (The value of `<Project>_SET_INSTALL_RPATH` has no effect but setting it to `FALSE` may help to avoid confusion.)

   Once the install directories are moved to the final location, the executables can be run without any need to set environment variables.

   Note that TriBITS also accepts the directory separator "`:`" for:

   ```
   -DCMAKE_INSTALL_RPATH="<path0>:<path1>:..." \
   ```

   and replaces it internally with "`;`" which raw CMake requires. (This makes it more robust to pass around inside of CMake code since "`;`" means array boundary with CMake.). However, since "`:`" is not a valid character for a path for any Unix system, this is a safe substitution (and `CMAKE_INSTALL_RPATH` is not used on Windows systems that allow "`:`" in a directory path).

   Also note that Linux supports RPATHs with the special value `$ORIGIN` to allow for relative paths and for relocatable installations. (Mac OSX has similar variables like `@executable_path`.) With this, one can define `CMAKE_INSTALL_RPATH` using something like `$ORIGIN/../lib`. See the above CMake RPATH handling reference for more details.

5. *Define all shared library paths at runtime using environment variables:* If one wants complete freedom to define the paths for the shared libraries at runtime with `LD_LIBRARY_PATH` on Linux (and similar variables on other platforms), then one can completely strip RPATH out of the installed libraries and executables by configuring with:

   ```
   -DCMAKE_SKIP_INSTALL_RPATH=TRUE \
   ```

This will result in all paths being stripped out of RPATH regardless of the values of `<Project>_SET_INSTALL_RPATH` or `CMAKE_INSTALL_RPATH_USE_LINK_PATH`. (This is the same default behavior as raw CMake, see Use default CMake behavior).

Then the runtime environment must be set up to find the correct shared libraries in the correct order at runtime (e.g. by setting `LD_LIBRARY_PATH`) . But this approach provides the most flexibility about where executables and libraries are installed and run from.

Also note that, while not necessary, in order to avoid confusion, it is likely desired to configure with:

```
-D<Project>_SET_INSTALL_RPATH=FALSE \
-DCMAKE_INSTALL_RPATH_USE_LINK_PATH=FALSE \
-DCMAKE_SKIP_INSTALL_RPATH=TRUE \
```

This will produce the least confusing CMake configure output.

One additional issue about RPATH handling on Mac OSX systems needs to be mentioned. That is, in order for this default RPATH approach to work on OSX systems, all of the upstream shared libraries must have `@rpath/lib<libname>.dylib` embedded into them (as shown by `otool -L <lib_or_exec>`). For libraries built and installed with CMake, the parent CMake project must be configured with:

```
-DBUILD_SHARED_LIBS=ON \
-DCMAKE_INSTALL_RPATH_USE_LINK_PATH=TRUE \
-DCMAKE_MACOSX_RPATH=TRUE \
```

For other build systems, see their documentation for shared library support on OSX. To see the proper way to handle RPATH on OSX, just inspect the build and install commands that CMake generates (e.g. using `make VERBOSE=1 <target>`) for shared libraries and then make sure that these other build systems use equivalent commands. If that is done properly for the chain of all upstream shared libraries then the behaviors of this <Project> CMake project described above should hold on OSX systems as well.

8.3 Avoiding installing libraries and headers

By default, any libraries and header files defined by in the TriBITS project <Project> will get installed into the installation directories specified by `CMAKE_INSTALL_PREFIX`, `<Project>_INSTALL_INCLUDE_DIR` and `<Project>_INSTALL_LIB_DIR`. However, if the primary desire is to install executables only, then the user can set:

```
-D <Project>_INSTALL_LIBRARIES_AND_HEADERS=OFF
```

which, if in addition static libraries are being built (i.e. `BUILD_SHARED_LIBS=OFF`), this this option will result in no libraries or headers being installed into the `<install>/include/` and `<install>/lib/` directories, respectively. However, if shared libraries are being built (i.e. `BUILD_SHARED_LIBS=ON`), they the libraries will be installed in `<install>/lib/` along with the executables because the executables can't run without the shared libraries being installed.

8.4 Installing the software

To install the software, type:

```
$ make install
```

Note that CMake actually puts in the build dependencies for installed targets so in some cases you can just type `make -j<N> install` and it will also build the software before installing. However, it is advised to always build and test the software first before installing with:

```
$ make -j<N> && ctest -j<N> && make -j<N> install
```

This will ensure that everything is built correctly and all tests pass before installing.

WARNING: When using shared libraries, one must be careful to avoid the error **"RegularExpression::compile(): Expression too big."** when using RPATH and when RPATH gets stripped out on install. To avoid this error, use the shortest build directory you can, like:

```
$HOME/<Project>_BUILD/
```

This has been shown to allow even the most complex TriBITS-based CMake projects to successfully install and avoid this error.

NOTE: This problem has been resolved in CMake versions 3.6.0+ and does not require a short build directory path.

9 Packaging

Packaged source and binary distributions can also be created using CMake and CPack.

9.1 Creating a tarball of the source tree

To create a source tarball of the project, first configure with the list of desired packages (see Selecting the list of packages to enable) and pass in

```
-D <Project>_ENABLE_CPACK_PACKAGING=ON
```

To actually generate the distribution files, use:

```
$ make package_source
```

The above command will tar up *everything* in the source tree except for files explicitly excluded in the CMakeLists.txt files and packages that are not enabled so make sure that you start with a totally clean source tree before you do this. You can clean the source tree first to remove all ignored files using:

```
$ git clean -fd -x
```

You can include generated files in the tarball, such as Doxygen output files, by creating them first, then running `make package_source` and they will be included in the distribution (unless there is an internal exclude set).

Disabled subpackages can be included or excluded from the tarball by setting `<Project>_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION` (the TriBITS

project has its own default, check `CMakeCache.txt` to see what the default is). If `<Project>_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION=ON` and but one wants to include some subpackages that are otherwise excluded, just enable them or their outer package so they will be included in the source tarball. To get a printout of set regular expressions that will be used to match files to exclude, set:

```
-D <Project>_DUMP_CPACK_SOURCE_IGNORE_FILES=ON
```

While a set of default CPack source generator types is defined for this project (see the `CMakeCache.txt` file), it can be overridden using, for example:

```
-D <Project>_CPACK_SOURCE_GENERATOR="TGZ;TBZ2"
```

(see CMake documentation to find out the types of supported CPack source generators on your system).

NOTE: When configuring from an untarred source tree that has missing packages, one must configure with:

```
-D <Project>_ASSERT_MISSING_PACKAGES=OFF
```

Otherwise, TriBITS will error out complaining about missing packages. (Note that `<Project>_ASSERT_MISSING_PACKAGES` will default to `'OFF'` in release mode, i.e. `<Project>_ENABLE_DEVELOPMENT_MODE==OFF`.)

10 Dashboard submissions

All TriBITS projects have built-in support for submitting configure, build, and test results to CDash using the custom `dashbaord` target. This uses the TRIBITS_CTEST_DRIVER() function internally set up to work correctly from an existing binary directory with a valid initial configure. The few of the advantages of using the custom TriBITS-enabled `dashboard` target over just using the standard `ctest -D Experimental` command are:

- The configure, build, and test results are broken down nicely package-by-package on CDash.
- Additional notes files will be uploaded to the build on CDash.

For more details, see TRIBITS_CTEST_DRIVER().

To use the `dashboard` target, first, configure as normal but add cache vars for the the build and test parallel levels with:

```
-DCTEST_BUILD_FLAGS=-j4 -DCTEST_PARALLEL_LEVEL=4
```

(or with some other values `-j<N>`). Then, invoke the (re)configure, build, test and submit with:

```
$ make dashboard
```

This invokes the TRIBITS_CTEST_DRIVER() function to do an experimental build for all of the packages that you have enabled tests. (The packages that are implicitly enabled due to package dependencies are not directly processed and no rows on CDash will be show up for those packages.)

NOTE: This generates a lot of output, so it is typically better to pipe this to a file with:

```
$ make dashboard &> make.dashboard.out
```

and then watch that file in another terminal with:

```
$ tail -f make.dashboard.out
```

There are a number of options that you can set in the cache and/or in the environment to control what this script does. For the full set of options, see TRIBITS_CTEST_DRIVER(). To see the full list of options, and their default values, one can run with:

```
$ env CTEST_DO_SUBMIT=FALSE CTEST_DEPENDENCY_HANDLING_UNIT_TESTING=TRUE \
  make dashboard
```

This will print the options with their default values and then do a sort of mock running of the CTest driver script and point out what it will do with the given setup.

One option one might what to set is the build name with:

```
$ env CTEST_BUILD_NAME=MyBuild make dashboard
```

After this finishes running, look for the build 'MyBuild' (or whatever build name you used above) in the <Project> CDash dashboard (the CDash URL is printed at the end of STDOUT). It is useful to set `CTEST_BUILD_NAME` to some unique name to make it easier to find your results in the CDash dashboard. If one does not set `CTEST_BUILD_NAME`, the name of the binary directory is used instead by default (which may not be very descriptive if it called `BUILD` or something like that).

If there is already a valid configure and build and one does not want to submit configure and build results to CDash, then one can run with:

```
$ env CTEST_BUILD_NAME=<build-name> CTEST_DO_CONFIGURE=OFF CTEST_DO_BUILD=OFF \
  make dashboard
```

which will only run the enabled tests and submit results to the CDash build `<build-name>`.

The configure, builds, and submits are either done package-by-package or all-at-once as controlled by the varaible `<Project>_CTEST_DO_ALL_AT_ONCE`. This can be set in the CMake cache when configuring the project using:

```
-D<Project>_CTEST_DO_ALL_AT_ONCE=TRUE
```

or when running the `dashboard` target with:

```
$ env <Project>_CTEST_DO_ALL_AT_ONCE=TRUE make dashbaord.
```

Using the `dashboard` target, one can also run coverage and memory testing and submit to CDash as described below. But to take full advantage of the all-at-once mode and to have results displayed on CDash broken down package-by-package, one must be using CMake/CTest 3.10 or newer and be submitting to a newer CDash version (from about mid 2018 and newer).

For submitting line coverage results, once you configure with `-D<Project>_ENABLE_COVERAGE_TESTING=ON` the environment variable `CTEST_DO_COVERAGE_TESTING=TRUE` is automatically

set by the target `dashboard` so you don't have to set this yourself. Then when you run the `dashboard` target, it will automatically submit coverage results to CDash as well.

Doing memory checking running the enabled tests with Valgrind requires that you set `CTEST_DO_MEMORY_TESTING=TRUE` with the 'env' command when running the `dashboard` target as:

```
$ env CTEST_DO_MEMORY_TESTING=TRUE make dashboard
```

but also note that you may also need to set the valgrind command and options with:

```
$ env CTEST_DO_MEMORY_TESTING=TRUE \
  CTEST_MEMORYCHECK_COMMAND=<abs-path-to-valgrind> \
  CTEST_MEMORYCHECK_COMMAND_OPTIONS="-q --trace-children=yes --tool=memcheck \
   --leak-check=yes --workaround-gcc296-bugs=yes \
   --num-callers=50 --suppressions=<abs-path-to-supp-file1> \
   ... --suppressions=<abs-path-to-supp-fileN>" \
  make dashboard
```

The CMake cache variable `<Project>_DASHBOARD_CTEST_ARGS` can be set on the cmake configure line in order to pass additional arguments to `ctest -S` when invoking the package-by-package CTest driver. For example:

```
-D<Project>_DASHBOARD_CTEST_ARGS="-VV"
```

will set verbose output with CTest.

Also note that one can submit results to a second CDash site as well by setting:

```
$ env \
  TRIBITS_2ND_CTEST_DROP_SITE=<second-site> \
  TRIBITS_2ND_CTEST_DROP_LOCATION=<second-location> \
  ... \
  make dashboard
```

If left the same as `CTEST_DROP_SITE` or `CTEST_DROP_LOCATION`, then `TRIBITS_2ND_CTEST_DROP_SITE` and `TRIBITS_2ND_CTEST_DROP_LOCATION` can be left empty "" and the defaults will be used. However, the user must set at least one of these variables to non-empty in order to trigger the second submit. For example, to submit to an experimental CDash site on the same machine, one would run:

```
$ env TRIBITS_2ND_CTEST_DROP_LOCATION="/testing/cdash/submit.php?project=<Project>" \
  ... \
  make dashboard
```

and `TRIBITS_2ND_CTEST_DROP_SITE` would be used for `CTEST_DROP_SITE`. This is a common use case when upgrading to a new CDash installation or testing new features for CDash before impacting the existing CDash site.

Finally, note in package-by-package mode (i.e. `<Project>_CTEST_DO_ALL_AT_ONCE=FALSE`) that if one kills the `make dashboard` target before it completes, then one must reconfigure from scratch in order to get the build directory back into the same

state before the command was run. This is because the `dashboard` target in package-by-package mode must first reconfigure the project with no enabled packages before it does the package-by-package configure/build/test/submit which enables each package one at a time. After the package-by-package configure/build/test/submit cycles are complete, then the project is reconfigured with the original set of package enables and returned to the original configure state. Even with the all-at-once mode, if one kills the `make dashboard` command before the reconfigure completes, one may be left with an invalid configuration of the project. In these cases, one may need to configure from scratch to get back to the original state before calling `make dashboard`.