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

Author: Roscoe A. Bartlett
Contact: bartlett.roscoe@gmail.com
Date: 2014-10-08
Version: TriBITS Development at 2014-10-08

Abstract: This document is generated from the generic template body document `TribitsBuildQickRefBody.rst` and provides a general project-independent quick 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-quickref.sh` in this directory which just runs:

```
$ ./create-project-build-quickref.py \
  --project-name="<Project>" \
  --project-template-file=TribitsBuildQuickRefTemplate.rst \
  --file-base=TribitsBuildQuickRef
```

to generate this document. In a project-specific version, `<Project>` is replaced with the actual project name (e.g. `Trilinos`, or `TriBITSProj`, etc.). 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, then install CMake with:

```
$ $TRIBITS_BASE_DIR/python/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.

Getting help for installing CMake with this script:

```
$ $TRIBITS_BASE_DIR/python/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).

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 Generator)

While CMake supports a number of different build generators (e.g. Eclipes, XCode, MS Visual Studio, etc.) the primary generator most people use on Unix/Linix system is make and CMake generates exceptional Makefiles. The materila in this section, while not exclusing to the makefile generator this should be assumed as the default.

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 seprate from the source tree. The build tree can be created under the source tree such as with:

```
$ $SOURCE_DIR
$ mkdir <SOME_BUILD_DIR>
$ cd <SOME_BUILD_DIR>
```

but it is generally recommended to create a build directory parallel from the soruce tree.

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

```
EXTRA_ARGS=$@

cmake \
  -D CMAKE_BUILD_TYPE:STRING=DEBUG \
  -D <Project>_ENABLE_TESTS:BOOL=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/*cmake* 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 CASL 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:FILEPATH=MyConfigureOptions.cmake \
  -D <Project>_ENABLE_TESTS:BOOL=ON \
  $EXTRA_ARGS \
  ${SOURCE_BASE}
```

where MyConfigureOptions.cmake might look like:

```
SET(CMAKE_BUILD_TYPE DEBUG CACHE STRING "" FORCE)
SET(<Project>_ENABLE_CHECKED_STL ON CACHE BOOL "" FORCE)
SET(BUILD_SHARED_LIBS ON CACHE BOOL "" FORCE)
...
```

Using a configuration fragment file allows for better reuse of configure options across different configure scripts and better version control of configure options.

NOTE: You can actually pass in a list of configuration fragment files which will be read in the order they are given.

NOTE: If you do not use 'FORCE' shown above, then the option can be overridden on the cmake command line with -D options. Also, if you don't use 'FORCE' then the option will not be set if it is already set in the case (e.g. by another configuration fragment file prior in the list).

c) Using ccmake to configure

```
$ ccmake $SOURCE_BASE
```

d) 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> 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 disbled). 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 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>:BOOL=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 in a project will be printed in the `cmake` STDOUT by setting:

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

This will print the basic backward dependencies for each SE package. To also see the direct forward dependencies for each SE package, also include:

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

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>:BOOL=ON \
-D <Project>_ENABLE_ALL_OPTIONAL_PACKAGES:BOOL=ON \
-D <Project>_ENABLE_TESTS:BOOL=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>:BOOL=ON \
-D <Project>_ENABLE_<TRIBITS_PACKAGE><B>:BOOL=ON \
 ...
```

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

4.3.4 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>:BOOL=ON \
-D <Project>_ENABLE_ALL_FORWARD_DEP_PACKAGES:BOOL=ON \
-D <Project>_ENABLE_TESTS:BOOL=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.

4.3.5 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:BOOL=ON \
-D <Project>_ENABLE_TESTS:BOOL=ON \
```

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

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

4.3.6 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>:BOOL=OFF
```

For example:

```
-D <Project>_ENABLE_<PACKAGE_A>:BOOL=ON \
-D <Project>_ENABLE_ALL_OPTIONAL_PACKAGES:BOOL=ON \
-D <Project>_ENABLE_<PACKAGE_B>:BOOL=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>:BOOL=OFF \
-D <Project>_ENABLE_<TRIBITS_PACKAGE><B>:BOOL=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.

4.3.7 Remove all package enables in the cache

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

```
$ ./-do-confiugre -D <Project>_UNENABLE_ENABLED_PACKAGES:BOOL=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 <Project> TriBITS CMake build system offers the ability to tweak the built-in CMake approach for setting compiler flags. 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"`). 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.

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 langauge 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`.

a) Configuring to build with default debug or release compiler flags:

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

   ```
   -D CMAKE_BUILD_TYPE:STRING=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:STRING=RELEASE
   ```

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

b) Adding arbitrary compiler flags but keeping other default 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:STRING="<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.

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

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

The package-specific options get appened 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, this means that users setting the CMAKE_<LANG>_FLAGS and <TRIBITS_PACKAGE>_<LANG>_FLAGS will *not* override the flags in CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> 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_TYPE> 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>! To override those options, see CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>_OVERRIDE.

c) 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:STRING="<OPTIONS_TO_OVERRIDE>"
```

For example, to default debug options use:

```
-D CMAKE_C_FLAGS_DEBUG_OVERRIDE:STRING="-g -O1" \
-D CMAKE_CXX_FLAGS_DEBUG_OVERRIDE:STRING="-g -O1"
```

and to override default release options use:

```
-D CMAKE_C_FLAGS_RELEASE_OVERRIDE:STRING="-O3 -funroll-loops" \
-D CMAKE_CXX_FLAGS_RELEASE_OVERRIDE:STRING="-O3 -fexceptions"
```

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.

d) 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:STRING="<EXTRA_LINK_LIBRARIES>" \
-DCMAKE_EXE_LINKER_FLAGS:STRING="<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.

e) Turning off strong warnings for individual packages:

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

```
-D <TRIBITS_PACKAGE>_DISABLE_STRONG_WARNINGS:BOOL=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==ON`). A release of <Project> should therefore not have strong warning options enabled.

f) 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:STRING="-O3 -funroll-loops" \
-D CMAKE_CXX_FLAGS:STRING="-O3 -fexceptions" \
-D CMAKE_BUILD_TYPE:STRING=NONE \
-D <Project>_ENABLE_STRONG_C_COMPILE_WARNINGS:BOOL=OFF \
-D <Project>_ENABLE_STRONG_CXX_COMPILE_WARNINGS:BOOL=OFF \
-D <Project>_ENABLE_SHADOW_WARNINGS:BOOL=OFF \
-D <Project>_ENABLE_COVERAGE_TESTING:BOOL=OFF \
-D <Project>_ENABLE_CHECKED_STL:BOOL=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.

g) 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:BOOL=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:BOOL=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.

h) 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 Teuchos, set the following when configuring:

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

i) Adding debug symbols to the build:

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

```
-D <Project>_ENABLE_DEBUG_SYMBOLS:BOOL=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 C++11

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

```
-D <Project>_ENABLE_CXX11:BOOL=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:STRING="<compiler flags>"
```

4.6 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:BOOL=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.

4.7 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:STRING=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:BOOL=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.

4.8 Configuring with MPI support

To enable MPI support you must minimally set:

```
-D TPL_ENABLE_MPI:BOOL=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="path"
```

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

```
-D MPI_BIN_DIR:PATH="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.

The value of `LD_LIBRARY_PATH` will also automatically be set to `${MPI_BASE_DIR}/lib` if it exists. This is needed for the basic compiler tests for some MPI implementations that are installed in non-standard locations.

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 that will try to find the right 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. If this is just the name of the program it will be looked for in ${MPI_BIN_DIR} and in other standard locations with that name. If this is an absolute path, then this will be used as CMAKE_[C,CXX,Fortran]_COMPILER to compile and link code.

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:BOOL=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:STRING="$EXTRA_COMPILE_FLAGS"
```

The link and library flags must be set through:

```
-D <Project>_EXTRA_LINK_FLAGS:STRING="$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:STRING=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:STRING=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:STRING=-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:STRING="arg1;arg2;...;argn"
```

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

```
-D MPI_EXEC_POST_NUMPROCS_FLAGS:STRING="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 seprated by `';'` as these variables are interpreted as CMake arrays.

4.9 Configuring for OpenMP support

To enable OpenMP support, one must set:

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

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

4.10 Building shared libraries

To configure to build shared libraries, set:

```
-D BUILD_SHARED_LIBS:BOOL=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).

4.11 Building static libraries and executables

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

```
-D BUILD_SHARED_LIBS:BOOL=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:BOOL=OFF \
-D TPL_FIND_SHARED_LIBS:BOOL=OFF \
-D <Project>_LINK_SEARCH_START_STATIC:BOOL=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 autodetection 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.

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

To enable a given TPL, set:

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

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

The headers, libraries, and library directories can then be specified with the input cache variables:

- `<TPLNAME>_INCLUDE_DIRS:PATH`: List of paths to the header include directories. For example:

  ```
  -D Boost_INCLUDE_DIRS:PATH=/usr/local/boost/include
  ```

- `<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:

  ```
  -D BLAS_LIBRARY_NAMES:STRING="blas;gfortran"
  ```

- `<TPLNAME>_LIBRARY_DIRS:PATH`: The list of directories where the library files can be found. For example:

  ```
  -D BLAS_LIBRARY_DIRS:PATH=/usr/local/blas
  ```

The variables `TPL_<TPLNAME>_INCLUDE_DIRS` and `TPL_<TPLNAME>_LIBRARIES` are what are directly used by the TriBITS dependency infrastructure. These variables are normally set by the variables `<TPLNAME>_INCLUDE_DIRS`, `<TPLNAME>_LIBRARY_NAMES`, and `<TPLNAME>_LIBRARY_DIRS` using CMake `find` commands but one can always override these by directly setting these cache variables `TPL_<TPLNAME>_INCLUDE_DIRS` and `TPL_<TPLNAME>_LIBRARIES`, for example, as:

```
-D TPL_Boost_INCLUDE_DIRS=/usr/local/boost/include \
-D TPL_Boost_LIBRARIES="/user/local/boost/lib/libprogram_options.a;..."
```

This gives the user complete and direct control in specifying exactly what is used in the build process. The other variables that start with `<TPLNAME>_` are just a convenience to make it easier to specify the location of the libraries.

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

```
-D BLAS_LIBRARY_NAMES:STRING=""
```

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

```
-D <TRIBITS_PACKAGE>_ENABLE_<TPLNAME>:BOOL=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>:BOOL=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.

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

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

This is not compatible with proper CMake usage and it not guaranteed to be supported.

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

Disabling a TPL explicitly can be done using:

```
-D TPL_ENABLE_<TPLNAME>:BOOL=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.

4.14 Disabling tentatively enabled TPLs

To disable a tentatively enabled TPL, set:

```
-D TPL_ENABLE_<TPLNAME>:BOOL=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.

4.15 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:BOOL=ON
```

This will cause TriBITS to print out a trace for all of the project's, repositorie'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:**

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

This produces a *lot* of output but can be very useful when debugging configuration problems.

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

```
-D CMAKE_VERBOSE_MAKEFILE:BOOL=TRUE
```

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

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

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

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

4.16 Enabling/disabling deprecated warnings

To turn off all deprecated warnings, set:

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

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

4.17 Disabling deprecated code

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

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

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

4.18 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.

The filepath for <Project>PackageDependencies.xml can be overridden using:

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

The filepath for <Project>PackageDependenciesTable.html can be overridden using:

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

The filepath for CDashSubprojectDependencies.xml can be overridden 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.

4.19 Enabling different test categories

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

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

Valid categories include `BASIC`, `CONTINUOUS`, `NIGHTLY`, `WEEKLY` 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. `WEEKLY` tests are for more expensive tests that are run approximately weekly. `PERFORMANCE` tests a special category used only for performance testing.

4.20 Disabling specific tests

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

```
-D <fullTestName>_DISABLE:BOOL=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.

4.21 Setting test timeouts at configure time

A maximum default time limit for any single test can be set at configure time by setting:

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

where `<maxSeconds>` is the number of wall-clock seconds. By default there is no timeout limit so it is a good idea to set some limit just so tests don't hang and run forever. When an MPI code has a defect, it can easily hang forever until it is manually killed. If killed, CTest will kill all of this child processes correctly.

NOTES:

- Be careful not set the 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.
- Individual tests can have there timeout limit increased 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()`).
- To set or override the test timeout limit at runtime, see Overridding test timeouts.

4.22 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_TESTING_TIMEOUT:STRING=<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 test using valgrind (see Running memory checking).

NOTES:

- 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 is used so `1.57` is truncated to `1.5` before scaling the test timeouts.
- The cache value of the variable `DART_TESTING_TIMEOUT` is not changed in the CMake cache file. Only the value of the timeout written into the DartConfiguration.tcl file will be scaled.

4.23 Enabling support for coverage testing

To turn on support for coverage testing set:

```
-D <Project>_ENABLE_COVERAGE_TESTING:BOOL=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

4.24 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:BOOL=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.

4.25 Enabling extra repositories with add-on packages:

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

```
-D<Project>_EXTRA_REPOSITORIES:STRING="<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:STRING=SomeExtraRepo \
   [Other Options]
```

After that, all of the extra packages defined in `SomeExtraRepo` will appear in the list of official <Project> packages and you are free to enable any of the defined add-on packages that you would like just like any other <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.

4.26 Enabling extra repositories through a file

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

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

Specifing 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 developers develops or tests with 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 repostories are defined by default unless <Project>_ENABLE_KNOWN_EXTERNAL_REPOS_TYPE is specifically set to one of the allowed values.

4.27 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.

4.28 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 managling determination, and other CMake-specific information.

4.29 Adding configure timers

To add timers to various configure steps, configure with:

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

4.30 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:BOOL=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:BOOL=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.

4.31 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:BOOL=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.

4.32 CMake configure-time development mode and debug checking

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

```
-D <Project>_ENABLE_DEVELOPMENT_MODE:BOOL=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. 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:BOOL=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.

5 Building (Makefile generator)

This section described building using the default CMake Makefile generator. TriBITS supports other CMake generators such as Visual Studio on Windows, XCode on Macs, and Eclipe project files but using those build systems are not documented here.

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 `CMakeList.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 type 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.

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 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.

6.1 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 ballance 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).

6.2 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 online documentation, and experiment for more details.

6.3 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 usally 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.

6.4 Overridding test timeouts

The configured 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`.

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

6.5 Running memory checking

To run the memory tests for just a single package, from the *base* build directory, run:

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

Detailed output form the memory checker (i.e. 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 build directory and then use `-L <TRIBITS_PACKAGE>` or any CTest test filtering command you would like.

7 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`. The other commands described below can all be run after the build and testing is complete.

7.1 Setting the install prefix at configure time

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:PATH=$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:STRING="include" \
-D <Project>_INSTALL_LIB_DIR:STRING="lib" \
-D <Project>_INSTALL_RUNTIME_DIR:STRING="bin" \
-D <Project>_INSTALL_EXAMPLE_DIR:STRING="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 abritrary location use:

```
-D <Project>_INSTALL_INCLUDE_DIR:PATH="/usr/trilinos_include" \
-D <Project>_INSTALL_LIB_DIR:PATH="/usr/trilinos_lib" \
-D <Project>_INSTALL_RUNTIME_DIR:PATH="/usr/trilinos_bin" \
-D <Project>_INSTALL_EXAMPLE_DIR:PATH="/usr/share/trilinos/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.

7.2 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:BOOL=ON
```

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.

7.3 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. However, it is advanced 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.

8 Packaging

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

8.1 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:BOOL=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 expresions that will be used to match files to exclude, set:

```
-D <Project>_DUMP_CPACK_SOURCE_IGNORE_FILES:BOOL=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:STRING="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:BOOL=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`.)

9 Dashboard submissions

You can use the TriBITS scripting code to submit package-by-package build, test, coverage, memcheck results to the project's CDash dashboard.

First, configure as normal but add the build and test parallel levels with:

```
-DCTEST_BUILD_FLAGS:STRING=-j4 -DCTEST_PARALLEL_LEVEL:STRING=4
```

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

```
$ make dashboard
```

This invokes the advanced TriBITS CTest scripts to do an experimental build for all of the packages that you have explicitly enabled. The packages that are implicitly enabled due to package dependencies are not directly processed by the experimental_build_test.cmake script.

There are a number of options that you can set in the environment to control what this script does. This set of options can be found by doing:

```
$ grep 'SET_DEFAULT_AND_FROM_ENV(' \
    <Project>/cmake/tribits/ctest/TribitsCTestDriverCore.cmake
```

Currently, this options includes:

```
SET_DEFAULT_AND_FROM_ENV( CTEST_TEST_TYPE Nightly )
SET_DEFAULT_AND_FROM_ENV(<Project>_TRACK "")
SET_DEFAULT_AND_FROM_ENV( CTEST_SITE ${CTEST_SITE_DEFAULT} )
SET_DEFAULT_AND_FROM_ENV( CTEST_DASHBOARD_ROOT "" )
SET_DEFAULT_AND_FROM_ENV( BUILD_TYPE NONE )
SET_DEFAULT_AND_FROM_ENV(COMPILER_VERSION UNKNOWN)
SET_DEFAULT_AND_FROM_ENV( CTEST_BUILD_NAME
SET_DEFAULT_AND_FROM_ENV( CTEST_START_WITH_EMPTY_BINARY_DIRECTORY TRUE )
SET_DEFAULT_AND_FROM_ENV( CTEST_WIPE_CACHE TRUE )
SET_DEFAULT_AND_FROM_ENV( CTEST_CMAKE_GENERATOR ${DEFAULT_GENERATOR})
SET_DEFAULT_AND_FROM_ENV( CTEST_DO_UPDATES TRUE )
SET_DEFAULT_AND_FROM_ENV( CTEST_GENERATE_DEPS_XML_OUTPUT_FILE FALSE )
SET_DEFAULT_AND_FROM_ENV( CTEST_UPDATE_ARGS "")
SET_DEFAULT_AND_FROM_ENV( CTEST_UPDATE_OPTIONS "")
SET_DEFAULT_AND_FROM_ENV( CTEST_BUILD_FLAGS "-j2")
SET_DEFAULT_AND_FROM_ENV( CTEST_DO_BUILD TRUE )
SET_DEFAULT_AND_FROM_ENV( CTEST_DO_TEST TRUE )
SET_DEFAULT_AND_FROM_ENV( MPI_EXEC_MAX_NUMPROCS 4 )
SET_DEFAULT_AND_FROM_ENV( CTEST_PARALLEL_LEVEL 1 )
SET_DEFAULT_AND_FROM_ENV( <Project>_WARNINGS_AS_ERRORS_FLAGS "" )
SET_DEFAULT_AND_FROM_ENV( CTEST_DO_COVERAGE_TESTING FALSE )
SET_DEFAULT_AND_FROM_ENV( CTEST_COVERAGE_COMMAND gcov )
SET_DEFAULT_AND_FROM_ENV( CTEST_DO_MEMORY_TESTING FALSE )
SET_DEFAULT_AND_FROM_ENV( CTEST_MEMORYCHECK_COMMAND valgrind )
SET_DEFAULT_AND_FROM_ENV( CTEST_DO_SUBMIT TRUE )
SET_DEFAULT_AND_FROM_ENV( <Project>_ENABLE_SECONDARY_TESTED_CODE OFF )
SET_DEFAULT_AND_FROM_ENV( <Project>_ADDITIONAL_PACKAGES "" )
SET_DEFAULT_AND_FROM_ENV( <Project>_EXCLUDE_PACKAGES "" )
SET_DEFAULT_AND_FROM_ENV( <Project>_BRANCH "" )
SET_DEFAULT_AND_FROM_ENV( <Project>_REPOSITORY_LOCATION "software.sandia.gov:/space/g
SET_DEFAULT_AND_FROM_ENV( <Project>_PACKAGES "${<Project>_PACKAGES_DEFAULT}" )
SET_DEFAULT_AND_FROM_ENV( CTEST_SELECT_MODIFIED_PACKAGES_ONLY OFF )
```

For example, to run an experimental build and in the process change the build name and the options to pass to 'make', use:

```
$ 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.

NOTE: It is useful to set CTEST_BUILD_NAME to some unique name to make it easier to find your results in the CDash dashboard.

NOTE: A number of the defaults set in TribitsCTestDriverCore.cmake are overridden from experimental_build_test.cmake (such as CTEST_TEST_TYPE=Experimental) so you will want to look at experimental_build_test.cmake to see how these are changed. The script experimental_build_test.cmake sets reasonable values for these options in order to use the 'make dashboard' target in iterative development for experimental builds.

NOTE: The target 'dashboard' is not directly related to the built-in CMake targets 'Experimental*' that run standard dashboards with CTest without the custom package-by-package driver in TribitsCTestDriverCore.cmake. The package-by-package extended CTest driver is more appropriate for <Project>.

NOTE: Once you configure with -D<Project>_ENABLE_COVERAGE_TESTING:BOOL=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.

NOTE: Doing a memory check with Valgrind requires that you set CTEST_DO_MEMORY_TESTING= with the 'env' command as:

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

NOTE: 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:STRING="-VV"
```

will set verbose output with CTest.