TriBITS Developers Guide and Reference

Author: Roscoe A. Bartlett (bartlettra@ornl.gov)

Abstract

This document describes the usage of TriBITS to build, test, and deploy complex software. The primary audience are those individuals who develop on a software project which uses TriBITS. The overall structure of a TriBITS project is described including all of the various project- and package-specific files that TriBITS requires or can use and how and what order these files are processed. It also contains detailed reference information on all of the various TriBITS macros and functions directly used in TriBITS project CMake files. Many other topics of interest to a TriBITS project developer and architect are discussed as well.

Contents

1 Introduction

This document describes the usage of the TriBITS (Tribal Build, Integration, Test System) to develop software projects. An initial overview of TriBITS is provided in the TriBITS Overview document which contains the big picture and provides a high-level road map to what TriBITS provides. This particular document, however, describes the details on how to use the TriBITS system to create a CMake build system for a set of compiled software packages. Also described are an extended set of tools and processes to create a complete software development, testing, and deployment environment consistent with modern agile software development best practices.

TriBITS is a fairly extensive framework that is built on top of the open-source CMake/CTest/CPack/CDash system (which in itself is a very extensive system of software and tools). The most important thing to remember is that a software project that use TriBITS are really just a CMake project. TriBITS makes no attempt to hide that fact either from the TriBITS project developers or from the users that need to configure and build the software. Therefore, to make effective usage of TriBITS, one must learn the basics of CMake (both as a developer and as a user). In particular, CMake is a Turning-complete programming language with local and global variables (with strange scoping rules), macros, functions, targets, commands, and other features. One needs to understand how to define and use variables, macros, and functions in CMake. One needs to know how to debug CMakeLists.txt files and CMake code in general (i.e. using `MESSAGE()` print statements). One needs to understand how CMake defines and uses targets for various qualities like libraries, executables, etc. Without this basic understanding of CMake, one will have trouble resolving problems when they might occur.

The remainder of this documented is structured as follows. First, there is some additional Background material provided. Then, a detailed specification of TriBITS Project Structure is given which lists and defines all of the files that a TriBITS project contains and how they are processed. This is followed up by short descriptions of Example TriBITS Projects that are provided with the TriBITS source tree that are used throughout this document. The topic of Package Dependencies and Enable/Disable Logic is then discussed which is the backbone of the TriBITS system. An overview of the foundations for TriBITS Automated Testing is then given. The topic of TriBITS Multi-Repository Support is examined next. Development Workflows using TriBITS is then explored. This is followed by a set of detailed Howtos. Later some Additional Topics are presented that don't fit well into other sections. Then the main bulk of the detailed reference material for TriBITS is given in the section TriBITS Detailed Reference Documentation. Finally, several bits of information are provided in the Appendix.

2 Background

Before diving into the details about TriBITS in the following sections, first some more background is in order. First, a discussion of TriBITS Developer and User Roles is provided to help the reader identify their own role(s) and to help guide the reader to the appropriate documentation (which may or may not primarily be in this document). Then, section CMake Language Overview and Gotchas tries to orient readers with little to no CMake knowledge or experience on where to start and provides some warnings about non-obvious CMake behavior that often trips up new users of TriBITS.

2.1 TriBITS Developer and User Roles

There are approximately five different types roles related to TriBITS. These different roles require different levels of expertise and knowledge of CMake and knowledge of the TriBITS system. The primary roles are 1) TriBITS Project User, 2) TriBITS Project Developer, 3) TriBITS Project Architect, 4) TriBITS System Developer, and 5) TriBITS System Architect. Each of these roles builds on the necessary knowledge of the lower-level roles.

The first role is that of a **TriBITS Project User** who only needs to be able to configure, build, and test a project that uses TriBITS as its build system. A person acting in this role needs to know little about CMake other than basics about how to run the `cmake` and `ctest` exectuables, how to set CMake cache variables, and the basics of building software by typing `make` and running tests with `ctest`. The proper reference for a TriBITS Project User is the Project-Specific Build Quick Reference. The TriBITS Overview document may also be of some help. A TriBITS project user does not need to know anything about the CMake language itself or any of the TriBITS macros or functions described in TriBITS Macros and Functions or really anything else described in this current document except for Package Dependencies and Enable/Disable Logic.

A **TriBITS Project Developer** is someone who contributes to a software project that uses TriBITS. They will add source files, libraries and exectuables, test executables and define tests run with `ctest`. They have to configure and build the project code in order to be able to develop and run tests and therefore this role includes all of the necessary knowledge and functions of a TriBITS Project User. A casual TriBITS Project Developer typically does not need to know a lot about CMake and really only needs to know a subset of the TriBITS Macros and Functions defined in this document in addition to the genetic TriBITS Build Quick Reference document. A slightly more sophisticated TriBITS Project Developer will also add new packages, add new package dependencies, and define new TPLs. This current TriBITS Developers Guide and Reference document should supply everything such a developer needs to know and more. Only a smaller part of this document needs to be understood and accessed by people assuming this role.

The next level of roles is a **TriBITS Project Architect**. This is someone (perhaps only one person on a project development team) that knows the usage and functioning of TriBITS in great detail. They understand how to set up a TriBITS project from scratch, how to set up automated testing using the TriBITS system, and know how to use TriBITS to implement the overall software development process. A person in this role is also likely to be the one who makes the initial technical decision for their project to adopt TriBITS for its native build and test system. This document (along with detailed CMake/CTest/CDash documentation provided by Kitware and the larger community) should provide most of what a person in this role needs to know. A person assuming this role is the primary audience for a lot of the more advanced material in this document.

The last two roles **TriBITS System Developer** and **TriBITS System Architect** are for those individuals that actually extend and modify the TriBITS system itself. A TriBITS System Developer needs to know how to add new TriBITS functionality while maintaining backward compatibility, know how to add new unit tests for the TriBITS system (see The TriBITS Test Package), and perform other related tasks. Such a developer needs to be very knowledgeable of the basic functioning of CMake and know how TriBITS is implemented in the CMake language. A TriBITS System Architect is someone who must be consulted on almost all non-trivial changes or additions to the TriBITS system. A *TriBITS System Architect* in addition needs to know the entire TriBITS system, the design philosophy that provides the foundation for TriBITS and be an expert in CMake, CTest, and CDash. Everything that needs to be known by a TriBITS System Developer and a TriBITS System Architect is not contained in this document. Instead, the primary documentation will be in the TriBITS CMake source code and various unit tests itself defined in The TriBITS Test Package. At the time of this writing, there is currently there is only one TriBITS System Architect (who also happens to be the primary author of this document).

An explicit goal of this document is to make new TriBITS Project Architects (i.e. those who would make the decision to adopt TriBITS for their projects), and new TriBITS System Developers to help extend and maintain TriBITS. As TriBITS matures and its requirements further stabilize, the need for a TriBITS System Architect will hopefully be diminished.

So depending on the particular role that a reader falls into, this document may or may not be necessary but instead the TriBITS Overview or the <Project>BuildQuickRef documents may be more appropriate. Hopefully the above roles and discussion will help the reader select the right document to start with.

2.2 CMake Language Overview and Gotchas

TriBITS removes a lot of the boiler plate code needed to write a CMake project. As a result, many people can come into a project that uses TriBITS and quickly start to contribute by adding new source files, adding new libraries, adding new tests, and even adding new TriBITS packages and TPLs; all without really having learned anything about CMake. One just needs to copy-and-paste existing example CMake code and files as basically "monkey see, monkey do". As long as nothing out of the ordinary happens, many people can get along just fine in this mode for a time.

However, we have observed that most mistakes that people make when using TriBITS, and most of the problems they have when using the system, are due to a basic lack of knowledge of the CMake language. One can find basic tutorials and references on the CMake language in various locations online for free. One can also purchase the official CMake reference book. Also, documentation for any built-in CMake command is available locally by running:

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

Because tutorials and detailed documentation for the CMake language already exists, this document will not even attempt to provide a first reference to CMake (which is a large topic in itself). However, what we try to provide below is a short overview of the more quirky or surprising aspects of the CMake language that an programmer experienced in another language might get tripped up or surprised by. Some of the more unique features of the language are described in order to help avoid some of these common mistakes and provide greater understanding of how TriBITS works.

The CMake language that is used to write CMake projects with TriBITS (and that core TriBITS functionality itself is implemented in, see TriBITS System Project Dependencies) is a fairly simple programming language with fairly simple rules (for the most part). However, compared to other programming languages, there are a few peculiar aspects to the CMake language like strange variable scoping rules, and how arguments are passed to macros and functions, that can make working with it difficult if you don't understand these. Also, CMake has some interesting gotchas. In order to effectively use TriBITS (or just raw CMake) to construct and maintain a project's CMake files, one must know the basic rules of CMake and be aware of these gotchas.

The first thing to understand about the CMake language is that nearly every line of CMake code is just a command taking a string (or an array of strings) and functions that operate on strings. An array argument is just a single string literal with elements separated by semi-colons `"<str0>;<str1>;..."`. CMake is a bit odd in how it deals with these arrays (which are just represented as a string with elements separated with semi-colons `';'`). For example, all of the following are equivalent and pass in a CMake array with 3 elements [`A`], [`B`], and [`C`]:

```
SOME_FUNC(A B C)
SOME_FUNC("A" "B" "C")
SOME_FUNC("A;B;C")
```

However, the above is *not* the same as:

```
SOME_FUNC("A B C")
```

which just passes in a single element with value [`A B C`]. Raw quotes in CMake basically escape the interpretation of space characters as array element boundaries. Quotes around arguments with no spaces does nothing (as seen above, except for the interpretation as variable names in an `IF()` statement). In order to get a quote char [`"`] into string, you must escape it as:

```
SOME_FUNC(\"A\")
```

which passes an array with the single argument [`\"A\"`].

Variables are set using the built-in CMake `SET()` command that just takes string arguments like:

```
SET(SOME_VARIABLE "some_value")
```

In CMake, the above is identical, in every way, to:

```
SET(SOME_VARIABLE some_value)
SET("SOME_VARIABLE;"some_value")
SET("SOME_VARIABLE;some_value")
```

The function `SET()` simply interprets the first argument to as the name of a variable to set in the local scope. Many other built-in and user-defined CMake functions work the same way. That is some of the string arguments are interpreted as the names of variables. There is no special language feature that interprets them as variables (except in an `IF()` statement).

However, CMake appears to parse arguments differently for built-in CMake control structure functions like `FOREACH()` and `IF()` and does not just interpret them as a string array. For example:

```
FOREACH (SOME_VAR "a;b;c")
  MESSAGE("SOME_VAR='${SOME_VAR}'")
ENDFOREACH()
```

prints `'SOME_VAR='a;b;c'` instead of printing `SOME_VAR='a'` followed by `SOME_VAR='b'`, etc., as you would otherwise expect. Therefore, this simple rule for the handling of function arguments as string arrays does not hold for CMake logic control commands. Just follow the CMake documentation for these control structures (i.e. see `cmake --help-command if` and `cmake --help-command foreach`).

CMake offers a rich assortment of built-in commands for doing all sorts of things. Two of these are the built-in `MACRO()` and the `FUNCTION()` commands which allow you to create user-defined macros and functions (which is what TriBITS is built on). All of these built-in and user-defined macros and functions work exactly the same way; they take in an array of string arguments. Some functions take in positional arguments but most actually take a combination of positional and keyword arguments (see PARSE_ARGUMENTS()).

Variable names are translated into their stored values using `${SOME_VARIABLE}`. The value that is extracted depends on if the variable is set in the local or global (cache) scope. The local scopes for CMake start in the base project directory in its base `CMakeLists.txt` file. Any variables that are created by macros in that base local scope are seen across an entire project but are *not* persistent across multiple successive `cmake` configure invocations where the cache file `CMakeCache.txt` is not deleted in between.

The handling of variables is one area where CMake is radically different from most other languages. First, a variable that is not defined simply returns nothing. What is surprising to most people about this is that it does not even return an empty string that would register as an array element! For example, the following set statement:

```
SET(SOME_VAR a ${SOME_UNDEFINED_VAR} c)
```

(where `SOME_UNDEFINED_VAR` is an undefined variable) produces `SOME_VAR='a;c'` and *not* `'a;;c'`! The same thing occurs when an empty variable is de-references such as with:

```
SET(EMPTY_VAR "")
SET(SOME_VAR a ${EMPTY_VAR} c)
```

which produces `SOME_VAR='a;c'` and *not* `'a;;c'`. In order to always produce an element in the array even if the variable is empty, one must quote the argument as with:

```
SET(EMPTY_VAR "")
SET(SOME_VAR a "${EMPTY_VAR}" c)
```

which produces `SOME_VAR='a;;c'`, or three elements as one might assume.

This is a common error that people make when they call CMake functions (built-in or TriBITS-defined) involving variables that might be undefined or empty. For example, for the macro:

```
MACRO(SOME_MACRO  A_ARG  B_ARG  C_ARG)
   ...
ENDMACRO()
```

if someone tries to call it with (misspelled variable?):

```
SOME_MACRO(a ${SOME_OHTER_VAR} c)
```

and if `SOME_OHTER_VAR=""` or if it is undefined, then CMake will error out with the error message saying that the macro `SOME_MACRO()` takes 3 arguments but only 2 were provided. If a variable might be empty but that is still a valid argument to the command, then it must be quoted as:

```
SOME_MACRO(a "${SOME_OHTER_VAR}" c)
```

Related to this problem is that if you misspell the name of a variable in a CMake `IF()` statement like:

```
IF (SOME_VARBLE)
  ...
ENDIF()
```

then it will always be false and the code inside the if statement will never be executed! To avoid this problem, use the utility function ASSERT_DEFINED() as:

```
ASSERT_DEFINED(SOME_VARBLE)
IF (SOME_VARBLE)
  ...
ENDIF()
```

In this case, the misspelled variable would be caught.

While on the subject of `IF()` statements, CMake has a strange convention. When you say:

```
IF (SOME_VAR)
  DO_SOMETHING()
ENDIF()
```

then `SOME_VAR` is interpreted as a variable and will be considered true and `DO_SOMETHING()` will be called if `${SOME_VAR}` does *not* evaluate to `0`, `OFF`, `NO`, `FALSE`, `N`, `IGNORE`, `""`, or ends in the suffix `-NOTFOUND`. How about that for a true/false rule! To be safe, use `ON/OFF` and `TRUE/FALSE` pairs for setting variables. Look up native CMake documentation on `IF()` for all the interesting details and all the magical things it can do.

WARNING: If you mistype `"ON"` as `"NO"`, it evaluates to `FALSE`/`OFF`! (That is a fun defect to track down!)

CMake language behavior with respect to case sensitivity is also strange:

- Calls of built-in and user-defined macros and functions is *case insensitive*! That is `set(...)`, `SET(...)`, `Set()`, and all other combinations of upper and lower case characters for 'S', 'E', 'T' all call the built-in `SET()` function. The convention in TriBITS is to use all caps for functions and macros (which was adopted by following the conventions used in the early versions of TriBITS, see the History of TriBITS). The convention in CMake literature from Kitware seems to use lower-case letters for functions and macros.
- However, the names of CMake (local or cache/global) variables are *case sensitive*! That is, `SOME_VAR` and `some_var` are *different* variables. Built-in CMake variables tend use all caps with underscores (e.g. `CMAKE_CURRENT_SOURCE_DIR`) but other built-in CMake variables tend to use mixed case with underscores (e.g. `CMAKE_Fortran_FLAGS`). TriBITS tends to use a similar naming convention where variables have mostly upper-case letters except for parts that are proper nouns like the project, package or TPL name (e.g. `TribitsProj_TRIBITS_DIR`, `TriBITS_SOURCE_DIR`, `Boost_INCLUDE_DIRS`).

I don't know of any other programming language that uses different case sensitivity rules for variables and functions. However, because we must parse macro and function arguments when writing user-defined macros and functions, it is a good thing that CMake variables are case sensitive. Case insensitivity would make it much harder and more expensive to parse argument lists that take keyword-based arguments (see PARSE_ARGUMENTS()).

Other mistakes that people make result from not understanding how CMake scopes variables and other entities. CMake defines a global scope (i.e. "cache" variables) and several nested local scopes that are created by `ADD_SUBDIRECTORY()` and entering FUNCTIONS. See DUAL_SCOPE_SET() for a short discussion of these scoping rules. And it is not just variables that can have local and global scoping rules. Other entities, like defines set with the built-in command `ADD_DEFINITIONS()` only apply to the local scope and child scopes. That means that if you call `ADD_DEFINITIONS()` to set a define that affects the meaning of a header-file in C or C++, for example, that definition will *not* carry over to a peer subdirectory and those definitions will not be set (see warning in Miscellaneous Notes (TRIBITS_ADD_LIBRARY())).

Now that some CMake basics and common gotchas have been reviewed, we now get into the meat of TriBITS starting with the overall structure of a TriBITS project in the next section.

3 TriBITS Project Structure

TriBITS is a framework, implemented in CMake to create CMake projects. As a framework, it defines the the overall structure of a CMake build system for a project and processes project, repository, and package specific files in a specified order. Almost all of this processing takes place in the TRIBITS_PROJECT() macro (or macros and functions it calls).

3.1 TriBITS Structural Units

A CMake project that uses TriBITS as its build and test system is composed of a single TriBITS Project, one or more TriBITS Repositories and one or more TriBITS Packages. In addition, a TriBITS Package can be broken up into TriBITS Subpackages. Together, the collection of TriBITS Packages and TriBITS Subpackages are called *TriBITS Software Engineering Packages*, or TriBITS SE Packages for short.

First, to establish the basic nomenclature, the key structural TriBITS units are:

- TriBITS Package: A collection of related software that typically includes one or more source files built into one or more libraries and has assoicated tests to help define and protect the functionality provided by the software. A package also typically defines a unit of documentation and testing (see TriBITS Automated Testing). A TriBITS package may or may not be broken down into multiple subpackages. Examples of TriBITS packages in `TribitsExampleProject` include `SimpleCXX`, `MixedLanguage` and `PackageWithSubpackages`. (Don't confuse a TriBTS "Package" with a raw CMake "Package" (see History of TriBITS). A raw CMake "Package" actually maps to a TriBITS TPL.)

- TriBITS Subpackage: A part of a parent TriBITS Package that also typically has source files built into libraries and tests but is documented and tested along with the other subpackages the parent package. The primary purpose for supporting subpackages to provide finer-grained of control software dependencies. In `TribitsExampleProject`, `PackageWithSubpackages` is an example of a package with subpackages `SubpackageA`, `SubpackaeB`, and `SubpackageC`. The full subpackage name has the parent package name prefixed the the subpackage name (e.g. `PackageWithSubpackagesSubpackageA`). The parent package is always implicitly dependent on its subpackages.

- **TriBITS SE Package**: The combined set of TriBITS Packages and TriBITS Subpackages that constitute the basice *Softare Engineering* packages (see ???) of a TriBITS proejct: SE packages are the basis for setting dependencies in the TriBITS system. For example, the SE Packages provided by the top-level example `PackageWithSubpackages` is (in order of increasing dependencies) `PackageWithSubpackagesSubpackageA`, `PackageWithSubpackagesSubpackaeB`, `PackageWithSubpackagesSubpackageC`, and `PackageWithSubpackages` (see TribitsExampleProject).

- TriBITS TPL: The specification for a particular external dependency that is required or can be used in one or more TriBITS SE Packages. A TPL (a Third Party Library) typically provides a list of libraries or a list include directories for header files but can also be manifisted in order ways as well. Examples of basic TPLs include `BLAS`, `LAPACK`, and `Boost`.

- TriBITS Repository: A collection of one or more TriBITS Packages specified in a <repoDir>/PackagesList.cmake file.

- TriBITS Project: A collection of TriBITS Repositories and TriBITS Packages that defines a CMake `PROJECT` defining soiftware which can be directly configured with `cmake` and then built, tested, and installed.

- **TriBITS Meta-Project**: A TriBITS Project that contains no TriBITS packages or TriBITS TPLs but is composed out of other TriBITS Repositories.

In this document, dependenices are described as either being *upstream* or *downstream/forward* as defined as:

- If unit "B" requires (or can use, or comes before) unit "A", then "A" is an **upstream dependency** of "B".

- If unit "B" requires (or can use, or comes before) unit "A", then "B" is an **downstream or forward dependency** of "A".

The following subsections define the major structural units of a TriBITS project in more detail. Each structural unit is described along with the files and directories assoicated with each. In addition, a key set of TriBITS CMake variables for each are defined as well.

In the next major section following this one, some Example TriBITS Projects are described. For those who just want to jump in and learn best by example, these exmaple projects are a good way to start. These example projects will be referenced in the more detailed descriptions given in this document.

The CMake varaibles defined by TriBITS described in the structural using below below fall into one of two types:

- *Local Fixed-Name Variables* are used temporarily in the processing of a TriBITS unit. These include variables such as `PROJECT_NAME`, `REPOSITORY_NAME`, `PACKAGE_NAME`, and `PARENT_PACKAGE_NAME` These are distinguished by having a fixed/constant name. They are typically part of TriBITS reflection system, allowing subordinate units to determine the encapsulating unit in which they are participating. For, example, a TriBITS subpackage can determine its name, its parent package's name and directories, its parent repository name and directories, and the enclosing project's name and directories.

- *Namespaced Variables* are used to refer to properties of a named TriBITS unit. These include variables such as `${REPOSTORY_NAME}_SOURCE_DIR` (e.g. `TribitsExProj_SOURCE_DIR`) and `${PACKAGE_NAME}_BINARY_DIR` (e.g. `SimpleCXX_BINARY_DIR`). They are available after processing the unit, for use by downstream or subordinate units. They are part of the TriBITS dependency system, allowing downstream units to access properties of their known upstream dependencies.

More information about these various files is described in section Processing of TriBITS Files: Ordering and Details.

TriBITS Project

A TriBITS Project:

- Defines a complete CMake project and calls `PROJECT(${PROJECT_NAME} ...)`.
- Consists of one or more TriBITS Repositories (see TriBITS Repository).
- Defines the `PROJECT_NAME` CMake variable (defined in <projectDir>/ProjectName.cmake)
- Defines a set of native Repositories (see below) that define packages and TPLs.
- Allows for extra Repositories to be added on before or after the set of native Repositories (specified in <projectDir>/cmake/ExtraRepositoriesList.cmake or by CMake variables)
- Defines a default CDash server and default project name on the server (the project name on the CDash server must be the same as `${PROJECT_NAME}`).

For more details on the definition of a TriBITS Project, see:

- TriBITS Project Core Files
- TriBITS Project Core Variables

TriBITS Project Core Files The core files making up a TriBITS Project (where `<projectDir> = ${PROJECT_SOURCE_DIR}`) are:

```
<projectDir>/
   ProjectName.cmake     # Defines PACAKGE_NAME
   CMakeLists.txt        # Base project CMakeLists.txt file
   CTestConfig.cmake     # [Optional] Needed for CDash submits
   Version.cmake         # [Optional] Dev mode, Project version, VC branch
   project-checkin-test-config.py # [Optional] checkin-test.py default builds
   cmake/
     NativeRepositoriesList.cmake    # [Optional] Used for some meta-projects
     ExtraRepositoriesList.cmake     # [Optional] Lists repos and VC URLs
     ProjectDependenciesSetup.cmake  # [Optional] Project deps overrides
     CallbackDefineProjectPackaging.cmake  # [Optional] CPack settings
     tribits/    # [Optional] Or provide ${PROJECT_NAME}_TRIBITS_DIR
     ctest/
       CTestCustom.cmake.in  # [Optional] Custom ctest settings
```

These TriBITS Project files are documented in more detail below:

- <projectDir>/ProjectName.cmake
- <projectDir>/CMakeLists.txt
- <projectDir>/CTestConfig.cmake
- <projectDir>/Version.cmake
- <projectDir>/project-checkin-test-config.py
- <projectDir>/cmake/NativeRepositoriesList.cmake

- <projectDir>/cmake/ExtraRepositoriesList.cmake
- <projectDir>/cmake/ProjectDependenciesSetup.cmake
- <projectDir>/cmake/CallbackDefineProjectPackaging.cmake
- <projectDir>/cmake/tribits/
- <projectDir>/cmake/ctest/CTestCustom.cmake.in

<projectDir>/ProjectName.cmake: [Required] At a minimum provides a `SET()` statement to set the local variable `PROJECT_NAME`. This file is the first file is read by a number of tools in order to get the TriBITS project's name. This file is read first in every context that involves processing the TriBITS project's files, including processes and tools that just need to build a package and TPL dependnecy tree (see Package Dependencies and Enable/Disable Logic). Being this is the first file read in for a TriBITS project and that it is read in first at the top level scope in every context, this is a good file to put other universal static project options in that affect dependency handling. Note that this is a project, not a repository file so no general repository-specific settings should go in this file. A simple example of this file is TribitsExampleProject/`PackageName.cmake`:

```
# Must set the project name at very beginning before including anything else
SET(PROJECT_NAME TribitsExProj)
```

<projectDir>/CMakeLists.txt: [Required] The top-level CMake project file. This is the first file that the `cmake` exectuable processes that starts everything off and is the base level scope for local CMake variables. Due to a number of CMake limitations and quarks, a project's top-level `CMakeLists.txt` file is not a clean as one might otherwise hope would be but it is not too bad. A simple, but representative, example is TribitsExampleProject/`CMakeLists.txt`:

```
############################################################################
#                                                                          #
#                        TribitsExampleProject                             #
#                                                                          #
############################################################################

# Make CMake set WIN32 with CYGWIN for older CMake versions.  CMake requires
# this to be in the top-level CMakeLists.txt file and not an include file :-(
SET(CMAKE_LEGACY_CYGWIN_WIN32 1 CACHE BOOL "" FORCE)

#
# A) Define your project name and set up major project options
#
# NOTE: Don't set options that would impact what packages get defined or
# enabled/disabled in this file as that would not impact other tools that
# don't process this file.
#

# Get PROJECT_NAME (must be in file for other parts of system to read)
INCLUDE("${CMAKE_CURRENT_SOURCE_DIR}/ProjectName.cmake")

# CMake requires that you declare the CMake project in the top-level file and
# not in an include file :-(
PROJECT(${PROJECT_NAME} NONE)

# Turn on export depenency generation for WrapExteranl package
SET(${PROJECT_NAME}_GENERATE_EXPORT_FILE_DEPENDENCIES_DEFAULT ON)

#
# B) Pull in the TriBITS system and execute
#
```

```
SET(${PROJECT_NAME}_TRIBITS_DIR  ""  CACHE  STRING "TriBITS base directory (required!)
INCLUDE("${${PROJECT_NAME}_TRIBITS_DIR}/TriBITS.cmake")

# CMake requires this be in the top file and not in an include file :-(
CMAKE_MINIMUM_REQUIRED(VERSION ${TRIBITS_CMAKE_MINIMUM_REQUIRED})

# Do all of the processing for this Tribits project
TRIBITS_PROJECT()
```

A couple of CMake and TriBITS quarks that that above example `CMakeLists.txt` addresses are worth some discussion. First, to avoid duplication, the project's `ProjectName.cmake` file is read in with an `INCLUDE()` that defines the local variable `PROJECT_NAME`. Right after this initial include, the built-in CMake command `PROJECT(${PROJECT_NAME} NONE)` is run. This command must be explicitly called with `NONE` so as to avoid default CMake behavior for defining compilers. The definition of compilers comes later as part of the TriBITS system inside of the TRIBITS_PROJECT() command (see Full Processing of TriBITS Project Files).

As noted in the above example file, the only project defaults that should be set in this top-level `CMakeLists.txt` file are those that do not impact the list of package enables/disables. The latter type of defaults should set in other files (see below).

In this example project, a CMake cache variable `${PROJECT_NAME}_TRIBITS_DIR` must be set by the user to define where the base `tribits` source directory is located. With this variable set (i.e. passed into `cmake` command-line use `-DTribitsExProj_TRIBITS_DIR=<someDir>`), one just includes a single file to pull in the TriBITS system:

```
INCLUDE("${${PROJECT_NAME}_TRIBITS_DIR}/TriBITS.cmake")
```

With the `TriBITS.cmake` file included, the configuration of the project using TriBITS occurs with a single call to TRIBITS_PROJECT().

Some projects, like Trilinos, actually snapshot the `tribits` directory into their source tree and therefore don't need to have this variable set. In Trilinos, the include line is just:

```
INCLUDE(${CMAKE_CURRENT_SOURCE_DIR}/cmake/tribits/TriBITS.cmake)
```

The minimum CMake version must also be delcared in the top-level `CMakeLists.txt` file as shown. Explicitly setting the minimum CMake version avoids strange errors that can occur when someone tries to build the project using a version of CMake that is too old. If the given project requires a version of CMake newer than what is required by TriBITS itelf (as defined in the variable `TRIBITS_CMAKE_MINIMUM_REQUIRED` which was set when the `TriBITS.cmake` file was included), then that version can be passed instead of using `${TRIBITS_CMAKE_MINIMUM_REQUIRED}` (the current minimum version of CMake required by TriBITS is given at in TribitsBuildQuickRef) . For example, the `VERA/CMakeLists.txt` file lists as its first line:

```
SET(VERA_TRIBITS_CMAKE_MINIMUM_REQUIRED 2.8.5)
CMAKE_MINIMUM_REQUIRED(VERSION ${VERA_TRIBITS_CMAKE_MINIMUM_REQUIRED})
```

<projectDir>/CTestConfig.cmake: [Optional] Specifies the CDash site and project to submit results to when doing an automated build driven by the CTest driver code in TRIBITS_CTEST_DRIVER() is used (see TriBITS Package-by-Package CTest/Dash Driver). This file is also required to use the TriBITS-generated `dashboard` target (see Dashboard Submissions). An example of this file is TribitsExampleProject/`CTestConfig.cmake`:

```
INCLUDE(SetDefaultAndFromEnv)

SET(CTEST_NIGHTLY_START_TIME "04:00:00 UTC") # 10 PM MDT or 9 PM MST

IF (NOT DEFINED CTEST_DROP_METHOD)
  SET_DEFAULT_AND_FROM_ENV(CTEST_DROP_METHOD "http")
ENDIF()

IF (CTEST_DROP_METHOD STREQUAL "http")
  SET_DEFAULT_AND_FROM_ENV(CTEST_DROP_SITE "casl-dev.ornl.gov")
```

```
  SET_DEFAULT_AND_FROM_ENV(CTEST_PROJECT_NAME "TribitsExProj")
  SET_DEFAULT_AND_FROM_ENV(CTEST_DROP_LOCATION "/cdash/submit.php?project=TribitsExPro
  SET_DEFAULT_AND_FROM_ENV(CTEST_TRIGGER_SITE "")
  SET_DEFAULT_AND_FROM_ENV(CTEST_DROP_SITE_CDASH TRUE)
ENDIF()
```

All of the variables set in this file are directly understood by raw `ctest` and will not be explained here further (see documentation for the standard CMake module `CTest`). The usage of the function SET_DEFAULT_AND_FROM_ENV() allows the varaibles to be overridded both as CMake cache variables and in the environment. The latter is needed when running using `ctest` as the driver. Given that all of these variables are nicely namespaced, overriding them on the shell environment is not as dangerous as might otherwise be the case but this is what had to be done to get around limitations for older versions of CMake/CTest.

<projectDir>/Version.cmake: If defined, gives the project's version and determines development/release mode (see Project and Repositiory Versioning and Release Mode). This file is read in (using `INCLUDE()`) in the project's base-level `<projectDir>/CMakeLists.txt` file scope so local variables set in this file are seen by the entire CMake project. For example, TribitsExampleProject/`Version.cmake`, this looks like:

```
SET(${REPOSITORY_NAME}_VERSION 1.1)
SET(${REPOSITORY_NAME}_MAJOR_VERSION 01)
SET(${REPOSITORY_NAME}_MAJOR_MINOR_VERSION 010100)
SET(${REPOSITORY_NAME}_VERSION_STRING "1.1 (Dev)")
SET(${REPOSITORY_NAME}_ENABLE_DEVELOPMENT_MODE_DEFAULT ON) # Change to 'OFF' for a rel
```

Note that the prefix `${REPOSITORY_NAME}_` is used instead of hard-coding the project name. This is so that the same `Version.txt` file can be used as the the <repoDir>/Version.cmake file and have the repository name be flexible. TriBITS sets `REPOSITORY_NAME = ${PROJECT_NAME}` when it reads in this file at the project-level scope.

It is strongly recommended that every TriBITS project contain a `Version.cmake` file, even if a release has never occured. Otherwise, the project needs to define the variable `${PROJECT_NAME}_ENABLE_DEVELOPMENT_MODE_DEFAULT` at the global project scope (perhaps in `<projectDir>/ProjectName.cmake`) to get right development mode of behavior (see ${PROJECT_NAME}_ENABLE_DEVELOPMENT_MODE).

<projectDir>/project-checkin-test-config.py: [Optional] Used to define the `--default-builds` and other project-level configuration options for the project's usage of the checkin-test.py script. Machine or package-specific options should **not** be placed in this file. An example of this file for TribitsExampleProject/`project-checkin-test-config.py` is shown below:

```
#
# Define project-specific options for the checkin-test script for
# TribitsExampleProject.
#

configuration = {

    # Default command line arguments
    'defaults': {
        '--send-email-to-on-push': 'trilinos-checkin-tests@software.sandia.gov',
        },

    # CMake options (-DVAR:TYPE=VAL) cache variables.
    'cmake': {

        # Options that are common to all builds.
        'common': [],

        # Defines --default-builds, in order.
        'default-builds': [
            # Options for the MPI_DEBUG build.
```

```
            ('MPI_DEBUG', [
                '-DTPL_ENABLE_MPI:BOOL=ON',
                '-DCMAKE_BUILD_TYPE:STRING=RELEASE',
                '-DTribitsExProj_ENABLE_DEBUG:BOOL=ON',
                '-DTribitsExProj_ENABLE_CHECKED_STL:BOOL=ON',
                '-DTribitsExProj_ENABLE_DEBUG_SYMBOLS:BOOL=ON',
                ]),
            # Options for the SERIAL_RELEASE build.
            ('SERIAL_RELEASE', [
                '-DTPL_ENABLE_MPI:BOOL=OFF',
                '-DCMAKE_BUILD_TYPE:STRING=RELEASE',
                '-DTribitsExProj_ENABLE_DEBUG:BOOL=OFF',
                '-DTribitsExProj_ENABLE_CHECKED_STL:BOOL=OFF',
                ]),
            ], # default-builds

        }, # cmake

    } # configuration
```

The contents of the file `project-checkin-test-config.py` show above are pretty self explanatory. This file defines a single python dictionary data-structure called `configuration` which gives some default arguments in `defaults`, and then cmake options that define the projects `--default-builds`. For more details, see the section Pre-push Testing using checkin-test.py.

<projectDir>/cmake/NativeRepositoriesList.cmake: [Optional] If present, this file gives the list of native repositories to this TriBITS project. The file must contain a `SET()` statement defining the variable `${PROJECT_NAME}_NATIVE_REPOSITORES` which is just a flat list of repository names that must also be directory names under `<projectDir>/`. For example, if this file contains:

```
SET(${PROJECT_NAME}_NATIVE_REPOSITORES Repo0 Repo1)
```

then the directories `<projectDir>/Repo0` and `<projectDir>/Repo1` must exist and must be valid TriBITS repositories (see *TriBITS Repository*).

There are no examples for the usage of this file in any of the TriBITS examples or test projects. However, support for this file is maintained for backward compatibility since there are some TriBITS projects that use it. It is recommended instead to define multiple repositories using the <projectDir>/cmake/ExtraRepositoriesList.cmake file as it allows for more flexibility in how extra repositories are specified and how they are accessed.

If this file `NativeRepositoriesList.cmake` does not exist, then TriBITS sets `'${PROJECT_NAME}_NATIVE_REPOSITORES` eqaual to ".", or the base project directory (i.e. `<projectDir>/.`). In this case, the file `<projectDir>/PackagesList.cmake` and `<projectDir>/TPLsList.cmake` must exist. However, if the project has no native packages or TPLs, then these files be set up to list no packages or TPLs. This is the case for meta-projects like VERA that have only extra repositories specified in the file <projectDir>/cmake/ExtraRepositoriesList.cmake.

<projectDir>/cmake/ExtraRepositoriesList.cmake: [Optional] If present, this file defines a list of extra repositories that are added on to the project's native repositories. The list of repositories is defined using the macro TRIBITS_PROJECT_DEFINE_EXTRA_REPOSITORIES(). For example, the extra repos file:

```
TRIBITS_PROJECT_DEFINE_EXTRA_REPOSITORIES(
  ExtraRepo1  ""  GIT  someurl.com:/ExtraRepo1  ""  Continuous
  ExtraRepo2  packages/SomePackage/Blah  GIT  someurl2.com:/ExtraRepo2  NOPACKAGES  Ni
  ExtraRepo3  ""  HG  someurl3.com:/ExtraRepo3 ""  Continuous
  ExtraRepo4  ""  SVN  someurl4.com:/ExtraRepo4 ""  Nightly
  )
```

shows the speification of both TriBITS Repositories and non-TriBITS VC Repositories. In the above file, the repositories `ExtraRepo1`, `ExtraRepo3`, and `ExtraRepo4` are VC repositories that are cloned into directories under `<projectDir>` of the same names from the URLs `someurl.com:/ExtraRepo1`,

someurl3.com:/ExtraRepo3, and someurl4.com:/ExtraRepo4, respectively. However, the repository ExtraRepo2 is **not** a TriBITS Repository because it is marked as NOPACKAGES. In this case, it gets cloned as the directory:

```
<projectDir>/packages/SomePackage/Blah
```

However, the code in the tools checkin-test.py and TribitsCTetsDriverCore.cmake will consider this repository and directory and any changes to this repository will be listed as changes to somePackage.

<projectDir>/cmake/ProjectDependenciesSetup.cmake: [Optional] If present, this file is included a single time as part of the generation of the project's dependency data-structure (see Reduced Package Dependency Processing). It gets included at the top project level scope after all of the <repoDir>/cmake/RepositoryDependenciesSetup.cmake files have been included but before all of the package <packageDir>/cmake/Dependencies.cmake files are included. Any local variables set in this file have project-wide scope. The primary purpose for this file is to set variables that will impact the processing of project's package's Dependencies.cmake files.

The typical usage of this file is to set the default CDash email base address that will be the default for all of the defined packages (see CDash regression email addresses). For example, to set the default email address for all of the packages, one would set in this file:

```
SET_DEFAULT(${PROJECT_NAME}_PROJECT_MASTER_EMAIL_ADDRESSS
    projectx-regressions@somemailserver.org)
```

The repository email address varaibles ${REPOSITORY_NAME}_REPOSITORY_EMAIL_URL_ADDRESSS_BASE and ${REPOSITORY_NAME}_REPOSITORY_MASTER_EMAIL_ADDRESSS possibly set in the just processed <repoDir>/cmake/RepositoryDependenciesSetup.cmake files can also be overridded here. The CASL VERA meta-project uses this file to override several of the repository-specific email addresses for its constituent CDash email addresses.

In general, variables that affect how package dependnecies are defined or affect package and TPL enable/disable logic should be defined in this file.

<projectDir>/cmake/CallbackDefineProjectPackaging.cmake: [Optional] If exists, defines the CPack settings for the project (see Offical CPack Documentation and Online CPack Wiki). This file must define a macro called TRIBITS_PROJECT_DEFINE_PACKAGING() which is then invoked by TriBITS. The file:

TribitsExampleProject/cmake/CallbackDefineProjectPackaging.cmake

provides a good example which is:

```
MACRO(TRIBITS_PROJECT_DEFINE_PACKAGING)

    TRIBITS_COPY_INSTALLER_RESOURCE(TribitsExProj_README
      "${TribitsExProj_SOURCE_DIR}/README"
      "${TribitsExProj_BINARY_DIR}/README.txt")
    TRIBITS_COPY_INSTALLER_RESOURCE(TribitsExProj_LICENSE
      "${TribitsExProj_SOURCE_DIR}/LICENSE"
      "${TribitsExProj_BINARY_DIR}/LICENSE.txt")

    SET(CPACK_PACKAGE_DESCRIPTION "TribitsExampleProject just shows you how to use Tri
    SET(CPACK_PACKAGE_FILE_NAME "tribitsexproj-setup-${TribitsExProj_VERSION}")
    SET(CPACK_PACKAGE_INSTALL_DIRECTORY "TribitsExProj ${TribitsExProj_VERSION}")
    SET(CPACK_PACKAGE_REGISTRY_KEY "TribitsExProj ${TribitsExProj_VERSION}")
    SET(CPACK_PACKAGE_NAME "tribitsexproj")
    SET(CPACK_PACKAGE_VENDOR "Sandia National Laboratories")
    SET(CPACK_PACKAGE_VERSION "${TribitsExProj_VERSION}")
    SET(CPACK_RESOURCE_FILE_README "${TribitsExProj_README}")
    SET(CPACK_RESOURCE_FILE_LICENSE "${TribitsExProj_LICENSE}")
    SET(${PROJECT_NAME}_CPACK_SOURCE_GENERATOR_DEFAULT "TGZ;TBZ2")
    SET(CPACK_SOURCE_FILE_NAME "tribitsexproj-source-${TribitsExProj_VERSION}")
    SET(CPACK_COMPONENTS_ALL ${TribitsExProj_PACKAGES} Unspecified)

ENDMACRO()
```

The CPack variables that should be defined at the project-level should be described in the Offical CPack Documentation.

Settings that are general for all distributions (like non-package repository files to exclude from the tarball) should be set at the in the file <repoDir>/cmake/CallbackDefineRepositoryPackaging.cmake.

<projectDir>/cmake/tribits/: [Optional] The typical location of the `tribits` soruce tree for projects that choose to snapshot or checkout TriBITS into their source tree. Trilinos, for example, currently snapshots the TriBITS source tree into this directory.

<projectDir>/cmake/ctest/CTestCustom.cmake.in: [Optional] If this file exists, it is processed using a `CONFIGURE_FILE()` command to write the file `CTestCustom.cmake` in the project base build tree. This file is picked up automatically by `ctest` (see CTest documentation). This file is typically used to change the maximum size of test output. For example, the TribitsExampleProject/cmake/ctest/CTestCustom.cmake.in" looks like:

```
# Allow full output to go to CDash
SET(CTEST_CUSTOM_MAXIMUM_PASSED_TEST_OUTPUT_SIZE 0)
SET(CTEST_CUSTOM_MAXIMUM_FAILED_TEST_OUTPUT_SIZE 0)
# WARNING!  This could be a lot of output and could overwhelm CDash and the
# MySQL DB so this might not be a good idea!
```

which sets the output size for each test submitted to CDash be unlimited (which is not really recommended). These variables used by Trilinos at one time were:

```
SET(CTEST_CUSTOM_MAXIMUM_PASSED_TEST_OUTPUT_SIZE 50000)
SET(CTEST_CUSTOM_MAXIMUM_FAILED_TEST_OUTPUT_SIZE 5000000)
```

which sets the max output for passed and failed tests to 50000k and 5000000k, respectively.

For documentation of the options one can change for CTest, see This Online Wiki Page.

TriBITS Project Core Variables The following local variables are defined in the top-level Project `CMakeLists.txt` file scople and are therefore acceessible by all files prcessed by TriBITS:

`PROJECT_NAME`

> The name of the TriBTS Project. This exists to support, among other things, the ability for subordinate units (Repositories and Packages) to determine the Project in which is participating. This is typically read from a `SET()` statement in the project's <projectDir>/ProjectName.cmake file.

`PROJECT_SOURCE_DIR`

> The absolute path to the base Project source directory. This is set automatically by TriBITS given the directory passed into `cmake` at configure time at the beginning of the TRIBITS_PROJECT() macro.

`PROJECT_BINARY_DIR`

> The absolute path to the base Project binary/build directory. This is set automatically by TriBITS and is the directory where `cmake` is run from and is set at the beginning of the TRIBITS_PROJECT() macro

`${PROJECT_NAME}_SOURCE_DIR`

> Set to `${PROJECT_SOURCE_DIR}` automatically by TriBITS at the beginning of the TRIBITS_PROJECT() macro.

`${PROJECT_NAME}_BINARY_DIR`

> Set to `${PROJECT_BINARY_DIR}` automatically by TriBITS at the beginning of the TRIBITS_PROJECT() macro.

`${PACKAGE_NAME}_ENABLE_TESTS`

> CMake cache varaibles that if set to `ON`, then tests for all explicitly enabled packages will be turned on.

`${PACKAGE_NAME}_ENABLE_EXAMPLES`

CMake cache varaibles that if set to `ON`, then examples for all explicitly enabled packages will be turned on.

The following internal project-scope local CMake variables are defined by TriBITS for the project's repositoreis.:

`${PROJECT_NAME}_NATIVE_REPOSITORIES`

The list of Native Repositories for a given TriBITS project (i.e. Repositories that are always present when configuring the Project and are managed in the same VC repo typically). If the file `${PROJECT_SOURCE_DIR}/NativeRepositoriesList.cmake` exists, then the list of native repositories will be read from that file. If the file `NativeRepositoriesList.cmake` does not exist, then the project is assumed to also be a repository and the list of native repositories is just the local project directory `${PROJECT_SOURCE_DIR}/.`. In this case, the `${PROJECT_SOURCE_DIR}/` must contain at a minumum a `PackagesList.cmake`, and a `TPLsList.cmake` file (see TriBITS Repository).

`${PROJECT_NAME}_EXTRA_REPOSITORIES`

The list of Extra Repositories that the project is being configured with. The packages in these repositories are *not* listed in the main project dependency files but are listed in the dependenicy files in other contexts. This list of repositories either comes from the project's `ExtraRepositoriesList.cmake` file or comes from the CMake variables ${PROJECT_NAME}_EXTRA_REPOSITORIES. See Enabling extra repositories with add-on packages for details.

`${PROJECT_NAME}_ALL_REPOSITORIES`

Concatenation of all the repos listed in `${PROJECT_NAME}_NATIVE_REPOSITORIES` and `${PROJECT_NAME}_EXTRA_REPOSITORIES` in the order that the project is being configured with.

TriBITS Repository

A TriBITS Repository is the basic unit of ready-made composition between different collections of software that use the TriBITS CMake build and system.

In short, a TriBITS Repository:

- Is a named collection of related TriBITS Packages and TPLs (defined in <repoDir>/PackagesList.cmake and <repoDir>/TPLsList.cmake respectively)
- Defines the base source and binary directories for the Repository `${REPOSITORY_NAME}_SORUCE_DIR` and `${REPOSITORY_NAME}_BINARY_DIR`.
- [Optional] Defines a common set of initializations and other hooks for a all the packages in the repository.

For more details on the definition of a TriBITS Repository, see:

- TriBITS Repository Core Files
- TriBITS Repository Core Variables

TriBITS Repository Core Files The core files making up a TriBITS Repository (where `<reposDir> = ${${REPOSITORY_NAME}_SOURCE_DIR}`) are:

```
<repoDir>/
  PackagesList.cmake
  TPLsList.cmake
  Copyright.txt  # [Optional] Only needed if creating version header file
  Version.cmake  # [Optional] Info inserted into ${REPO_NAME}_version.h
```

```
cmake/
   RepositoryDependenciesSetup.cmake # [Optional]
   CallbackSetupExtraOptions.cmake # [Optional] Called after tribits options
   CallbackDefineRepositoryPackaging.cmake # [Optioinal] CPack packaging
```

These TriBITS Repository files are documented in more detail below:

- <repoDir>/PackagesList.cmake
- <repoDir>/TPLsList.cmake
- <repoDir>/Copyright.txt
- <repoDir>/Version.cmake
- <repoDir>/cmake/RepositoryDependenciesSetup.cmake
- <repoDir>/cmake/CallbackSetupExtraOptions.cmake
- <repoDir>/cmake/CallbackDefineRepositoryPackaging.cmake

<repoDir>/PackagesList.cmake: [Required] Provides the list of top-level packages defined by the repository. This file typically just calls the macro TRIBITS_DEFINE_REPOSITORY_PACKAGES() to define the list of packages along with their directories and other properties. For example, the file TribitsExampleProject/`PackagesList.cmake` looks like:

```
TRIBITS_DEFINE_REPOSITORY_PACKAGES(
  SimpleCxx               packages/simple_cxx                 PT
  MixedLanguage           packages/mixed_language             PT
  PackageWithSubpackages  packages/package_with_subpackages   PT
  WrapExternal            packages/wrap_external              PT
  )

TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS(WrapExternal Windows)
```

Other comamnds that are appropriate to use in this file include TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS(). Also, if the binary directory for any package `<packageName>` needs to be changed from the default, then the variable `<packageName>_SPECIFIED_BINARY_DIR` can be set. One can see an example of this in the file `tribits/PackageList.cmake` which shows

```
TRIBITS_DEFINE_REPOSITORY_PACKAGES(
  TriBITS  .    PT
  )

# Must create subdir for binary dir for the TriBITS package
SET(TriBITS_SPECIFIED_BINARY_DIR tribits)
```

(see TriBITS Package, TriBITS Repository, TriBITS Package sharing the same source directory).

It is perfectly legal for a TriBITS repjository to define no packages at all with:

```
TRIBITS_DEFINE_REPOSITORY_PACKAGES()
```

and this would be the case for a TriBITS meta-project that has no native packages, only extra repositories.

<repoDir>/TPLsList.cmake: [Required] Provides the list of TPLs that are listed as TPLs in the repository's SE packages <packageDir>/cmake/Dependencies.cmake files (see *TriBITS TPL*). This file typically just calls the macro TRIBITS_DEFINE_REPOSITORY_TPLS() to define the TPLs along with their find modules and other properties. See an example from TribitsExampleProject/`TPLsList.cmake` which shows:

```
TRIBITS_DEFINE_REPOSITORY_TPLS(
  MPI  "${${PROJECT_NAME}_TRIBITS_DIR}/tpls/FindTPLMPI.cmake"   PT
  )
```

Once processed, each listed TPL `TPL_NAME` is given global non-cache variables for the TPL's find module `${TPL_NAME}_FINDMOD` test group `${TPL_NAME}_TESTGROUP` (see SE Package Test Group).

It is perfectly fine to specify no TPLs for a repository with:

```
TRIBITS_DEFINE_REPOSITORY_TPLS()
```

but the macro `TRIBITS_DEFINE_REPOSITORY_TPLS()` has to be called, even if there are no TPLs. See TRIBITS_DEFINE_REPOSITORY_TPLS() for further details.

<repoDir>/Copyright.txt: [Optional] Gives the default copyright and license declaration for all of the software in the TriBITS repository `<repoDir>`. This file is read into a string and then used to configure the repository's version file (see Project and Repositiory Versioning and Release Mode).

<repoDir>/Version.cmake: Contains version information for the repository (and the project also if this is also the base project). For example, TribitsExampleProject/`Version.cmake`, this looks like:

```
SET(${REPOSITORY_NAME}_VERSION 1.1)
SET(${REPOSITORY_NAME}_MAJOR_VERSION 01)
SET(${REPOSITORY_NAME}_MAJOR_MINOR_VERSION 010100)
SET(${REPOSITORY_NAME}_VERSION_STRING "1.1 (Dev)")
SET(${REPOSITORY_NAME}_ENABLE_DEVELOPMENT_MODE_DEFAULT ON) # Change to 'OFF' for a rel
```

Note that the prefix `${REPOSITORY_NAME}_` is used instead of hard-coding the repository's name to allow flexibility in what a meta-project names a given TriBITS repository.

The local variables in these set statements are processed in the base project directory's local scope and are therefore seen by the entire CMake project. When this file is read in repository mode, the variable `${REPOSITORY_NAME}_ENABLE_DEVELOPMENT_MODE_DEFAULT` is ignored.

<repoDir>/cmake/RepositoryDependenciesSetup.cmake: [Optional] If present, this file is included a single time as part of the generation of the project dependency data-structure (see Reduced Package Dependency Processing). It gets included at the top project level scope in order with the other repositories listed in `${PROJECT_NAME}_ALL_REPOSITORIES`. Any local variables set in this file have project-wide scope. The primary purpose for this file is to set variables that will impact the processing of project's package's `Dependencies.cmake` files and take care of other enable/disable issues that are not clearly handled by the TriBITS system automatically.

The typical usage of this file is to set the default CDash email base address that will be the default for all of the defined packages (see CDash regression email addresses). For example, to set the default email address for all of the packages in this repository, one would set in this file:

```
SET_DEFAULT(${REPOSITORY_NAME}_REPOSITORY_MASTER_EMAIL_ADDRESSS
    repox-regressions@somemailserver.org)
```

<repoDir>/cmake/CallbackSetupExtraOptions.cmake: [Optional] If defined, this file is processed (included) for each repo in order right after the basic TriBITS options are defined in the macro `TRIBITS_DEFINE_GLOBAL_OPTIONS_AND_DEFINE_EXTRA_REPOS()`. This file must define the macro `TRIBITS_REPOSITORY_SETUP_EXTRA_OPTIONS()` which is then called by the TriBITS system. This file is only processed when doing a basic configuration of the project and **not** when it is just building up the dependency data-structures as part of other tools (like `checkin-test.py`, TRIBITS_CTEST_DRIVER(), etc.). Any local variables set in this file or macro have project-wide scope.

A few additional variables are defined by the time this file is procesed and can be ued in the logic in these files. The variables that should already be defined, in addition to all of the basic user TriBITS cache variables, include `CMAKE_HOST_SYSTEM_NAME`, `${PROJECT_NAME}_HOSTNAME`, and `PYTHON_EXECUTABLE`. The types of logic to put in this file includes:

- Setting additional user cache variable options that are used by multiple packages into the TriBITS Repository. For example, Trilinos defines a `Trilinos_DATA_DIR` user cache varaible that several Trilinos packages use.

- Disabling packages in the TriBITS Repository when conditions will not allow them to be enabled. For example, Trilinos disables the package ForTrilinos when Fortran is disabled and disables PyTrilinos when Python support is disabled.

An example of this file is:

TribitsExampleProject//cmake/CallbackSetupExtraOptions.cmake

which currently looks like:

```
MACRO(TRIBITS_REPOSITORY_SETUP_EXTRA_OPTIONS)

  ASSERT_DEFINED(${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES)
  ASSERT_DEFINED(${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES)

  IF (${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES OR
    ${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES
    )
    MESSAGE(
      "\n***"
      "\n*** Warning: Setting ${PROJECT_NAME}_ENABLE_WrapExternal=OFF"
      " because ${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES or"
      " ${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES is on!"
      "\n***\n"
      )
    SET(${PROJECT_NAME}_ENABLE_WrapExternal OFF)
  ENDIF()

ENDMACRO()
```

<repoDir>/cmake/CallbackDefineRepositoryPackaging.cmake: [Optional] If this file exists, then this file defines extra CPack-related options that are specific to the TriBITS Repository. This file must define the macro `TRIBITS_REPOSITORY_DEFINE_PACKAGING()` which is called by TriBITS. This file is procdessed as the top project-level scope so any local variables set have project-wide effect. This file is processed before the project's <projectDir>/cmake/CallbackDefineProjectPackaging.cmake so any options defined in the repositories file are overridden by the project. This file typically this just involves setting extra excludes to remove files from the tarball. The file:

TribitsExampleProject/cmake/CallbackDefineRepositoryPackaging.cmake

provides a good example which is:

```
MACRO(TRIBITS_REPOSITORY_DEFINE_PACKAGING)

  ASSERT_DEFINED(${REPOSITORY_NAME}_SOURCE_DIR)
  APPEND_SET(CPACK_SOURCE_IGNORE_FILES
    "${${REPOSITORY_NAME}_SOURCE_DIR}/cmake/ctest/"
    )

ENDMACRO()
```

TriBITS Repository Core Variables The following local variables are defined automatically by TriBITS before processing a given TriBITS repositories files (e.g. `PackagesList.cmake`, `TPLsList.cmake`, etc.):

`REPOSITORY_NAME`

The name of the current TriBITS repository.

`REPOSITORY_DIR`

Path of the current Repository relative to the Project directory. This is typically just the repository name but can be an arbitrary directory if specified through a `ExtraRepositoriesList.cmake` file.

The following base project-scope local variables are available once the list of TriBITS repositories are defined:

`${REPOSITORY_NAME}_SOURCE_DIR`

The absolute path to the base of a given Repository source directory. CMake code, for example in a packages's `CMakeLists.txt` file, typically refers to this by the raw name like `RepoX_SOURCE_DIR`. This makes such CMake code independent of where the various TriBITS repos are in relation to each other or the Project.

`${REPOSITORY_NAME}_BINARY_DIR`

The absolute path to the base of a given Repository binary directory. CMake code, for example in packages, refer to this by the raw name like `RepoX_SOURCE_DIR`. This makes such CMake code independent of where the various TriBITS repos are in relation to each other or the Project.

TriBITS Package

A TriBITS Package:

- Typically defines a set of libraries and/or header files and/or executables and/or tests with CMake build targets for building these and exports the list of include directories, libraries, and targets that are created (along with CMake dependencies).
- Is declared in its parent repository's <repoDir>/PackagesList.cmake file.
- Defines dependencies on upstream TPLs and/or other SE packages by just naming the dependencies in the file <packageDir>/cmake/Dependencies.cmake..
- Can optionally have subpackages listed in the argument `SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS` to TRIBITS_DEFINE_PACKAGE_DEPENDENCIES().
- Is the fundamental unit of software partitioning and aggregation and must have a unique packae name that is globally unique (see Globally unique TriBITS package names).
- Is the unit of testing as drived by TRIBITS_CTEST_DRIVER() and displayed on CDash.

WARNING: As noted above, one must be very careful to pick package names that will be globally unique not only within its defined repository but also across all SE packages in all TriBITS repositories that ever might be cobbled together into a single Tribits (meta) project! Choosing the package name is the single most important decision when it comes to defining TriBITS packages. Do not pick names like "Debug" or "StandardUtils" that have a high chance of clashing with other sloppy project development teams.

For more details on the definition of a TriBITS Package (or subpackage), see:

- TriBITS Package Core Files
- TriBITS Package Core Variables

TriBITS Package Core Files The core files that make up TriBITS Package (where `<packageDir> = ${${PACKAGE_NAME}_SOURCE_DIR}`) are:

```
<packageDir>/
  CMakeLists.txt  # Only processed if the package is enabled
  cmake/
    Dependencies.cmake  # Always processed if the package is listed in the
                        # enclosing Repository
```

NOTE: Before a TriBITS Package's files are described in more detail, it is important to note that all of the package's files that define its behavior and tests should strictly be contained under the package's base source directory `<packageDir>/` if at all possible. While this is not a requirement for the basic TriBITS build system, the approach for automatically detecting when a package has changed by looking at what files have changed (which is used in `checkin-test.py` and TRIBITS_CTEST_DRIVER()) requires that the package's files be listed under `<packageDir>/` (see Pre-push Testing using checkin-test.py). Without this, these development tetsing tools will not be able to effectively determine what needs to be rebuilt and retested which can clean to pushing broken software.

These TriBITS Package files are documented in more detail below:

- <packageDir>/cmake/Dependencies.cmake
- <packageDir>/CMakeLists.txt

<packageDir>/cmake/Dependencies.cmake: [Required] Defines the dependencies of a given SE package using the macro TRIBITS_DEFINE_PACKAGE_DEPENDENCIES(). This file is processed at the top-level project scope (using an `INCLUDE()`) so any local variables set will be seen by the entire project. This file is always processed, including when just building the project's <Project>PackageDependencies.xml file.

An example of a `Dependencies.cmake` file for a package with optional and required dependencies is for the mock `Panzer` package in MockTrilinos:

```
TRIBITS_DEFINE_PACKAGE_DEPENDENCIES(
  LIB_REQUIRED_PACKAGES Teuchos Sacado Phalanx Intrepid Thyra
    Tpetra Epetra EpetraExt
  LIB_OPTIONAL_PACKAGES Stokhos
  TEST_OPTIONAL_PACKAGES Stratimikos
  LIB_REQUIRED_TPLS MPI Boost
  )
```

An example of a package with subpackages is `PackageWithSubpackages` in TribitsExampleProject with the `Dependencies.cmake` file:

```
TRIBITS_DEFINE_PACKAGE_DEPENDENCIES(
  SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS
    SubpackageA   A   PT  REQUIRED
    SubpackageB   B   PT  REQUIRED
    SubpackageC   C   PT  REQUIRED
  )
```

The last case defines three subpackages which creates three new SE packages with names `PackageWithSubpackagesSubpackageA`, `PackageWithSubpackagesSubpackageB`, and `PackageWithSubpackagesSubpackageC`.

<packageDir>/CMakeLists.txt: [Required] The package's top-level `CMakeLists.txt` file that defines the libraries, include directories, and contains the tests for the package.

The basic structure of this file for a **package without subpackages** is shown in:

TribitsExampleProject/`packages/simple_cxx/CMakeLists.txt`

which is:

```
#
# A) Define the package
#
TRIBITS_PACKAGE( SimpleCxx  ENABLE_SHADOWING_WARNINGS  CLEANED )

#
# B) Platform-specific checks
#
```

```
INCLUDE(CheckFor__int64)
CHECK_FOR___INT64(HAVE_SIMPLECXX___INT64)

#
# C) Set up package-specific options
#
TRIBITS_ADD_DEBUG_OPTION()
TRIBITS_ADD_SHOW_DEPRECATED_WARNINGS_OPTION()

#
# D) Add the libraries, tests, and examples
#
ADD_SUBDIRECTORY(src)
TRIBITS_ADD_TEST_DIRECTORIES(test)
#TRIBITS_ADD_EXAMPLE_DIRECTORIES(example)

#
# E) Do standard postprocessing
#
TRIBITS_PACKAGE_POSTPROCESS()
```

The first command at the top of the file in TRIBITS_PACKAGE() where the package name is passed in in addition to a few other options. TriBITS obviously already knows the package name. The purpose for repeating it is as documentation for the developer's sake. Then a set of platform confgiure-time tests is typically performed (if there are any). In this example, the existance of the C++ `__int64` data-type is checked using the module *CheckFor__int64.cmake*' (which is in the `cmake/` directory of this package. (CMake has great support for configure-time tests, see Configure-time System Tests.) This is followed by package-specific options. In this case, the standard TriBITS options for debug checking and deprecated warnings are added using the standard macros TRIBITS_ADD_DEBUG_OPTION() and TRIBITS_ADD_SHOW_DEPRECATED_WARNINGS_OPTION(). After all of this up front stuff (which will be present in any moderately complex CMake-configured project) the source and the test subdirectories are added that actually define the library and the tests. In this case, the standard TRIBITS_ADD_TEST_DIRECTORIES() macro is used which only conditionally adds the tests for the package.

The final command in the package's `CMakeLists.txt` file must always be TRIBITS_PACKAGE_POSTPROCESS(). This is needed in order to perform some necessary post-processing by TriBITS.

It is also possible to for the package's top-level `CMakeLists.txt` to be the only such file in a package. Such an example can be see in the example project TribitsHelloWorld.

When a TriBITS package is broken up into subpackages (see TriBITS Subpackage), its `CMakeLists.txt` file looks a little different. The basic structure of this file for a **package with subpackages** is shown in:

TribitsExampleProject/`packages/package_with_subpackages/CMakeLists.txt`

which contains:

```
#
# A) Forward declare the package so that certain options are also defined for
# subpackages
#
TRIBITS_PACKAGE_DECL(PackageWithSubpackages)

#
# B) Define the common options for the package first so they can be used by
# subpackages as well.
#
TRIBITS_ADD_DEBUG_OPTION()

#
# C) Process the subpackages
#
```

```
TRIBITS_PROCESS_SUBPACKAGES()

#
# D) Define the package now and perform standard postprocessing
#
TRIBITS_PACKAGE_DEF()
TRIBITS_PACKAGE_POSTPROCESS()
```

What is different about `CMakeLists.txt` files for packages without subpackages is that the TRIBITS_PACKAGE() command is broken up into two parts TRIBITS_PACKAGE_DECL() and TRIBITS_PACKAGE_DEF(). In between these two comamnds, the parent package can define the common package options and then calls the commnand TRIBITS_PROCESS_SUBPACKAGES() which fully processes the packages. If the parent package has libraries and/or tests/example of its own, it can define those after calling TRIBITS_PACKAGE_DEF(), just like with a regular package. However, it is rare for a package broken up into subpackages to have its own libraries and/or tests and examples. As always, the final command called inside of the `CMakeLists.txt` is TRIBITS_PACKAGE_POSTPROCESS().

NOTE: The package's `CMakeLists.txt` file only gets processed if the package is actually enabled with `${PROJECT_NAME}_ENABLE_${PACKAGE_NAME}=ON`. This is an important design feature of TriBITS is that the contents of non-enabled package's can't damnage the configure, build, and test of the enabled packages based on errors in non-enabled packages. This is critical to allow experimental `EX` test packages and lower-maturity packages to exist in the same soruce repositories safely.

TriBITS Package Core Variables The following locally scoped **TriBITS Package Local Varaibles** are defined when the files for a given TriBITS Package (or any SE packaeg for that matter) are being processed:

`PACKAGE_NAME`

> The name of the current TriBITS SE package. This is set automatically by TriBITS before the packages's `CMakeLists.txt` file is processed. **WARNING:** This name must be globally unique across the entire project (see Globally unique TriBITS package names).

`PACKAGE_SOURCE_DIR`

> The absolute path to the base package's base source directory. This is set automatically by TriBITS in the macro TRIBITS_PACKAGE().

`PACKAGE_BINARY_DIR`

> The absolute path to the base package's base binary/build directory. This is set automatically by TriBITS in the macro TRIBITS_PACKAGE().

`PACKAGE_NAME_UC`

> This is set to the upper-case versionof `${PACKAGE_NAME}`. This is set automatically by TriBITS in the macro TRIBITS_PACKAGE().

Once all of the TriBITS SE package's `Dependencies.cmake` files have been processed, the following **TriBITS Package Top-Level Local Variables** are defined:

`${PACKAGE_NAME}_SOURCE_DIR`

> The absolute path to the base of a given package's source directory. CMake code, for example in other packages, refer to this by the raw name like `PackageX_SOURCE_DIR`. This makes such CMake code independent of where the package is in relation to other packages. This variable is defined for all processed packages, independent of whether they are enabled.

`${PACKAGE_NAME}_BINARY_DIR`

> The absolute path to the base of a given package's binary directory. CMake code, for example in other packages, refer to this by the raw name like `PackageX_BINARY_DIR`. This makes such CMake code independent of where the package is in relation to other packages. This variable is only defined if the package is enabled.

`${PACKAGE_NAME}_PARENT_REPOSITORY`

The name of the package's parent repository. This can be used by a package to access information about its parent repository. For example, the variable `${${PACKAGE_NAME}_PARENT_REPOSITORY}_SOURCE_DIR` can be dereferenced.

`${PACKAGE_NAME}_TESTGROUP`

Defines the SE Package Test Group for the package. This determines in what contexts the package is enbled or not for testing-related purposes (see Nested Layers of TriBITS Project Testing)

In addition, the following user-settable **TriBITS Package Cache Variables** are defined before an SE Package's `CMakeLists.txt` file is processed:

`${PROJECT_NAME}_ENABLE_${PACKAGE_NAME}`

Set to `ON` if the package is to be enabled.

`${PACKAGE_NAME}_ENABLE_${OPTIONAL_DEP_PACKAGE_NAME}`

Set to `ON` if support for the optional upstream dependent package `${OPTIONAL_DEP_PACKAGE_NAME}` is enabled in package `${PACKAGE_NAME}`. Here `${OPTIONAL_DEP_PACKAGE_NAME}` corresponds to each optional upstream SE package listed in the `LIB_OPTIONAL_PACKAGES` and `TEST_OPTIONAL_PACKAGES` arguments to the TRIBITS_DEFINE_PACKAGE_DEPENDENCIES() macro. **NOTE:** It is important that the CMake code in the package `${PACAKGE_NAME}` key off of this variable and **not** the global `${PROJECT_NAME}_ENABLE_${OPTIONAL_DEP_PACKAGE_NAME}` variable because `${PACKAGE_NAME}_ENABLE_${OPTIONAL_DEP_PACKAGE_NAME}` can be explicitly turned off by the user even through the packages `${PACAKGE_NAME}` and `${OPTIONAL_DEP_PACKAGE_NAME}` are both enabled (see Support for optional SE package/TPL can be explicitly disabled).

`${PACKAGE_NAME}_ENABLE_${OPTIONAL_DEPENDENT_TPL_NAME}`

Set to `ON` if support for the optional upstream dependent TPL `${OPTIONAL_DEPENDENT_TPL_NAME}` is enabled in package `${PACKAGE_NAME}`. Here `${OPTIONAL_DEPENDENT_TPL_NAME}` corresponds each to the optional upstream TPL listed in the `LIB_OPTIONAL_TPLS` and `TEST_OPTIONAL_TPLS` arguments to the TRIBITS_DEFINE_PACKAGE_DEPENDENCIES() macro.

`${PACKAGE_NAME}_ENABLE_TESTS`

Set to `ON` if the package's tests are to be enabled. This will enable a package's tests and all of its subpackage's tests.

`${PACKAGE_NAME}_ENABLE_EXAMPLES`

Set to `ON` if the package's examples are to be enabled. This will enable a package's examples and all of its subpackage's examples.

The above varaibles can be set by the user or may be set automatically as part of the Package Dependencies and Enable/Disable Logic.

Currently, a Package can refer to its containing Repository and refer to its source and binary directories. This is so that it can refer to repository-level resources (e.g. like the `Trilinos_version.h` file for Trilinos packages). This may be undesirable because it will make it very hard to pull a package out of one Repository and place it in another repository for a different use. However, a package can indirectly refer to its own repository without concern for what it is call by reading the variable `${PACKAGE_NAME}_PARENT_REPOSITORY`.

TriBITS Subpackage

A TriBITS Subpackage:

- Typically defines a set of libraries and/or header files and/or executables and/or tests with CMake build targets for building these and exports the list of include directories, libraries, and targets that are created (along with CMake dependencies).
- Is declared in its parent packages's <packageDir>/cmake/Dependencies.cmake file in a call to TRIBITS_DEFINE_PACKAGE_DEPENDENCIES() using the argument `SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS`.
- Defines dependencies on upstream TPLs and/or other SE packages by just naming the dependencies in the file `cmake/Dependencies.cmake` using the macro TRIBITS_DEFINE_PACKAGE_DEPENDENCIES().
- Can **NOT** have its own subpackages defined (only top-level packages can have subpackages).
- Is enabled or disabled along with all other subpackages in the parent package automatically if it's parent package is enabled or disabled with `${PROJECT_NAME}_ENABLE_${PARENT_PACKAGE_NAME}` set to `ON` or `OFF` respectively.
- Has tests turned on automatically if `${PARENT_PACKAGE_NAME}_ENABLE_TESTS==ON`.

The contents of a TriBITS Subpackage are almost idential to those of a TriBITS Package. The differences are described below.

For more details on the definition of a TriBITS Package (or subpackage), see:

- TriBITS Subpackage Core Files
- TriBITS Subpackage Core Variables

TriBITS Subpackage Core Files The set of core files for a subpackage are identical to the TriBITS Package Core Files. The core files that make up a TriBITS Subpackage (where `<packageDir> = ${${PARENT_PACKAGE_NAME}_SOURCE_DIR}` and `<spkgDir>` is the subpackage directory listed in the `SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS` to TRIBITS_DEFINE_PACKAGE_DEPENDENCIES()) are:

```
<packageDir>/<spkgDir>/
  CMakeLists.txt  # Only processed if this subpackage is enabled
  cmake/
    Dependencies.cmake  # Always processed if the parent package
                        # is listed in the enclosing Repository
```

These TriBITS Subpackage files are documented in more detail below:

- <packageDir>/<spkgDir>/cmake/Dependencies.cmake
- <packageDir>/<spkgDir>/CMakeLists.txt
- How is TriBITS Subpackage is different from a TriBITS Package?

<packageDir>/<spkgDir>/cmake/Dependencies.cmake: The contents of this file for subpackages is idential as for top-level packages. It just contains a call to the macro TRIBITS_DEFINE_PACKAGE_DEPENDENCIES() to define this SE package's upstream TPL and SE package dependencies. A simple example is for `SubpackageB` declared in package_with_subpackages/cmake/Dependencies.cmake shown shown in:

TribitsExampleProject/`packages/package_with_subpackages/B/cmake/Dependneices.cmake`

which is:

```
TRIBITS_DEFINE_PACKAGE_DEPENDENCIES(
  LIB_REQUIRED_PACKAGES PackageWithSubpackagesSubpackageA
  )
```

What this shows is that subpackages must list their dependenices on each other (if such dependencies exist) using the full SE package name `${PARENT_PACKAGE_NAME}${SUBPACKAGE_NAME}` or in this case `PackageWithSubpackages` + `SubpackageA` = `PackageWithSubpackagesSubpackageA`.

Note that the parent SE package depends on its subpackages, not the other way around. For example, the `PackageWithSubpackages` parent SE package depends its SE subpackages `PackageWithSubpackagesSubpackageA`, `PackageWithSubpackagesSubpackageC`, and `PackageWithSubpackagesSubpackageC`. As such all (direct) dependencies for a subpackage must be listed in its own `Dependencies.cmake` file. For example, the `PackageWithSubpackages` subpackage `SubpackageA` depends on the `SimpleCxx` package and is declared as such as shown in:

TribitsExampleProject/`packages/package_with_subpackages/A/cmake/Dependneices.cmake`

which is:

```
TRIBITS_DEFINE_PACKAGE_DEPENDENCIES(
  LIB_REQUIRED_PACKAGES SimpleCxx
  )
```

What this means is that any TPL or library depencenices listed in the parent package's <packageDir>/cmake/Dependencies.cmake file are **NOT** dependencies of its subpackages. For example, if package_with_subpackages/cmake/Dependencies.cmake where changed to be:

```
TRIBITS_DEFINE_PACKAGE_DEPENDENCIES(
  LIB_REQUIRED_TPLS Boost
  SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS
    SubpackageA   A   PT  REQUIRED
    ...
  )
```

then the `Boost` TPL would **NOT** be a dependency of the SE package `PackageWithSubpackagesSubpackageA` but instead would be listed as a dependency of the parent SE package `PackageWithSubpackages`. (And in this case, this TPL dependency is pretty worthless since the SE package `PackageWithSubpackages` does not even define any libraries or tests of its own.)

<packageDir>/<spkgDir>/CMakeLists.txt: [Required] The subpackage's top-level `CMakeLists.txt` file that defines the libraries, include directoies, and contains the tests for the subpackage. The contents of a subpackage's top-level `CMakeLists.txt` is almost indentical to a top-level package's <packageDir>/CMakeLists.txt file. The primary difference is that the commmands TRIBITS_PACKAGE() and TRIBITS_PACKAGE_POSTPROCESS() and replaced with TRIBITS_SUBPACKAGE() and TRIBITS_SUBPACKAGE_POSTPROCESS() as shown in the file:

TribitsExampleProject/`packages/package_with_subpackages/A/CMakeLists.txt`

which contains:

```
#
# A) Define the subpackage
#
TRIBITS_SUBPACKAGE(SubpackageA)

#
# B) Set up subpackage-specific options
#
# Typically there are none or are few as must are picked up from the parent
# package's CMakeLists.txt file!
```

```
#
# C) Add the libraries, tests, and examples
#

INCLUDE_DIRECTORIES(${CMAKE_CURRENT_SOURCE_DIR})
TRIBITS_ADD_LIBRARY(pws_a
  SOURCES A.cpp
  HEADERS A.hpp
  NOINSTALLHEADERS
)

TRIBITS_ADD_TEST_DIRECTORIES(tests)

#
# D) Do standard postprocessing
#
TRIBITS_SUBPACKAGE_POSTPROCESS()
```

Unlike TRIBITS_PACKAGE(), TRIBITS_SUBPACKAGE() does not take any extra arguments. Those extra settings are assumed to be defined by the top-level parent package. Like top-level packages, subpackages are free to define user-setable options and configure-time tests but typically don't. The idea is that subpackages should be lighter weight than top-level packages. OTher than using TRIBITS_SUBPACKAGE() and TRIBITS_SUBPACKAGE_POSTPROCESS(), a subpackage can be layed out just like any other package and can call on any other commands to add libraries, add executables, add test, etc.

TriBITS Subpackage Core Variables The core varaibles assoicated with a subpackage are identicial to the TriBITS Package Core Variables. The only difference is that a subpackage may need to refer to its parent package where a top-level package does not have a parent package. The extra varaibles that are defined when processing a subpackages files are:

`PARENT_PACKAGE_NAME`

The name of the parent package.

`PARENT_PACKAGE_SOURCE_DIR`

The absolute path to the parent package's source directory. This this only defined for a subpackage.

`PARENT_PACKAGE_BINARY_DIR`

The absolute path to the parent package's binary directory. This this only defined for a subpackage.

How is TriBITS Subpackage is different from a TriBITS Package? A common question this is natural to ask is how a TriBITS Subpackage is different from a TriBITS Package? They contain the same basic files (i.e. a `cmake/Dependencies.cmake`, a top-level `CMakeList.txt` file, source files, test files, etc.). They both are included in the list of TriBITS SE Packages and therefore can both be enabled/disabled by the user or in automatica dependnecy logic. The primary difference is that a subpackage is meant to involve less overhead in defining and is to be used to partition the parent package's software into chunks according to software engineering packaging principles. Also, the dependency logic treats a parent package's subpackages as part of itself so when the parent package is explicitly enabled or disabled, it is identical to explicitly enabling or disabling all of its subpackages. Other differences and issues between packages as subpackages are discussed throughout this guide.

TriBITS TPL

A TriBITS TPL:

- Defines a set of pre-built libraries and/or header files and/or executables and/or some other resoruces used by one or more TriBITS Packages and publishes the list of include directories and/or libraries and/or executables provided by the TPL to the TriBITS project.
- Is given a globally unique name (see Globally unique TriBITS TPL names) and is declared in a <repoDir>/TPLsList.cmake file.
- Is listed as an explicit optional or required dependency in one or more TriBITS SE package's <packageDir>/cmake/Dependencies.cmake files.

WARNING: One must be very careful to pick TPL names that will be globally unique across all packages in all TriBITS repositories that ever might be cobbled together into a single TriBITS (meta) project! However, choose TPL names is usually much easier and less risky than choosing Globally unique TriBITS package names because the native TPLs themselves tend to be uniquely named.

Using a TriBITS TPL is to be preferred over using a raw CMake `FIND_PACKAGE(<someCMakePackage>)` because the TriBITS system guarantees that only a single unique version of TPL will be used by multiple packages and by declaring a TPL using TriBITS, automatical enable/disable logic will be applied as described in Package Dependencies and Enable/Disable Logic.

For each TPL referenced in a `TPLsList.cmake` file using the macro TRIBITS_DEFINE_REPOSITORY_TPLS(), there should exist a file, typically called `FindTPL${TPL_NAME}.cmake`, that once processed, produces the variables `${TPL_NAME}_LIBRARIES` and `${TPL_NAME}_INCLUDE_DIRS`. Most `FindTPL${TPL_NAME}.cmake` files just use the the function TRIBITS_TPL_DECLARE_LIBRARIES() the define the TriBITS TPL. A simple example of such a file is the standard `FindTPLPETSC.cmake` module which is:

```
INCLUDE(TribitsTplDeclareLibraries)

TRIBITS_TPL_DECLARE_LIBRARIES( PETSC
  REQUIRED_HEADERS petsc.h
  REQUIRED_LIBS_NAMES petsc
  )
```

Some concrete `FindTPL${TPL_NAME}.cmake` files actually do use `FIND_PACKAGE()` and a standard CMake package find modulue to fill in the guts of finding at TPL.

Note that the TriBITS system does not require the usage of of the function TRIBITS_TPL_DECLARE_LIBRARIES() and does not even care about the TPL module name `FindTPL${TPL_NAME}.cmake`. All that is required is that some CMake file fragment exist that once included, will define the varaibles `${TPL_NAME}_LIBRARIES` and `${TPL_NAME}_INCLUDE_DIRS`. However, to be user friendly, such a CMake file should respond to the same variables as accepted by the standard TRIBITS_TPL_DECLARE_LIBRARIES() funtion.

The core variables related to an enabled TPL are `${TPL_NAME}_LIBRARIES`, `${TPL_NAME}_INCLUDE_DIRS`, and `${TPL_NAME}_TESTGROUP` as defined in TRIBITS_TPL_DECLARE_LIBRARIES() need to be defined. For more details, see TRIBITS_DEFINE_REPOSITORY_TPLS().

3.2 Processing of TriBITS Files: Ordering and Details

One of the most important things to know about TriBITS is what files it processes, in what order, and in what context. This is critical to being able to understand what impact (if any) setting a variable or otherwise changing the CMake runtime state will have on configuring a CMake project which uses TriBITS. While the different files that make up a TriBITS Project, TriBITS Repository, TriBITS Package, and TriBITS TPL were defined in the section TriBITS Project Structure, that material did not fully describe the context and in what order these files are processed by the TriBITS framework.

The TriBITS system processes the project's files in one of two use cases. The first use case is in the basic configuration of the project with a standard `cmake` command invocation in order to set up the build files in the binary directory (see Full TriBITS Project Configuration). The second use case is in reading the project's dependency-related files in order to build a package dependency datastructure (e.g. the <Project>PackageDependencies.xml file, see Reduced Package Dependency Processing)). The second use case of reading the project's dependency files is largely a subset of the first.

Another factor that is important to understand is the scoping in which the various files are processed (with `INCLUDE()` or `ADD_SUBDIRECTORY()`). This scoping has a large impact on the configuration of the project and what effect the

processing of files and setting varaibles have on the project as a whole. Some of the strange scoping rules for CMake are discussed in CMake Language Overview and Gotchas and should be understood before trying to debug issues with processesing. Many of the basic files are processed (included) in the base project <projectDir>/CMakeLists.txt scope and therefore any local variables set in these files are accessible to the entire CMake project (after the file is processed, of course). Other files get processed inside of functions which have their own local scope and therefore only impact the rest of the project in more purposeful ways.

Full TriBITS Project Configuration

The first use case to describe is the full processing of all of the TriBITS project's files starting with the base <projectDir>/CMakeLists.txt file. This begins with the invocation of the command:

```
$ cmake [options] <projectDir>
```

Below, is a short pseudo-code algorithm for the TriBITS framework processing and callbacks that begin in the <projectDir>/CMakeLists.txt and proceed through the call to TRIBITS_PROJECT().

Full Processing of TriBITS Project Files:

1. Read <projectDir>/ProjectName.cmake (sets `PROJECT_NAME`)
2. Call `PROJECT(${PROJECT_NAME} NONE)` (sets `${PROJECT_NAME}_SOURCE_DIR` and `${PROJECT_NAME}_BINARY_DIR`)
3. Execute TRIBITS_PROJECT():
    1) Set `PROJECT_SOURCE_DIR` and `PROJECT_BINARY_DIR`
    2) For each `<optFilei>` in `${${PROJECT_NAME}_CONFIGURE_OPTIONS_FILE}`:
        * `INCLUDE(<optFilei>)`
    3) Set variables `CMAKE_HOST_SYSTEM_NAME` and `${PROJECT_NAME}_HOSTNAME`
        (both of these can be overridden in the cache by the user)
    4) Find Python (sets `PYTHON_EXECUTABLE`)
    5) `INCLUDE(` <projectDir>/Version.cmake `)`
    6) Define primary TriBITS options and read in the list of extra repositories
        (calls `TRIBITS_DEFINE_GLOBAL_OPTIONS_AND_DEFINE_EXTRA_REPOS()`)
        * `INCLUDE(` <projectDir>/cmake/ExtraRepositoriesList.cmake `)`
    7) For each `<repoDir>` in all defined TriBITS repositories:
        * `INCLUDE(` <repoDir>/cmake/CallbackSetupExtraOptions.cmake `)`
        * Call `TRIBITS_REPOSITORY_SETUP_EXTRA_OPTIONS()`
    9) Call `TRIBITS_READ_PACKAGES_PROCESS_DEPENDENCIES_WRITE_XML()`:
        a) For each `<repoDir>` in all defined TriBITS repositories:
            * `INCLUDE(` <repoDir>/PackagesList.cmake `)`
            * `INCLUDE(` <repoDir>/TPLsList.cmake `)`
        b) For each `<repoDir>` in all defined TriBITS repositories:
            * `INCLUDE(` <repoDir>/cmake/RepositoryDependenciesSetup.cmake `)`
        c) `INCLUDE(` <projectDir>/cmake/ProjectDependenciesSetup.cmake `)`
        d) For each `<packageDir>` in all defined top-level packages:
            * `INCLUDE(` <packageDir>/cmake/Dependencies.cmake `)`
                - Sets all package-specific options (see TriBITS Package Cache Variables)
            * For each `<spkgDir>` in all subpackages for this package:
                * `INCLUDE(` <packageDir>/<spkgDir>/cmake/Dependencies.cmake `)`
                    - Sets all subpackage-specific options
    10) Adjust SE packae and TPLs enable/disable
        (see Package Dependencies and Enable/Disable Logic)
    11) Probe and set up the environment (finds MPI, compilers, etc.)
        (see TriBITS Environment Probing and Setup)
    12) For each enabled TPL, `INCLUDE(FindTPL<tplName>.cmake` (see TriBITS TPL)
    13) For each `<repoDir>` in all defined TriBITS repositories:

    * Read <repoDir>/Copyright.txt
    * INCLUDE(<repoDir>/Version.cmake)
    (see Project and Repositiory Versioning and Release Mode)

14) For each <packageDir> in all enabled top-level packages
    * ADD_SUBDIRECTORY(<packageDir>/CMakeLists.txt)
    * For each <spkgDir> in all enabled subpackages for this package:
        * ADD_SUBDIRECTORY(<packageDir>/<spkgDir>/CMakeLists.txt)

15) For each <repoDir> in all defined TriBITS repositories:
    * INCLUDE(<repoDir>/cmake/CallbackDefineRepositoryPackaging.cmake)
    * Call TRIBITS_REPOSITORY_DEFINE_PACKAGING()

17) INCLUDE(<projectDir>/cmake/CallbackDefineProjectPackaging.cmake)
    * Call TRIBITS_PROJECT_DEFINE_PACKAGING()

The TriBITS Framework obviously does a lot more that what is described above but the basic trace of major operations and ordering and the processing of project, repository, package, and subpackage files should be clear. All of this information should also be clear when enabling File Processing Tracing.

Reduced Package Dependency Processing

In addition to the full processing that occurs as part of the Full TriBITS Project Configuration, there are also TriBITS tools that only process as subset of project's file. This reduced processing is performed in order to build up the project's package dependenices data-structure (see TriBITS Environment Probing and Setup) and to write the file <Project>PackageDependencies.xml. For example, the tool checkn-test.py and the script TRIBITS_CTEST_DRIVER() both drive this type of processing. In particular, the CMake -P script TribitsDumpDepsXmlScript.cmake reads all of the project's dependency-related files and dumps out the <Project>PackageDependencies.xml file a defined set of native and extra repositories defined for the project. This reduced processing is given below.

Reduced Dependency Processing of TriBITS Project:

1. Read <projectDir>/ProjectName.cmake (sets PROJECT_NAME)
2. INCLUDE(<projectDir>/cmake/ExtraRepositoriesList.cmake)
3. Call TRIBITS_READ_PACKAGES_PROCESS_DEPENDENCIES_WRITE_XML():
    a) For each <repoDir> in all defined TriBITS repositories:
        * INCLUDE(<repoDir>/PackagesList.cmake)
        * INCLUDE(<repoDir>/TPLsList.cmake)
    b) For each <repoDir> in all defined TriBITS repositories:
        * INCLUDE(<repoDir>/cmake/RepositoryDependenciesSetup.cmake)
    c) INCLUDE(<projectDir>/cmake/ProjectDependenciesSetup.cmake)
    d) For each <packageDir> in all defined top-level packages:
        * INCLUDE(<packageDir>/cmake/Dependencies.cmake)
            - Sets all package-specific options (see TriBITS Package Cache Variables)
        * For each <spkgDir> in all subpackages for this package:
            * INCLUDE(<packageDir>/<spkgDir>/cmake/Dependencies.cmake)
                - Sets all subpackage-specific options

When comparing the above reduced dependency processing to the Full Processing of TriBITS Project Files it is important to note that that several files are **not** processed in these cases. The files that are not processed include <projectDir>/Version.cmake, <repoDir>/Version.cmake and <repoDir>/cmake/CallbackSetupExtraOptions.cmake. Therefore, you can't put anything in these files that would impact the definition of TriBITS repositories, packages, TPLs, etc. Anything that would affect the dependencies data-structure that gets written out as <Project>PackageDependencies.xml must be contained in the files that are processed shown above.

Debugging issues with *Reduced Dependency Processing of TriBITS Project Files* is more difficult because one can not easily turn on *File Processing Tracing* like they can when doing the full CMake configure. However, options may be added to the various tools to show this file processing and help debug problems.

File Processing Tracing

In order to aid in debugging problems with configuration, TriBITS defines the CMake cache option `${PROJECT_NAME}_TRACE_FILE_PROCESSING`. When enabled, TriBITS will print out when any of the project-related, repository-related, or package-related file is being processed by TriBITS. When `${PROJECT_NAME}_TRACE_FILE_PROCESSING=ON`, lines starting with `"-- File Trace:"` are printed in the `cmake` STDOUT for files that TriBITS automatically processes where there may be any confusion about what files are processed and when.

For example, for TribitsExampleProject, the configure file trace for the configure command:

```
$ cmake \
  -DTribitsExProj_TRIBITS_DIR=<tribitsDir> \
  -DTribitsExProj_ENABLE_MPI=ON \
  -DTribitsExProj_ENABLE_ALL_PACKAGES=ON \
  -DTribitsExProj_ENABLE_TESTS=ON \
  -DTribitsExProj_TRACE_FILE_PROCESSING=ON \
  -DTribitsExProj_ENABLE_CPACK_PACKAGING=ON \
  -DTribitsExProj_DUMP_CPACK_SOURCE_IGNORE_FILES=ON \
  <tribitsDir>/doc/examples/TribitsExampleProject \
  | grep "^-- File Trace:"
```

looks something like:

```
-- File Trace: PROJECT    INCLUDE    [...]/Version.cmake
-- File Trace: REPOSITORY INCLUDE    [...]/cmake/CallbackSetupExtraOptions.cmake
-- File Trace: REPOSITORY INCLUDE    [...]/PackagesList.cmake
-- File Trace: REPOSITORY INCLUDE    [...]/TPLsList.cmake
-- File Trace: PACKAGE    INCLUDE    [...]/packages/simple_cxx/cmake/Dependencies.cmak
-- File Trace: PACKAGE    INCLUDE    [...]/packages/mixed_language/cmake/Dependencies.
-- File Trace: PACKAGE    INCLUDE    [...]/packages/package_with_subpackages/cmake/Dep
-- File Trace: PACKAGE    INCLUDE    [...]/packages/package_with_subpackages/A/cmake/D
-- File Trace: PACKAGE    INCLUDE    [...]/packages/package_with_subpackages/B/cmake/D
-- File Trace: PACKAGE    INCLUDE    [...]/packages/package_with_subpackages/C/cmake/D
-- File Trace: PACKAGE    INCLUDE    [...]/packages/wrap_external/cmake/Dependencies.c
-- File Trace: PROJECT    CONFIGURE  [...]/cmake/ctest/CTestCustom.cmake.in
-- File Trace: REPOSITORY READ       [...]/Copyright.txt
-- File Trace: REPOSITORY INCLUDE    [...]/Version.cmake
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/simple_cxx/CMakeLists.txt
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/simple_cxx/test/CMakeLists.txt
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/mixed_language/CMakeLists.txt
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/mixed_language/test/CMakeLists.txt
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/package_with_subpackages/CMakeList
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/package_with_subpackages/A/CMakeLi
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/package_with_subpackages/A/tests/C
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/package_with_subpackages/B/CMakeLi
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/package_with_subpackages/B/tests/C
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/package_with_subpackages/C/CMakeLi
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/package_with_subpackages/C/tests/C
-- File Trace: REPOSITORY INCLUDE    [...]/cmake/CallbackDefineRepositoryPackaging.cma
-- File Trace: PROJECT    INCLUDE    [...]/cmake/CallbackDefineProjectPackaging.cmake
```

However, every file that TriBITS processes is not printed in this file trace if it should be obvious that the file is being processed. For example, the package's configured header file created using TRIBITS_CONFIGURE_FILE() does not result in a file trace print statment because this is an unconditional command that is explicitly called in one of the the package's `CMakeLists.txt` files so it should be clear that this file is being processed.

3.3 Coexisting Projects, Repositories, and Packages

Certain simplifications are allowed when defining TriBITS projects, repositories and packages. The known allowed simplifications are described below.

TriBITS Repository == TriBITS Project: It is allowed for a TriBITS Project and a TriBITS Repository to be the same source directory and in fact this is the default for every TriBITS project (unless the the <projectDir>/cmake/NativeRepositoriesList.cmake is defined). In this case, the repository name and the project name are the same as well. This is quite common and is in fact the default that every TriBITS Project is also a TriBITS repository (and therefore must contain <repoDir>/PackagesList.cmake and <repoDir>/TPLsList.cmake files). This is the case, for example, with the the Trilinos and the TribitsExampleProject projects and repositories. In this case, the Project's and the Repository's `Version.cmake` and `Copyright.txt` files are one and the same, as they should be (see Project and Repositiory Versioning and Release Mode).

TriBITS Package == TriBITS Repository: It is also allowed for a TriBITS Repository to have only one package and to have that package be the base repository directory. The TriBITS Repository and the single TriBITS Package would typically have the same name in this case (but that is actually not required but it is confusing if they are not the same). For example, in the TriBITS test project `MockTrilinos`, the repostory and package `extraRepoOnePackage` are one in the same. In this case, the file `extraRepoOnePackage/PackagesList.cmake` looks like:

```
TRIBITS_DEFINE_REPOSITORY_PACKAGES(
  extraRepoOnePackage    .      ST
  )
```

This is used in the real TriBITS repository DataTransferKit.

However, to maximize flexibility, it is recommended that a TriBITS package and TriBITS repository not share the same directory.

TriBITS Package, TriBITS Repository, TriBITS Package sharing the same source directory: In the extreme, it is posisble to collapase a single TriBITS package, repository, and project into the same base source directory. However, in this case, the TriBITS Project name and the TriBITS Package name cannot be the same and some modifications and tricks are needed to allow this to work. One example of this is the TriBITS project and The TriBITS Test Package themselves, which are both rooted in the base `tribits` source directory. There are a few restructions and modifications needed to get this to work:

- The Project and Package names cannot be the same: In the case of the TriBITS project, the project name is `TriBITSProj` (as defined in `tribits/ProjectName.cmake`) and the package name is `TriBITS` (as defined in `tribits/PackagesList.cmake`).
- The base `CMakeLists.txt` file must be modified to allow it to be processed both as the base project `CMakeLists.txt` file and as the package's base `CMakeLists.txt` file: In the case of `tribits/CMakeLists.txt`, a big if statement is used.
- An extra subdirectory must be created for TriBITS package's binary directory: Because of directory-level targets like `${PROJECT_NAME}_libs` and `${PACKAGE_NAME}_libs`, a subdirectory for package's the binary directory must be created. This is simply done by overriding the binary directory name `${PACKAGE_NAME}_SPECIFIED_BINARY_DIR`. In the case of TriBITS, this is set to `tribits` in the `tribits/PackagesList.cmake` file.

Other than those modifications, a TriBITS project, repository, and package can all be rooted in the same source directory. However, as one can see above, to do so is a little messy and is not recommended. It was only done this way with the base TriBITS directory in order to maintain backward compatibility for the usage of TriBITS in existing TriBITS projects.

However, one possible use case for collapsing a project, repository, and package into a single base source directory would be to support the stand-alone build of a TriBITS package as its own entity that uses an installation of the TriBITS. If a given TriBITS package has no required upstream TriBITS package dependencies and minimal TPL dependencies (or only uses Standard TriBITS TPLs already defined in the `tribits/tpls/` directory), then creating a stand-alone project build of a loan TriBITS package requires fairly little extra overhead or duplication. However, as mentioned above, one cannot use the same name for the package and the project.

3.4 Standard TriBITS TPLs

TriBITS contains find modules for a few standard TPLs that are either integral to the TriBITS system or are likely to be used across many independent TriBITS repositories. The goal of maintaining a few of these in the later case under TriBITS is to enforce conformity in case these independent repositories are combined into a single meta-project.

The standard TriBITS TPLs are contained under the directory:

```
tribits/tpls/
```

The current list of standard TriBITS TPLs is:

```
FindTPLCUDA.cmake
FindTPLMPI.cmake
FindTPLPETSC.cmake
```

The TPLs `MPI` and `CUDA` are standard because they are special in that they define compilers and other special tools that are used in TRIBITS_ADD_LIBRARY(), TRIBITS_ADD_EXECUTABLE(), TRIBITS_ADD_TEST() and other commands.

These standard TPLs are used in a <repoDir>/TPLsList.cmake file as:

```
TRIBITS_DEFINE_REPOSITORY_TPLS(
  MPI    "${${PROJECT_NAME}_TRIBITS_DIR}/tpls/"  PT
  CUDA   "${${PROJECT_NAME}_TRIBITS_DIR}/tpls/"  ST
  ...
  )
```

Other than the special TPLs `MPI` and `CUDA`, other TPLs that are candidates to put into TriBITS are those that are likley to be used by different stand-alone TriBITS repositories that need to be combined into a single TriBITS meta-project. By using a standard TPL definition, it is guaranteed that the TPL used will be consistent with all of the repositories.

Note that just because packages in two repositories reference the same TPL does not necessarily mean that it needs to be a standard TriBITS TPL. For example, if the TPL `BLAS` is defined in an upstream repository (e.g. Trilinos), then a package in a downstream repository can list a dependency on the TPL `BLAS` without having to define its own `BLAS` TPL in its repository's <repoDir>/TPLsList.cmake file. For more details on TPLs, see TriBITS TPL.

4 Example TriBITS Projects

In this section, a few different example TriBITS projects and packages are previewed. All of these examples exist in the TriBITS source directory `tribits` itself so they are available to all users of TriBITS. These examples also provide a means to test the TriBITS system itself (see The TriBITS Test Package).

The first example covered is the bare bones TribitsHelloWorld example project. The second example covered in detail is TribitsExampleProject. This example covers all the basics for setting up a simple multi-package TriBITS project. The third example outlined is *MockTrilinos* which mostly exists to test the TriBITS system itself but use contains some nice examples of a few different TriBITS features and behaviors. The last example mentioned is The TriBITS Test Package itself which allows the TriBITS system to be tested and installed from any TriBITS project that lists it, including the `TriBITSProj` project itself (see Coexisting Projects, Repositories, and Packages).

The directory `tribits/doc/examples/` contains some other example TriBITS projects and repositories as well that are refered to in this and other documents.

4.1 TribitsHelloWorld

This is the simplest possible TriBITS project that you can imagine and is contained under the directory:

```
tribits/doc/examples/TribitsHelloWorld/
```

It contains only a single TriBITS package and no frills at all (does not support MPI or Fortran). However, it does show how minimal a TriBITS Project (which is also a TriBITS Repository) and a TriBITS Package can be and still show the value of TriBITS over raw CMake. The simple *HelloWorld* package is used to compare with the raw CMakeList.txt file in the `RawHeloWorld` example project in the TriBITS Overview document.

The directory structure for this examples shows what is necessary for a minimal TriBITS project:

```
TribitsHelloWorld/
  CMakeLists.txt
  PackagesList.cmake
  ProjectName.cmake
  README
  TPLsList.cmake
  hello_world/
    CMakeLists.txt
    cmake/
      Dependencies.cmake
    hello_world_lib.cpp
    hello_world_lib.hpp
    hello_world_main.cpp
    hello_world_unit_tests.cpp
```

This has all of the required TriBITS Project Core Files, TriBITS Repository Core Files, and TriBITS Package Core Files. It just build a simle library, a simple exectuable, a test exectuable, and the tests them as shown by the file `TribitsHelloWorld/hello_world/CMakeLists.txt` which is:

```
TRIBITS_PACKAGE(HelloWorld)
TRIBITS_ADD_LIBRARY(hello_world_lib
  HEADERS hello_world_lib.hpp SOURCES hello_world_lib.cpp)
TRIBITS_ADD_EXECUTABLE(hello_world NOEXEPREFIX SOURCES hello_world_main.cpp
  INSTALLABLE)
TRIBITS_ADD_TEST(hello_world NOEXEPREFIX PASS_REGULAR_EXPRESSION "Hello World")
TRIBITS_ADD_EXECUTABLE_AND_TEST(unit_tests SOURCES hello_world_unit_tests.cpp
  PASS_REGULAR_EXPRESSION "All unit tests passed")
TRIBITS_PACKAGE_POSTPROCESS()
```

The build and test of this simple project is tested in the TriBITS Package testing file:

```
tribits/doc/examples/UnitTests/CMakeLists.txt
```

Note that this little example is a fully functional TriBITS Repository and can be embedded in to a larger TriBITS meta-project and be seamlessly built along with any other such TriBITS-based software.

4.2 TribitsExampleProject

TribitsExampleProject in an example TriBITS Project and TriBITS Repository contained in the TriBITS source tree under:

```
tribits/doc/examples/TribitsExampleProject/
```

When this used as the base TriBITS project, this is the directory coresponds to `<projectDir>` and `<repoDir>` referenced in TriBITS Project Core Files and TriBITS Repository Core Files, respectively.

Several files from this project were used as examples in the section TriBITS Project Structure. Here, a fuller description is given of this project and how TriBITS works using it. From this simple example project, one can quickly see how the basic structural elements a TriBITS project, repository, and package (and subpackage) are pulled together.

The name of this project `PROJECT_NAME` given in its `TribitsExampleProject/ProjectName.cmake` file:

```
# Must set the project name at very beginning before including anything else
SET(PROJECT_NAME TribitsExProj)
```

The variable `PROJECT_NAME=TribitsExProj` is used to prefix with `"${PROJECT_NAME}_"` all of the projects global TriBITS variables like `TribitsExProj_ENABLE_TESTS`, `TribitsExProj_ENABLE_ALL_PACKAGES`, etc. Note, as shown in this example, the project name and the base project directory name do **not** need to match.

The directory structure and key files for this example project is shown in this partial list of **TribitsExampleProject Files and Directories**:

```
TribitsExampleProject/
  CMakeLists.txt
  Copyright.txt
  PackagesList.cmake
  ProjectName.cmake
  project-checkin-test-config.py
  TPLsList.cmake
  Version.cmake
  ...
  cmake/
    CallbackDefineProjectPackaging.cmake
    CallbackDefineRepositoryPackaging.cmake
    CallbackSetupExtraOptions.cmake
  packages/
    simple_cxx/
      CMakeLists.txt
      cmake/
        CheckFor__int64.cmake
        Dependencies.cmake
        SimpleCxx_config.h.in
      src/
        CMakeLists.txt
        SimpleCxx_HelloWorld.cpp
        SimpleCxx_HelloWorld.hpp
      test/
        CMakeLists.txt
        SimpleCxx_HelloWorld_Tests.cpp
    mixed_language/ ...
    package_with_subpackages/
      CMakeLists.txt
      cmake/
        Dependencies.cmake
      A/
        CMakeLists.txt
        cmake/
          Dependencies.cmake
        ...
      B/ ...
      C/ ...
    wrap_external/ ...
```

Above, the subdirectoires under `packages/` are sorted according to the order listed in the `TribitsExampleProject/PackagesList.cmake` file:

```
TRIBITS_DEFINE_REPOSITORY_PACKAGES(
  SimpleCxx               packages/simple_cxx                 PT
  MixedLanguage           packages/mixed_language             PT
  PackageWithSubpackages  packages/package_with_subpackages   PT
  WrapExternal            packages/wrap_external              PT
  )

TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS(WrapExternal Windows)
```

From this file, we get the list of top-level packages `SimpleCxx`, `MixedLanguage`, `PackageWithSubpackages`, and `WrapExternal` (and their base package directories and testing group, see <repoDir>/PackagesList.cmake).

The full listing of package files in TribitsExampleProject Files and Directories is only shown for the `SimpleCxx` package directory `packages/simple_cxx/`. This gives `<packageDir> = <repoDir>/packages/simple_cxx` for the package `PACKAGE_NAME = SimpleCxx` referenced in TriBITS Package Core Files. As explained there, the files <packageDir>/cmake/Dependencies.cmake and <packageDir>/CMakeLists.txt must exist for every package directory listed in <repoDir>/PackagesList.cmake and we see these files under in the directory `packages/simple_cxx/`. The package `SimpleCxx` does not have any upstream SE package dependencies.

Now consider the example top-level package `PackageWithSubpackages` which, as the name suggests, is broken down into subpackages. The `PackageWithSubpackages` dependencies file:

```
TribitsExampleProject/packages/package_with_subpackages/cmake/Dependencies.cmake
```

with contents:

```
TRIBITS_DEFINE_PACKAGE_DEPENDENCIES(
  SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS
    SubpackageA    A    PT  REQUIRED
    SubpackageB    B    PT  REQUIRED
    SubpackageC    C    PT  REQUIRED
  )
```

references the three subpackage with sub-directories `<spkgDir>` = `A`, `B`, and `C` under the parent package directory `packages/package_with_packages/` which are shown in TribitsExampleProject Files and Directories. This gives another set of three SE packages `PackageWithSubpackagesSubpackageA`, `PackageWithSubpackagesSubpackaeB`, and `PackageWithSubpackagesSubpackageC`. Combining `<packageDir> = packages/package_with_packages` and `<spkgDir>` for each subpackage gives the subpackage directories:

```
TribitsExampleProject/packages/package_with_subpackages/A/
TribitsExampleProject/packages/package_with_subpackages/B/
TribitsExampleProject/packages/package_with_subpackages/C/
```

Together with the top-level parent SE package `PackageWithSubpackages` itself, this top-level package provides four SE packages giving the final list of SE packages provided by this TriBITS repo as:

```
SimpleCxx MixedLanguage PackageWithSubpackagesSubpackageA \
  PackageWithSubpackagesSubpackaeB PackageWithSubpackagesSubpackaeC \
  PackageWithSubpackages WrapExternal 7
```

The above list of SE packages is shown formatted this way since this is the format that the SE packages are printed by TriBITS in the `cmake` STDOUT on the line starting with `"Final set of non-enabled SE packages:"` when no packages are enabled (see Selecting the list of packages to enable). TriBITS defines enable/disable cache variables for each of the defined SE packages like `TribitsExProj_ENBLE_SimpleCxx`, `TribitsExProj_ENBLE_PackageWithSubpackagesSubpackageA`, and defines all the variables listed in TriBITS Package Cache Variables that are settable by the users or by the dependnecy logic described in section Package Dependencies and Enable/Disable Logic.

Hopefully this simple project shows how what is listed in files:

- <repoDir>/PackagesList.cmake,
- <packageDir>/cmake/Dependencies.cmake, and
- <packageDir>/<spkgDir>/cmake/Dependencies.cmake

is used to specify the packages and SE packages in a TriBITS project and repository. More details about the contents of the `Dependencies.cmake` files is described in the section Package Dependencies and Enable/Disable Logic.

When starting a new TriBITS project, repository, or package, one should consider basing these on the examples in this project. In fact, the skeletons for any of the

- TriBITS Project Core Files,
- TriBITS Repository Core Files,
- TriBITS Package Core Files, or
- TriBITS Subpackage Core Files

should be copied from this example project as they represent best practice when using TriBITS for the typical use cases.

4.3 MockTrilinos

The TriBITS project `MockTrilinos` is contained under the directory:

```
tribits/package_arch/UnitTests/MockTrilinos/
```

This TriBITS project is not a full TriBITS project (i.e. it does not build anything). Instead, it is used to test the TriBITS system using tests defined in the The TriBITS Test Package. The `MockTrilinos` project is actually given the name `PROJECT_NAME = Trilinos` and contains a subset of packages with slightly modified dependencies from a snapshot of the real Trilinos project from May 2009. The list of packages in:

```
tribits/package_arch/UnitTests/MockTrilinos/PackagesList.cmake
```

is:

```
TRIBITS_DEFINE_REPOSITORY_PACKAGES(
  TrilinosFramework   cmake                       PT
  Teuchos             packages/teuchos            PT
  RTOp                packages/rtop               PT
  Epetra              packages/epetra             PT
  Zoltan              packages/zoltan             PT
  Shards              packages/shards             PT
  Triutils            packages/triutils           PT
  Tpetra              packages/tpetra             PT
  EpetraExt           packages/epetraext          PT
  Stokhos             packages/stokhos            EX
  Sacado              packages/sacado             ST
  Thyra               packages/thyra              PT
  Isorropia           packages/isorropia          PT
  AztecOO             packages/aztecoo            PT
  Galeri              packages/galeri             PT
  Amesos              packages/amesos             PT
  Intrepid            packages/intrepid           PT
  Ifpack              packages/ifpack             PT
  ML                  packages/ml                 PT
  Belos               packages/belos              ST
  Stratimikos         packages/stratimikos        PT
  RBGen               packages/rbgen              PT
  Phalanx             packages/phalanx            ST
  Panzer              packages/panzer             ST
  )

# NOTE: Sacado was really PT but for testing purpose it is made ST
# NOTE: Belos was really PT but for testing purpose it is made ST

TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS(ML BadSystem1)
TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS(Ifpack BadSystem1 BadSystem2)
```

All of the package directories listed above have `cmake/Dependenices.cmake` files but generally do not have `CMakeLists.txt` files since most of the testing of `MockTrilinos` just involves dependency handling.

`MockTrilinos` also contains a number of extra TriBITS repositories used in various tests. These extra repositories offer examples of different types of TriBITS repositories like:

- `extraRepoOnePackage`: Contains just the single package `extraRepoOnePackage` which is defined in the base repository directory.
- `extraRepoOnePackageThreeSubpackages`: Contains just the single package `extraRepoOnePackageThreeSubpackages` which is defined in the base repository directory but is broken up into subpackages.
- `extraRepoTwoPackages`: Contains just two packages but provides an exmaple of defining multiple repositories with possible missing required and optional upstream packages (see Multi-Repository Support).
- `extraTrilinosRepo`: Just a typical extra repo with add-on packages and new TPLs defined that depends on a few `MockTrilinos` packages.

New test extra repostories are added when new types of tests are needed that would require new package and TPL dependency structures since existing dependency tests based on `MockTrilinos` are expensive to change by their very nature.

The reason that the `MockTrilinos` test project is mentioned in this developers guide is because it contains a greater variety of packages, subpackages, and TPLs with a greater variety of different types of dependencies. This variety is is needed to fully test the TriBITS system but this project and the tests also serve as examples and extra documentation for the behavior of the TriBITS system. Several of the examples referenced in this document come from `MockTrilinos`.

Most of the dependency tests involving `MockTrilinos` are specified in:

```
tribits/package_arch/UnitTests/DependencyUnitTests/CMakeLists.txt
```

A great deal about the current behavior of TriBITS Package Dependencies and Enable/Disable Logic can be learned from inspecting the tests defined in this `CMakeLists.txt` file. There are also some faster-running unit tests involving `MockTrilinos` defined in the file:

```
tribits/package_arch/UnitTests/TribitsAdjustPackageEnables_UnitTests.cmake
```

4.4 ReducedMockTrilinos

The TriBITS project `ReducedMockTrilinos` is contained under the directory:

```
tribits/package_arch/UnitTests/ReducedMockTrilinos/
```

It is a scaled-down version of the MockTrilinos test project with just a handful of packages and some modified dependencies. Its primary purpose is to be used for examples in the section Package Dependencies and Enable/Disable Logic and to test a few features of the TriBITS system not tested in other tests.

The list of packages in:

```
tribits/package_arch/UnitTests/ReducedMockTrilinos/PackagesList.cmake
```

is:

```
TRIBITS_DEFINE_REPOSITORY_PACKAGES(
  Teuchos             packages/teuchos              PT
  RTOp                packages/rtop                 PT
  Epetra              packages/epetra               PT
  Triutils            packages/triutils             ST
  EpetraExt           packages/epetraext            ST
  Thyra               packages/thyra                PT
  )
```

All of the listed packages are standard TriBITS packages except for the mock `Thyra` package which is broken down into subpackages. More details of this example project are described in Package Dependencies and Enable/Disable Logic.

4.5 The TriBITS Test Package

The last TriBITS example mentioned here is the TriBITS test package named (appropriately) `TriBITS` itself. The directory for the `TriBITS` package is the base TriBITS source directory `tribits`. This allows any TriBITS project to add testing for the TriBITS system by just listing this package and its directory in its repository's <repoDir>/PackagesList.cmake file. For example, the Trilinos repository which currently snaphsots the TriBITS source tree lists the `TriBITS` package with:

```
TRIBITS_DEFINE_REPOSITORY_PACKAGES(
  TriBITS   cmake/tribits  PT   # Only tests, no libraries/capabilities!
  ...
  )
```

No downstream packages list a dependency on `TriBITS` in their <packageDir>/cmake/Dependencies.cmake files. Listing the `TriBITS` package in only done in the `PackagesList.cmake` file for testing TriBITS.

Other TriBITS projects/repositories that don't snapshot TriBITS but also want to test TriBITS (perhaps just to mine the running tests for examples) can do so by including the `TriBITS` test package in their `PackagesList.cmake` file using:

```
TRIBITS_DEFINE_REPOSITORY_PACKAGES(
  TriBITS   ${${PROJECT_NAME}_TRIBITS_DIR}   PT
  ...
  )
```

Once the `TriBITS` test package is added to the list of project/repository packages, it can be enabled just like any other package by adding the following to the `cmake` command-line options:

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

One can then inspect the added tests prefixed by `"TriBITS_"` to see what tests are defined and how they are run. There is a wealth of information about the TriBITS system embedded in these tests and where documentation and these tests disagreed, believe the tests!

5 Package Dependencies and Enable/Disable Logic

Arguably, the more important feature/aspect of the TriBITS system is the partitioning of a large software project into packages and managing the dependencies between these packages to support building, testing, and depoying different pieces as needed. This is especially useful in incremental CI testing of large projects. However, this is also a critical component in creating and maintaining Self Sustaining Software (see the TriBITS Lifecycle Model). The fundamental mechanism for breaking up a large software into manageable pieces is to partition the software into different TriBITS Packages and then define the dependencies between these packages (which are defined inside of the <packageDir>/cmake/Dependencies.cmake files for each package).

Note that the basic idea of breaking up a large set of software into pieces, defining dependencies between the pieces, and then applying algorithms to manipulate the dependency data-structures is nothing new. If fact, nearly every binary package deployment system provided in various Linux OS distributions have the concept of packages and dependnecies and will automatically install all of the necessary upstream dependencies when a downstream dependency install is requested. The main difference (and the added complexity) with TriBITS is that it can handle both required and optional dependenices since it can build from source. A binary package installation system, however, typically can't support optional dependencies because only pre-built binary libraries and tools are aviable to install.

5.1 Example ReducedMockTrilinos Project Dependency Structure

To demonstrate the TriBITS package and TPL dependency handling system, the small simple ReducedMockTrilinos project is used. The list of packages for this project is defined in `ReducedMockTrilinos/PackagesList.cmake` (see <repoDir>/PackagesList.cmake) which contents:

```
TRIBITS_DEFINE_REPOSITORY_PACKAGES(
  Teuchos             packages/teuchos               PT
  RTOp                packages/rtop                  PT
  Epetra              packages/epetra                PT
  Triutils            packages/triutils              ST
  EpetraExt           packages/epetraext             ST
  Thyra               packages/thyra                 PT
  )
```

All of the listed packages are standard TriBITS packages except for the mock `Thyra` package which is broken down into subpackages as shown in `thyra/cmake/Dependnecies.cmake` (see <packageDir>/cmake/Dependencies.cmake) which is:

```
TRIBITS_DEFINE_PACKAGE_DEPENDENCIES(
  SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS
    CoreLibs      src                  PT  REQUIRED
    GoodStuff     good_stuff           ST  OPTIONAL
    CrazyStuff    crazy_stuff          EX  OPTIONAL
    Epetra        adapters/epetra      PT  OPTIONAL
    EpetraExt     adapters/epetraext   ST  OPTIONAL
  )
```

This gives the full list of top-level packages:

```
Teuchos RTOp Epetra Triutils EpetraExt Thyra
```

Adding in the subpackages defined in the top-level `Thyra` package, the full set of TriBITS SE Packages for this project is:

```
Teuchos RTOp Epetra Triutils EpetraExt ThyraCoreLibs ThyraGoodStuff \
  ThyraCrazyStuff ThyraEpetra ThyraEpetraExt Thyra
```

Note that one can see this full list of top-level packages and SE packages in the lines starting with:

```
Final set of non-enabled packages:
Final set of non-enabled SE packages:
```

respectively, when configuring with no package enables as shown in the example Default configure with no packages enabled on input.

The list of TriBITS TPLs for this example project given in `ReducedMockTrilinos/TPLsList.cmake` (see <repoDir>/TPLsList.cmake) is:

```
TRIBITS_DEFINE_REPOSITORY_TPLS(
  MPI             "${${PROJECT_NAME}_TRIBITS_DIR}/tpls/"    PT
  BLAS            cmake/TPLs/    PT
  LAPACK          cmake/TPLs/    PT
  Boost           cmake/TPLs/    ST
  UMFPACK         cmake/TPLs/    ST
  AMD             cmake/TPLs/    EX
  PETSC           cmake/TPLs/    ST
  )
```

Take note of the testing group (i.e. `PT`, `ST`, or `EX`) assigned to each SE package and TPL plays a signficiant role in how the TriBITS dependency system handles enables and disable of the SE packages and TPLs.

The dependency structure of this simple TriBITS project is shown below in ReducedMockTrilinos Dependencies.

ReducedMockTrilinos Dependencies:

```
Printing package dependencies ...

-- Teuchos_LIB_REQUIRED_DEP_TPLS: BLAS LAPACK
-- Teuchos_LIB_OPTIONAL_DEP_TPLS: Boost MPI

-- RTOp_LIB_REQUIRED_DEP_PACKAGES: Teuchos

-- Epetra_LIB_REQUIRED_DEP_TPLS: BLAS LAPACK
-- Epetra_LIB_OPTIONAL_DEP_TPLS: MPI

-- Triutils_LIB_REQUIRED_DEP_PACKAGES: Epetra

-- EpetraExt_LIB_REQUIRED_DEP_PACKAGES: Teuchos Epetra
-- EpetraExt_LIB_OPTIONAL_DEP_PACKAGES: Triutils
-- EpetraExt_LIB_OPTIONAL_DEP_TPLS: UMFPACK AMD PETSC

-- ThyraCoreLibs_LIB_REQUIRED_DEP_PACKAGES: Teuchos RTOp

-- ThyraGoodStuff_LIB_REQUIRED_DEP_PACKAGES: ThyraCoreLibs

-- ThyraCrazyStuff_LIB_REQUIRED_DEP_PACKAGES: ThyraGoodStuff

-- ThyraEpetra_LIB_REQUIRED_DEP_PACKAGES: Epetra ThyraCoreLibs

-- ThyraEpetraExt_LIB_REQUIRED_DEP_PACKAGES: ThyraEpetra EpetraExt

-- Thyra_LIB_REQUIRED_DEP_PACKAGES: ThyraCoreLibs
-- Thyra_LIB_OPTIONAL_DEP_PACKAGES: ThyraGoodStuff ThyraCrazyStuff ThyraEpetra ThyraEp
```

The above dependency structure printout is produced by configuring with `${PROJECT_NAME}_DUMP_PACKAGE_DEPENDENCIES=ON` (which also results in more dependency information than what is shown above, e.g. like computed forward package dependencies). Note that the top-level SE package `Thyra` is shown to depend on its subpackages (not the other way around). (Many people are confused about this the nature of the dependencies between packages and subpackages.)

A number of user-setable cache variables determine what SE packages (and TPLs) and what tests and examples get enabled. These cache variables are described in Selecting the list of packages to enable and are described below. Also, the assigned SE Package Test Group (i.e. PT, ST, and EX) also affects what packages get enabled or disabled.

Any of these SE packages can be enabled or disabled with `${PROJECT_NAME}_ENABLE_<TRIBITS_PACKAGE>=(ON|OFF)` (the default enable is typically `""`, see PT/ST SE packages given default unset enable/disable state). For `ReducedMockTrilinos`, this gives the enable/disable cache variables (with the initial default values):

```
Trilinos_ENABLE_Teuchos=""
Trilinos_ENABLE_RTOp""
Trilinos_ENABLE_Epetra=""
Trilinos_ENABLE_Triutils=""
Trilinos_ENABLE_EpetraExt=""
Trilinos_ENABLE_ThyraCore=""
Trilinos_ENABLE_ThyraGoodStuff=""
Trilinos_ENABLE_ThyraCrazyStuff="OFF"  # Because it is 'EX'
Trilinos_ENABLE_ThyraEpetra=""
Trilinos_ENABLE_ThyraEpetraExt=""
```

Every TriBITS SE package is assumed to have tests and/or examples so TriBITS defines the following cache varaibles as well (with the initial default values):

```
Teuchos_ENABLE_TESTS=""
RTOp_ENABLE_TESTS=""
```

```
Epetra_ENABLE_TESTS=""
Triutils_ENABLE_TESTS=""
EpetraExt_ENABLE_TESTS=""
ThyraCoreLibs_ENABLE_TESTS=""
ThyraGoodStuff_ENABLE_TESTS=""
ThyraEpetra_ENABLE_TESTS=""
ThyraEpetraExt_ENABLE_TESTS=""
Thyra_ENABLE_TESTS=""
```

NOTE: TriBITS only sets the variables `<TRIBITS_PACKAGE>_ENABLE_TESTS` into the cache if the SE package `<TRIBITS_PACKAGE>` becomes enabled at some point. This cuts down the clutter in the CMake cache for large projects with lots of packages where the user only enbles a subset of the packages.

NOTE: TriBITS also defines the cache varaibles `<TRIBITS_PACKAGE>_ENABLE_EXAMPLES` for each enabled TriBITS package which is handled the same way as the TEST variables.

Also, every defined TPL is given its own `TPL_ENABLE_<TRIBITS_TPL>` enable/disable cache variable. For the TPLs in `ReducedMockTrilinos`, this gives the enable/disable cache variables (with default values):

```
TPL_ENABLE_MPI=""
TPL_ENABLE_BLAS=""
TPL_ENABLE_LAPACK=""
TPL_ENABLE_Boost=""
TPL_ENABLE_UMFPACK=""
TPL_ENABLE_AMD=""
TPL_ENABLE_PETSC=""
```

In addition, for every optional SE package and TPL dependency, TriBITS defines a cache variable `<TRIBITS_PACAKGE>_ENABLE_<OPTIONAL_DEP>`. For the optional dependenices shown in ReducedMockTrilinos Dependencies, that gives the additional cache variables (with default values):

```
Teuchos_ENABLE_Boost=""
Teuchos_ENABLE_MPI=""
Teuchos_ENABLE_Boost=""
Epetra_ENABLE_MPI=""
EpetraExt_ENABLE_Triutils=""
EpetraExt_ENABLE_UMFPACK=""
EpetraExt_ENABLE_AMD=""
EpetraExt_ENABLE_PETSC=""
Thyra_ENABLE_ThyraGoodStuff=""
Thyra_ENABLE_ThyraCrazytuff=""
Thyra_ENABLE_ThyraEpetra=""
Thyra_ENABLE_ThyraEpetraExt=""
```

The above optional package-specific cache variables allow one to control whether or not support for upstream dependency X is turned on in package Y independent of whether or not X and Y are themselves both enabled. For example, if the packages `Triutils` and `EpetraExt` are both enabled, one can explicitly disable support for the optional dependency `Triutils` in `EpetraExt` by setting `EpetraExt_ENABLE_Triutils=OFF`. One may want to do this for several reasons but the bottom line is that this gives the user more detailed control over package dependenices. See the TriBITS Dependency Handling Behaviors and Explicit disable of an optional package dependency for more discussion and examples.

Before getting into specific examples for some Example Enable/Disable Use Cases, some of the TriBITS Dependency Handling Behaviors are defined below.

5.2 TriBITS Dependency Handling Behaviors

Below, some of the rules and behaviors of the TriBITS dependency management system are described. Examples refer to the Example ReducedMockTrilinos Project Dependency Structure. More detailed examples of these behaviors are given in the section Example Enable/Disable Use Cases.

In brief, these rules/behaviors are:

1) PT/ST SE packages given default unset enable/disable state
2) EX SE packages disabled by default
3) SE package enable triggers auto-enables of upstream dependenices
4) SE package disable triggers auto-disables of downstream dependencies
5) PT/ST TPLs given default unset enable/disable state
6) EX TPLs given default unset enable/disable state
7) Required TPLs are auto-enabled for enabled SE packages
8) Optional TPLs only enabled explicitly by the user
9) TPL disable triggers auto-disables of downstream dependencies
10) Disables trump enables where there is a conflicit
11) Enable/disable of parent package is enable/disable for subpackages
12) Subpackage enable does not auto-enable the parent package
13) Support for enabled optional SE package/TPL is enabled by default
14) Support for optional SE package/TPL can be explicitly disabled
15) Explicit enable of optional SE package/TPL support auto-enables SE package/TPL
16) ST SE packages only auto-enabled if ST code is enabled
17) <Project>_ENABLE_ALL_FORWARD_DEP_PACKAGES downstream packages/tests
18) <Project>_ENABLE_ALL_PACKAGES enables all PT (cond. ST) SE packages
19) <Project>_ENABLE_TESTS only enables explicitly enabled SE package tests
20) If no SE packages are enabled, nothing will get built
21) TriBITS prints all enables and disables to STDOUT
22) TriBITS auto-enables/disables done using non-cache local variables

In more detail, these rules/behaviors are:

1) **PT/ST SE packages given default unset enable/disable state**: An SE package `<TRIBITS_PACKAGE>` with testing group `PT` or `ST` is given an **unset enable/disable state by default** (i.e. `${PROJECT_NAME}_ENABLE_<TRIBITS_PACKAGE>=""`). For example, the `PT` package `Teuchos` is not enabled or disabled by default and is given the initial value `Trilinos_ENABLE_Teuchos=""`. This allows `PT`, and `ST` packages to be enabled or disabled using other logic defined by TriBITS which is described below. For an example, see Default configure with no packages enabled on input.

2) **EX SE packages disabled by default**: An SE package `<TRIBITS_PACKAGE>` with testing group `EX` is **disabled by default** (i.e. `${PROJECT_NAME}_ENABLE_<TRIBITS_PACKAGE>=OFF`). This results in all required downstream SE packages to be disabled by default. However, the user can explicitly set `${PROJECT_NAME}_ENABLE_<TRIBITS_PACKAGE>=ON` for an `EX` package and it will be enabled (unless one of its required dependencies are not enabled for some reason). For an example, see Default configure with no packages enabled on input.

3) **SE package enable triggers auto-enables of upstream dependenices**: Any SE package `<TRIBITS_PACKAGE>` can be explictly **enabled** by the user by setting the cache variable `${PROJECT_NAME}_ENABLE_<TRIBITS_PACKAGE>=ON` (e.g. `Trilinos_ENABLE_EpetraExt=ON`). When a package is enabled in this way, the TriBITS system will try to enable all of the required upstream SE packages and TPLs defined by the package (specified in its `Dependencies.cmake` file). If an enabled SE package can't be enabled and has to be disabled, either a warning is printed or processing will stop with an error (depending on the value of `${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES`, see below). In addition, if

${PROJECT_NAME}_ENABLE_ALL_OPTIONAL_PACKAGES=ON, then TriBITS will try to enable all of the specified optional PT SE packages as well (and also optional upstream ST SE packages as well if ${PROJECT_NAME}_SECONDARY_TESTED_CODE=ON). For an example, see Explicit enable of a package, its tests, an optional TPL, with ST enabled.

4) **SE package disable triggers auto-disables of downstream dependencies**: Any SE package <TRIBITS_PACKAGE> can be explictly **disabled** by the user by setting the cache variable ${PROJECT_NAME}_ENABLE_<TRIBITS_PACKAGE>=OFF (e.g. Trilinos_ENABLE_Teuchos=OFF). When an SE package is explicitly disabled, it will result in the disable of all downstream SE packages that have required dependency on it. It will also disable optional support for the disabled packages in downstream packages that list it as an optional dependency. For an example, see Explicit disable of a package.

5) **PT/ST TPLs given default unset enable/disable state**: A TriBITS TPL <TRIBITS_TPL> with testing group PT or ST is given an **unset enable/disable state by default** (i.e. TPL_ENABLE_<TRIBITS_TPL>=""). For example, the PT TPL BLAS is not enabled or disabled by default (i.e. TPL_ENABLE_BLAS="") . This allows PT, and ST TPLs to be enabled or disabled using other logic. For an example, see Default configure with no packages enabled on input.

6) **EX TPLs given default unset enable/disable state**: A TriBITS TPL <TRIBITS_TPL> with testing group EX (as is for *PT*' and EX TPLs), is given an **unset enable/disable state by default** (i.e. TPL_ENABLE_<TRIBITS_TPL>""). This is different behavior than for EX SE packages described above which provides an initial hard disable. However, since TriBITS will never automatically enable an optional TPL (see below) and since only downstream EX SE packages are allowed to have a required dependenices on an EX TPL, there is no need to set the default enable for an EX TPL to OFF. For an example, see Default configure with no packages enabled on input.

7) **Required TPLs are auto-enabled for enabled SE packages**: All TPLs listed as required TPL dependencies for the final set of enabled SE packages are **set to enabled** (i.e. TPL_ENABLE_<TRIBITS_TPL>=ON) for the final set of enabled packages (unless the listed TPLs are already explicit disabled). For example, if the Epetra package is enabled, then that will trigger the enable of its required TPLs BLAS and LAPACK. For an example, see Explicit enable of a package and its tests.

8) **Optional TPLs only enabled explicitly by the user**: Optional TPLs with testing group PT, ST or EX will only be enabled if they are explicitly enabled by the user. For example, just because the package Teuchos is enabled, the optional TPLs Boost and MPI will **not** be enabled by default. To enable the optional TPL Boost, for example, and enable support for Boost in the Teuchos package, the user must explicitly set TPL_ENABLE_Boost=ON. For an example, see Explicit enable of a package, its tests, an optional TPL, with ST enabled.

9) **TPL disable triggers auto-disables of downstream dependencies**: Any TPLs that are explicitly disabled (i.e. TPL_ENABLE_<TRIBITS_TPL>=OFF) will result in the disable of all downstream dependent SE packages that have a required dependency on the TPL. For example, if the user sets TPL_ENABLE_LAPACK=OFF, then this will result in the disable of SE packages Teuchos and Epetra, and all of the required SE packages downstream from them (see ???). Also, the explicitly disabled TPL will result in disable of optional support in all downstream SE packages. For example, if the user sets TPL_ENABLE_MPI=OFF, then TriBITS will automatically set Teuchos_ENABLE_MPI=OFF and Epetra_ENABLE_MPI=OFF. For examples, see Explicit disable of an optional TPL and Explicit disable of a required TPL.

10) **Disables trump enables where there is a conflicit** and TriBITS will never override a disable in order to satisfy some dependency. For example, if the user sets Trilinos_ENABLE_Teuchos=OFF and Trilinos_ENABLE_RTOp=ON, then TriBITS will **not** override the disable of Teuchos in order to satisfy the required dependency of RTOp. In cases such as this, the behavior of the TriBITS dependency adjustment system will depend on the setting of the top-level user cache variable ${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES:

    - If ${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES=ON: TriBITS will disable explicit enable and continue on. i.e., TriBITS will override Trilinos_ENABLE_RTOp=ON and set Trilinos_ENABLE_RTOp=OFF and print a verbose warning to STDOUT.

- If `${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES=OFF`: TriBITS will generate a detailed error message and abort processing. i.e., TriBITS will report that `RTOp` is enabled but the required SE package `Teuchos` is disabled and therefore `RTOp` can't be enabled and processing must stop.

For an example, see Conflicting explicit disable of a package.

11) **Enable/disable of parent package is enable/disable for subpackages**: An explicit enable/disable of a top-level parent package with subpackages with `${PROJECT_NAME}_ENABLE_<TRIBITS_PACKAGE>=(ON|OFF)` is equivanent to the explicit enable/disable of all of the parent package's subpackages. For example, explicitly setting `Trilinos_ENABLE_Thyra=ON` is equivalent to explicitly setting:

```
Trilinos_ENABLE_ThyraCoreLibs=ON
Trilinos_ENABLE_ThyraGoodStuff=ON   # Only if enabling ST code!
Trilinos_ENABLE_ThyraEpetra=ON
Trilinos_ENABLE_ThyraEpetraExt=ON   # Only if enabling ST code!
```

(Note that `Trilinos_ENABLE_ThyraCrazyStuff` is **not** set to `ON` because it is aready set to `OFF` by default, see EX SE packages disabled by default.) Likewise, explicitly setting `Trilinos_ENABLE_Thyra=OFF` is equivalent to explicitly setting all of the `Thyra` subpackages to `OFF` at the outset. See an exmaple, see Explicit enable of a package and its tests.

12) **Subpackage enable does not auto-enable the parent package**: Enabling an SE package that is a subpackage does **not** automatically enable the parent package (except for at the very end, mostly just for show). For example, enabling the SE package `ThyraEpetra` does not result in enable of the parent `Thyra` package, (except when `${PROJEC_NAME}_ENABLE_ALL_FORWARD_DEP_PACKAGES=ON` for course). For an example, see Explicit enable of a subpackage. This means that if a downstream package declares a dependency on the SE package `ThyraEpetra`, but not the parent package `Thyra`, then the `Thyra` package (and its other subpackages and their dependencies) will not get auto-enabled. This is a key aspect of the SE package managment system. For an example, see Explicit enable of a subpackage.

13) **Support for enabled optional SE package/TPL is enabled by default**: For an SE package `<TRIBITS_PACKAGE>` with an optional dependency on an upstream SE package or TPL `<TRIBITS_DEP_PACKAGE_OR_TPL>`, TriBITS will automatically set the inter-enable variable `<TRIBITS_PACKAGE>_ENABLE_<TRIBITS_DEP_PACKAGE_OR_TPL>=ON` if `<TRIBITS_PACKAGE>` and `<TRIBITS_DEP_PACKAGE_OR_TPL>` are **both enabled**. This is obviously the logical behavior. For an example, see Explicit enable of a package, its tests, an optional TPL, with ST enabled.

14) **Support for optional SE package/TPL can be explicitly disabled:** Even through TriBITS will automatically set `<TRIBITS_PACKAGE>_ENABLE_<TRIBITS_DEP_PACKAGE_OR_TPL>=ON` by default if `<TRIBITS_PACKAGE>` and `<TRIBITS_DEP_PACKAGE_OR_TPL>` are both enabled (as described above) the user can explicitly set `<TRIBITS_PACKAGE>_ENABLE_<TRIBITS_DEP_PACKAGE_OR_TPL>=OFF` which will turn off optional support for the SE package or TPL `<TRIBITS_DEP_PACKAGE_OR_TPL>` in the SE package `<TRIBITS_PACKAGE>`. For examples, see Explicit disable of an optional package dependency and Explicit disable of an optional TPL dependency.

15) **Explicit enable of optional SE package/TPL support auto-enables SE package/TPL**: If the user explicitly enabled TriBITS SE package `<TRIBITS_PACKAGE>` and explicitly sets `<TRIBITS_PACKAGE>_ENABLE_<TRIBITS_DEP_PACKAGE_OR_TPL>=ON` on input, then that will automatically enable the SE package or TPL `<TRIBITS_DEP_PACKAGE_OR_TPL>``(and all of its upstream dependneices accordingly). For example, if the user sets ``Trilinos_ENABLE_EpetraExt=ON` and `EpetraExt_ENABLE_Triutils=ON`, then that will result in the auto-enable of `Trilinos_ENABLE_Triutils=ON` reguardless of the value of `${PROJECT_NAME}_ENABLE_ALL_OPTIONAL_PACKAGES` or `${PROJECT_NAME}_SECONDARY_TESTED_CODE`. This true even if the optional SE package or TPL is `EX`. For example, setting `Thyra_ENABLE_ThyraCrazyStuff=ON` will result in the enabling of the `EX` package `ThyraCrazyStuff`. However, always remember that Disables trump enables where there is a conflicit. For examples, see Explicit enable of an optional package dependency and Explicit disable of an optional TPL dependency.

16) **ST SE packages only auto-enabled if ST code is enabled**: TriBITS will only enable an optional `ST` SE package when `${PROJECT_NAME}_ENABLE_ALL_OPTIONAL_PACKAGES=ON` if `${PROJECT_NAME}_SECONDARY_TESTED_CODE=ON` is also set. If an optional `ST` upstream dependent SE package is not enabled due to `${PROJECT_NAME}_SECONDARY_TESTED_CODE=OFF`, then a one-line warning is printed to STDOUT. The TriBITS default is `${PROJECT_NAME}_ENABLE_ALL_OPTIONAL_PACKAGES=ON`. This helps to avoid problems when users try to set a permuation of enables/disables which is not regularly tested. For examples, see Explicit enable of a package and its tests and Explicit enable of a package, its tests, an optional TPL, with ST enabled.

17) **<Project>_ENABLE_ALL_FORWARD_DEP_PACKAGES downstream packages/tests**: Setting the user cache-variable `${PROJECT_NAME}_ENABLE_ALL_FORWARD_PACKAGES=ON` will result in the downstream `PT` SE packages and tests to be enabled (and all `PT` and `ST` SE packages and tests when `${PROJECT_NAME}_SECONDARY_TESTED_CODE=ON`) for all explicitly enabled SE packages. For example, configuring with `Trilinos_ENABLE_Epetra=ON`, `Trilinos_ENABLE_TESTS=ON`, and `Trilinos_ENABLE_ALL_FORWARD_PACKAGES=ON` will result the package enables (and test and example enables) for the SE packages `Triutils`, `EpetraExt`, `ThyraCoreLibs`, `ThyraEpetra` and `Thyra`. For an example, see Explicit enable of a package and downstream packages and tests.

18) **<Project>_ENABLE_ALL_PACKAGES enables all PT (cond. ST) SE packages**: Setting the user cache-variable `${PROJECT_NAME}_ENABLE_ALL_PACKAGES=ON` will result in the enable of all `PT` SE packages when `${PROJECT_NAME}_SECONDARY_TESTED_CODE=OFF` and all `PT` and `ST` SE packages when `${PROJECT_NAME}_SECONDARY_TESTED_CODE=ON`. For an example, see Enable all packages.

19) **<Project>_ENABLE_TESTS only enables explicitly enabled SE package tests**: Setting `${PROJECT_NAME}_ENABLE_TESTS=ON` will **only enable tests for explicitly enabled SE packages**. For example, configuring with `Trilinos_ENABLE_RTOp=ON` and `Trilinos_ENABLE_TESTS=ON` will only result in the enable of tests for `RTOp`, not `Teuchos` (even through TriBITS will enable `Teuchos` because it is a required dependency of `RTOp`). See an example, see Explicit enable of a package and its tests.

20) **If no SE packages are enabled, nothing will get built**: Most TriBITS projects are set up such that if the user does not explicitly enable at least one SE package in some way, then nothing will be enabled or built. In this case, when `${PROJECT_NAME}_ALLOW_NO_PACKAGES=TRUE` a warning will be printed and configuration will complete. However, if `${PROJECT_NAME}_ALLOW_NO_PACKAGES=FALSE`, then the configure will die with an error message. For example, the `checkin-test.py` script sets `${PROJECT_NAME}_ALLOW_NO_PACKAGES=OFF` to make sure that something gets enabled and tested in order to accept the results of the test and allow a push. For an example, see Default configure with no packages enabled on input.

21) **TriBITS prints all enables and disables to STDOUT**: TriBITS prints out (to `cmake` STDOUT) the initial set of enables/disables on input, prints a line whenever it sets (or overrides) an enable or disable, and prints out the final set of enables/disables. Therefore, the user just needs to grep the `cmake` STDOUT to find out why any particular SE package or TPL is enabled or disabled in the end. In addtion, will print out when tests/examples for a given SE package gets enabled and when support for optional SE packages and TPLs is enabled or not. A detailed description of this output is given in all of the below examples but in particular see Explicit enable of a package and its tests.

22) **TriBITS auto-enables/disables done using non-cache local variables**: TriBITS setting (or overrides) of enable/disable cache variables are done by setting local non-cache variables at the top project-level scope. This is done so they don't get set in the cache and so that the same dependency enable/disable logic is redone, from scratch, with each re-configure which results in the same enable/disable logic output as for the initial configure. This is to avoid confusion by the user about why some SE packages and TPLs are enabled and some are not on subsequent reconfigures.

TriBITS prints out a lot of information about the enable/disable logic as it applies the above rules/behaviors. For a large TriBITS project with lots of packages, this can produce a lot of output to STDOUT. One just needs to understand what TriBITS is printing out and where to look in the output for different information. The example Example Enable/Disable

Use Cases given below show what this output looks like for the various enable/disable scenarios and tries to explains in more detail the reasons for why the given behavior is implemented the way that it is. Given this output, the rule definitions given above, and the detailed example Example Enable/Disable Use Cases described below, one should always be able to figure out exactly why the final set of enables/disables is the way it is, even in the largest and most complex of TriBITS projects. (NOTE: The same can *not* be said for many other large software configuration and depolyment systems where basic decisions about what to enable and disable are hidden from the user and can be very difficult to debug).

The above behaviors address the majority of the TriBITS dependency management system. However, when dealing with TriBITS projects with multiple repositories, some other behaviors are supported through the defintion of a few more variables. The following TriBITS repository-related variables alter what packages in a given TriBITS repository get enabled implicitly or not:

`${REPOSITORY_NAME}_NO_IMPLICIT_PACKAGE_ENABLE`

> If set to `ON`, then the packages in Repository `${REPOSITORY_NAME}` will not be implicitly enabled in any of the package adjustment logic.

`${REPOSITORY_NAME}_NO_IMPLICIT_PACKAGE_ENABLE_EXCEPT`

> List of packages in the Repository `${REPOSITORY_NAME}` that will be allowed to be implicitly enabled. Only checked if `${REPOSITORY_NAME}_NO_IMPLICIT_PACKAGE_ENABLE` is true.

The above variables typically are defined in the outer TriBITS Project's CTest driver scripts or even in top-level project files in order to adjust how its listed repositories are handled. What these variable do is to allow a large project to turn off the enable of optional SE packages in a given TriBITS repository to provide more detailed control of what gets used from a given TriBITS repository. This, for example, is used in the CASL VERA project to mange some of its extra repositories and packages to further auto-reduce the number of qpckages that get enabled.

5.3 Example Enable/Disable Use Cases

Below, a few of the standard enable/disable use cases for a TriBITS project are given using the Example ReducedMockTrilinos Project Dependency Structure that demonstrate the TriBITS Dependency Handling Behaviors.

The use cases covered are:

- Default configure with no packages enabled on input
- Explicit enable of a package and its tests
- Explicit enable of a package, its tests, an optional TPL, with ST enabled
- Explicit disable of a package
- Conflicting explicit disable of a package
- Explicit enable of an optional TPL
- Explicit disable of an optional TPL
- Explicit disable of a required TPL
- Explicit enable of a subpackage
- Explicit enable of an optional package dependency
- Explicit disable of an optional package dependency
- Explicit enable of an optional TPL dependency
- Explicit disable of an optional TPL dependency
- Explicit enable of a package and downstream packages and tests
- Enable all packages

Below are the detailed use cases with example TriBITS output. All of these use cases and more can be easily run from the command-line by first setting:

```
$ export
REDUCED_MOCK_TRILINOS=<base-dir>/tribits/package_arch/UnitTests/ReducedMockTrilinos
```

and then copy and pasting the `cmake` commands shown below. Just make sure to run these in a temp directory because this actually configures a CMake project in the local directory. Just make sure and run:

```
$ rm -r CMake*
```

before each run to clear the CMake cache.

Default configure with no packages enabled on input

The first use-case to consider is the configure of a TriBITS project without enabling any packages by any means. For the `ReducedMockTrilinos` project, this is done with:

```
$ cmake ${REDUCED_MOCK_TRILINOS}
```

produces the relevant dependency-related output:

```
Explicitly enabled packages on input (by user):  0
Explicitly enabled SE packages on input (by user):  0
Explicitly disabled packages on input (by user or by default):  0
Explicitly disabled SE packages on input (by user or by default):  ThyraCrazyStuff 1
Explicitly enabled TPLs on input (by user):  0
Explicitly disabled TPLs on input (by user or by default):  0

...

Final set of enabled packages:  0
Final set of enabled SE packages:  0
Final set of non-enabled packages:  Teuchos RTOp Epetra Triutils EpetraExt Thyra 6
Final set of non-enabled SE packages:  Teuchos RTOp Epetra Triutils EpetraExt \
 ThyraCoreLibs ThyraGoodStuff ThyraCrazyStuff ThyraEpetra ThyraEpetraExt Thyra 11
Final set of enabled TPLs:  0
Final set of non-enabled TPLs:  MPI BLAS LAPACK Boost UMFPACK AMD PETSC 7

...

**
** WARNING:  There were no packages configured so no libraries or tests/examples will
**
```

The above example demonstrates the following behaviors of the TriBITS dependency handling system:

- PT/ST SE packages given default unset enable/disable state
- **EX SE packages disabled by default (i.e., the EX SE package** `ThyraCrazyStuff` is set to `OFF` by default at the very beginning).
- PT/ST TPLs given default unset enable/disable state
- EX TPLs given default unset enable/disable state
- If no SE packages are enabled, nothing will get built

Explicit enable of a package and its tests

One of the most typical use cases is for the user to explicity enable one or more top-level TriBITS package and enabled its tests. This configuration would be used to drive local development on a specific set of packages.

Consider the configure of the `ReducedMockTrilinos` project enabling the top-level `Thyra` package and its tests with:

```
$ cmake -DTrilinos_ENABLE_Thyra:BOOL=ON \
   -DTrilinos_ENABLE_TESTS:BOOL=ON \
   ${REDUCED_MOCK_TRILINOS}
```

which produces the relevant dependency-related output:

```
Explicitly enabled packages on input (by user):  Thyra 1
Explicitly enabled SE packages on input (by user):  Thyra 1
Explicitly disabled packages on input (by user or by default):  0
Explicitly disabled SE packages on input (by user or by default):  ThyraCrazyStuff 1
Explicitly enabled TPLs on input (by user):  0
Explicitly disabled TPLs on input (by user or by default):  0

Enabling subpackages for hard enables of parent packages due to \
 Trilinos_ENABLE_<PARENT_PACKAGE>=ON ...

-- Setting subpackage enable Trilinos_ENABLE_ThyraCoreLibs=ON because parent \
 package Trilinos_ENABLE_Thyra=ON
-- Setting subpackage enable Trilinos_ENABLE_ThyraEpetra=ON because parent package \
 Trilinos_ENABLE_Thyra=ON

Disabling forward required SE packages and optional intra-package support that have a
 dependancy on disabled SE packages Trilinos_ENABLE_<TRIBITS_PACKAGE>=OFF ...

-- Setting Thyra_ENABLE_ThyraCrazyStuff=OFF because Thyra has an optional library \
 dependence on disabled package ThyraCrazyStuff

Enabling all tests and/or examples that have not been explicitly disabled because \
 Trilinos_ENABLE_[TESTS,EXAMPLES]=ON ...

-- Setting ThyraCoreLibs_ENABLE_TESTS=ON
-- Setting ThyraCoreLibs_ENABLE_EXAMPLES=ON
-- Setting ThyraEpetra_ENABLE_TESTS=ON
-- Setting ThyraEpetra_ENABLE_EXAMPLES=ON
-- Setting Thyra_ENABLE_TESTS=ON
-- Setting Thyra_ENABLE_EXAMPLES=ON

Enabling all required (and optional since Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES=ON) \
 upstream SE packages for current set of enabled packages ...

-- WARNING: Not Setting Trilinos_ENABLE_ThyraGoodStuff=ON even though Thyra \
 has an optional dependence on ThyraGoodStuff because Trilinos_ENABLE_SECONDARY_TESTED
-- WARNING: Not Setting Trilinos_ENABLE_ThyraEpetraExt=ON even though Thyra \
 has an optional dependence on ThyraEpetraExt because Trilinos_ENABLE_SECONDARY_TESTED
-- Setting Trilinos_ENABLE_Epetra=ON because ThyraEpetra has a required dependence on
-- Setting Trilinos_ENABLE_Teuchos=ON because ThyraCoreLibs has a required dependence
-- Setting Trilinos_ENABLE_RTOp=ON because ThyraCoreLibs has a required dependence on

Enabling all optional intra-package enables <TRIBITS_PACKAGE>_ENABLE_<DEPPACKAGE> that
 not currently disabled if both sets of packages are enabled ...

-- Setting Thyra_ENABLE_ThyraEpetra=ON since Trilinos_ENABLE_Thyra=ON AND Trilinos_ENA

Enabling all remaining required TPLs for current set of enabled packages ...

-- Setting TPL_ENABLE_BLAS=ON because it is required by the enabled package Teuchos
-- Setting TPL_ENABLE_LAPACK=ON because it is required by the enabled package Teuchos

Final set of enabled packages:  Teuchos RTOp Epetra Thyra 4
Final set of enabled SE packages:  Teuchos RTOp Epetra ThyraCoreLibs ThyraEpetra Thyra
Final set of non-enabled packages:  Triutils EpetraExt 2
```

```
Final set of non-enabled SE packages:  Triutils EpetraExt ThyraGoodStuff ThyraCrazyStu
Final set of enabled TPLs:  BLAS LAPACK 2
Final set of non-enabled TPLs:  MPI Boost UMFPACK AMD PETSC 5

Getting information for all enabled TPLs ...

-- Processing enabled TPL: BLAS
-- Processing enabled TPL: LAPACK

Configuring individual enabled Trilinos packages ...

Processing enabled package: Teuchos (Libs)
Processing enabled package: RTOp (Libs)
Processing enabled package: Epetra (Libs)
Processing enabled package: Thyra (CoreLibs, Epetra, Tests, Examples)
```

This is a configuration that a developer would use to develop on the `Thyra` package and its subpackages for example. There is no need to be enabling the tests and examples for upstream packages unless those packages are going to be changed as well.

This case demonstrates a number of TriBITS dependency handling behaviors that are worth some discussion.

First, note that enabling the parent package `Thyra` with `Trilinos_ENABLE_Thyra=ON` right away results in the auto-enable of its `PT` subpackages `ThyraCoreLibs` and `ThyraEpetra` which demonstrates the behvior Enable/disable of parent package is enable/disable for subpackages. Note that the `ST` subpackages `ThyraGoodStuff` and `ThyraEpetraExt` where *not* enabled because `${PROJECT_NAME}_SECONDARY_TESTED_CODE=OFF` (which is off by default) which demonstrates the behavior ST SE packages only auto-enabled if ST code is enabled.

Second, note the auto-enable of required upstream SE packages `Epetra`, `RTOp` and `Teuchos` shown in lines like:

```
-- Setting Trilinos_ENABLE_Teuchos=ON because ThyraCoreLibs has a required dependence
```

Lastly, note that the final set of enabled pakcages, SE packages, tests/examples and TPLs can be clearly seen when processing the TPLs and top-level packages in the lines:

```
Getting information for all enabled TPLs ...

-- Processing enabled TPL: BLAS
-- Processing enabled TPL: LAPACK

Configuring individual enabled Trilinos packages ...

Processing enabled package: Teuchos (Libs)
Processing enabled package: RTOp (Libs)
Processing enabled package: Epetra (Libs)
Processing enabled package: Thyra (CoreLibs, Epetra, Tests, Examples)
```

Note that subpackage enables are listed with their parent packages along with if the tests and/or exmaples are enabled. Top-level level packages that don't have subpackages just show if the `Libs` or also the `Tests` and `Examples` have been enabled as well.

Explicit enable of a package, its tests, an optional TPL, with ST enabled

An extended use case shown here is for the explicit enable of a package and its tests along with the enable of an optional TPL and with `ST` code enabled. This is a configuration that would be used to support the local development of a TriBITS package that involves modifying `ST` software.

Consider the configure of the `ReducedMockTrilinos` project with:

```
$ cmake -DTPL_ENABLE_Boost:BOOL=ON \
   -DTrilinos_ENABLE_Thyra:BOOL=ON \
   -DTrilinos_ENABLE_TESTS:BOOL=ON \
```

```
  -DTrilinos_ENABLE_SECONDARY_TESTED_CODE:BOOL=ON \
  -DTrilinos_ENABLE_TESTS:BOOL=ON \
  ${REDUCED_MOCK_TRILINOS}
```

which produces the relevant dependency-related output:

```
Explicitly enabled packages on input (by user):  Thyra 1
Explicitly enabled SE packages on input (by user):  Thyra 1
Explicitly disabled packages on input (by user or by default):  0
Explicitly disabled SE packages on input (by user or by default):  ThyraCrazyStuff 1
Explicitly enabled TPLs on input (by user):  Boost 1
Explicitly disabled TPLs on input (by user or by default):  0

Enabling subpackages for hard enables of parent packages due to \
 Trilinos_ENABLE_<PARENT_PACKAGE>=ON ...

-- Setting subpackage enable Trilinos_ENABLE_ThyraCoreLibs=ON because parent package \
 Trilinos_ENABLE_Thyra=ON
-- Setting subpackage enable Trilinos_ENABLE_ThyraGoodStuff=ON because parent package
 Trilinos_ENABLE_Thyra=ON
-- Setting subpackage enable Trilinos_ENABLE_ThyraEpetra=ON because parent package \
 Trilinos_ENABLE_Thyra=ON
-- Setting subpackage enable Trilinos_ENABLE_ThyraEpetraExt=ON because parent package
 Trilinos_ENABLE_Thyra=ON

Disabling forward required SE packages and optional intra-package support that have a
 dependancy on disabled SE packages Trilinos_ENABLE_<TRIBITS_PACKAGE>=OFF ...

-- Setting Thyra_ENABLE_ThyraCrazyStuff=OFF because Thyra has an optional library \
 dependence on disabled package ThyraCrazyStuff

Enabling all tests and/or examples that have not been explicitly disabled because \
 Trilinos_ENABLE_[TESTS,EXAMPLES]=ON ...

-- Setting ThyraCoreLibs_ENABLE_TESTS=ON
-- Setting ThyraGoodStuff_ENABLE_TESTS=ON
-- Setting ThyraEpetra_ENABLE_TESTS=ON
-- Setting ThyraEpetraExt_ENABLE_TESTS=ON
-- Setting Thyra_ENABLE_TESTS=ON

Enabling all required (and optional since Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES=ON) \
 upstream SE packages for current set of enabled packages ...

-- Setting Trilinos_ENABLE_EpetraExt=ON because ThyraEpetraExt has a required dependen
-- Setting Trilinos_ENABLE_Epetra=ON because ThyraEpetra has a required dependence on
-- Setting Trilinos_ENABLE_Teuchos=ON because ThyraCoreLibs has a required dependence
-- Setting Trilinos_ENABLE_RTOp=ON because ThyraCoreLibs has a required dependence on
-- Setting Trilinos_ENABLE_Triutils=ON because EpetraExt has an optional dependence on

Enabling all optional intra-package enables <TRIBITS_PACKAGE>_ENABLE_<DEPPACKAGE> that
 not currently disabled if both sets of packages are enabled ...

-- Setting EpetraExt_ENABLE_Triutils=ON since Trilinos_ENABLE_EpetraExt=ON \
 AND Trilinos_ENABLE_Triutils=ON
-- Setting Thyra_ENABLE_ThyraGoodStuff=ON since Trilinos_ENABLE_Thyra=ON \
 AND Trilinos_ENABLE_ThyraGoodStuff=ON
-- Setting Thyra_ENABLE_ThyraEpetra=ON since Trilinos_ENABLE_Thyra=ON \
 AND Trilinos_ENABLE_ThyraEpetra=ON
-- Setting Thyra_ENABLE_ThyraEpetraExt=ON since Trilinos_ENABLE_Thyra=ON \
 AND Trilinos_ENABLE_ThyraEpetraExt=ON
```

```
Enabling all remaining required TPLs for current set of enabled packages ...

-- Setting TPL_ENABLE_BLAS=ON because it is required by the enabled package Teuchos
-- Setting TPL_ENABLE_LAPACK=ON because it is required by the enabled package Teuchos

Enabling all optional package TPL support for currently enabled TPLs ...

-- Setting Teuchos_ENABLE_Boost=ON since TPL_ENABLE_Boost=ON

Final set of enabled packages:  Teuchos RTOp Epetra Triutils EpetraExt Thyra 6
Final set of enabled SE packages:  Teuchos RTOp Epetra Triutils EpetraExt ThyraCoreLib
 ThyraGoodStuff ThyraEpetra ThyraEpetraExt Thyra 10
Final set of non-enabled packages:  0
Final set of non-enabled SE packages:  ThyraCrazyStuff 1
Final set of enabled TPLs:  BLAS LAPACK Boost 3
Final set of non-enabled TPLs:  MPI UMFPACK AMD PETSC 4

Getting information for all enabled TPLs ...

-- Processing enabled TPL: BLAS
-- Processing enabled TPL: LAPACK
-- Processing enabled TPL: Boost

Configuring individual enabled Trilinos packages ...

Processing enabled package: Teuchos (Libs)
Processing enabled package: RTOp (Libs)
Processing enabled package: Epetra (Libs)
Processing enabled package: Triutils (Libs)
Processing enabled package: EpetraExt (Libs)
Processing enabled package: Thyra (CoreLibs, GoodStuff, Epetra, EpetraExt, Tests, Exam
```

There are a few more behaviors of the TriBITS system this particular configuration shows.

First, note the enable of the `ST Thyra` subpackages in lines like:

```
-- Setting subpackage enable Trilinos_ENABLE_ThyraGoodStuff=ON because parent package
 Trilinos_ENABLE_Thyra=ON
```

Second, note the auto-enable of support for optional SE packages in lines like:

```
-- Setting EpetraExt_ENABLE_Triutils=ON since Trilinos_ENABLE_EpetraExt=ON \
 AND Trilinos_ENABLE_Triutils=ON
```

Third, note the auto-enable of support for the optional TPL `Boost` in the line:

```
-- Setting Teuchos_ENABLE_Boost=ON since TPL_ENABLE_Boost=ON
```

Explicit disable of a package

Another common use case is to enable a package but to disable an optional upstream package. This type of configuration would be used a part of a "black list" appraoch to enabling only a subset of packages and optional support. The "black list" approach is to enable a package with `${PROJECT_NAME}_ENABLE_ALL_OPTIONAL_PACKAGES=ON` (the TriBITS default) but then to turn off a specific set of packages that you don't want. This is contrasted with a "white list" approach where you would configure with `${PROJECT_NAME}_ENABLE_ALL_OPTIONAL_PACKAGES=OFF` and then have to manually enable all of the optional packages you want.

Consider the configure of the `ReducedMockTrilinos` project enabling `Thyra` but disabling `Epetra` with:

```
$ cmake  -DTrilinos_ENABLE_Thyra:BOOL=ON \
   -DTrilinos_ENABLE_Epetra:BOOL=OFF \
   -DTrilinos_ENABLE_TESTS:BOOL=ON \
   ${REDUCED_MOCK_TRILINOS}
```

which produces the relevant dependency-related output:

```
Disabling forward required SE packages and optional intra-package support that have a
 dependancy on disabled SE packages Trilinos_ENABLE_<TRIBITS_PACKAGE>=OFF ...

-- Setting Trilinos_ENABLE_Triutils=OFF because Triutils has a required library depend
 on disabled package Epetra
-- Setting Trilinos_ENABLE_EpetraExt=OFF because EpetraExt has a required library \
 disabled package Epetra
-- Setting Trilinos_ENABLE_ThyraEpetra=OFF because ThyraEpetra has a required library
 disabled package Epetra
-- Setting Trilinos_ENABLE_ThyraEpetraExt=OFF because ThyraEpetraExt has a required li
 disabled package EpetraExt
-- Setting Thyra_ENABLE_ThyraCrazyStuff=OFF because Thyra has an optional library \
 disabled package ThyraCrazyStuff
-- Setting Thyra_ENABLE_ThyraEpetra=OFF because Thyra has an optional library \
 disabled package ThyraEpetra
-- Setting Thyra_ENABLE_ThyraEpetraExt=OFF because Thyra has an optional library \
 disabled package ThyraEpetraExt

Final set of enabled packages:  Teuchos RTOp Thyra 3
Final set of enabled SE packages:  Teuchos RTOp ThyraCoreLibs Thyra 4
Final set of non-enabled packages:  Epetra Triutils EpetraExt 3
Final set of non-enabled SE packages:  Epetra Triutils EpetraExt ThyraGoodStuff \
 ThyraCrazyStuff ThyraEpetra ThyraEpetraExt 7
Final set of enabled TPLs:  BLAS LAPACK 2
Final set of non-enabled TPLs:  MPI Boost UMFPACK AMD PETSC 5

Configuring individual enabled Trilinos packages ...

Processing enabled package: Teuchos (Libs)
Processing enabled package: RTOp (Libs)
Processing enabled package: Thyra (CoreLibs, Tests, Examples)
```

Note how the disable of `Epetra` wipes out all of the required and optional SE packages that depend on `Epetra`. What is left is only the `ThyracoreLibs` and its upstream dependencies that don't depend on `Epetra` (which is only `RTOp` and `Teuchos`).

Conflicting explicit disable of a package

One use case that occasionally comes up is when a set of inconsistent enables and disables are set. While this seems illogical that anyone would ever do this, when it comes to larger more complex projects with lots of packages and lots of dependencies, this can happen very easily. In some cases, someone is enabling a set of packages they want and is trying to weed out as some of (what they think) are optional dependencies they don't need and accidentally disables a package that is an indirect required dependency of one of the packages they want. The other use case where conflicting enables/disables can occur is in CTest drivers using TRIBITS_CTEST_DRIVER() where an upstream package has failed and is explicitly disabled. TriBITS can either be set up to have the disable override the explicit enable or stop the configure and depending on the value of the cache variable `${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES` (see Disables trump enables where there is a conflict).

For example, consider what happens with the `ReducedMockTrilinos` project (see Example ReducedMockTrilinos Project Dependency Structure) if someone tries to enable the `RTOp` package and disable the `Teuchos` package. This is not consistent because `RTOp` has a required dependency on `Teuchos`. The default behavior of TriBITS is this case is shown in the below configure:

```
$ cmake -DTrilinos_ENABLE_Epetra:BOOL=ON \
   -DTrilinos_ENABLE_RTOp:BOOL=ON \
   -DTrilinos_ENABLE_Teuchos:BOOL=OFF \
   ${REDUCED_MOCK_TRILINOS}
```

which produces the relevant dependency-related output:

```
Explicitly enabled packages on input (by user):  RTOp Epetra 2
Explicitly disabled packages on input (by user or by default):  Teuchos 1

Disabling forward required SE packages and optional intra-package support that have \
 a dependancy on disabled SE packages Trilinos_ENABLE_<TRIBITS_PACKAGE>=OFF ...

   ***
   *** ERROR: Setting Trilinos_ENABLE_RTOp=OFF which was 'ON' because RTOp has a requi
 library dependence on disabled package Teuchos!
   ***
```

As shown above, the TriBITS default is to set `${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES=OFF` which results in a configure-time error with a good error message.

However, if one sets `${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES=ON` and configures with:

```
$ cmake -DTrilinos_ENABLE_Epetra:BOOL=ON \
   -DTrilinos_ENABLE_RTOp:BOOL=ON \
   -DTrilinos_ENABLE_Teuchos:BOOL=OFF \
   -DTrilinos_DISABLE_ENABLED_FORWARD_DEP_PACKAGES:BOOL=ON \
   ${REDUCED_MOCK_TRILINOS}
```

then the disable trumps the enable and results in a successful configure as shown in the relevant dependency-related output:

```
Explicitly enabled packages on input (by user):  RTOp Epetra 2
Explicitly disabled packages on input (by user or by default):  Teuchos 1

Disabling forward required SE packages and optional intra-package support that \
 have a dependancy on disabled SE packages Trilinos_ENABLE_<TRIBITS_PACKAGE>=OFF ...

 ***
 *** WARNING: Setting Trilinos_ENABLE_RTOp=OFF which was 'ON' because RTOp has \
 a required library dependence on disabled package Teuchos but \
 Trilinos_DISABLE_ENABLED_FORWARD_DEP_PACKAGES=ON!
 ***

Final set of enabled packages:  Epetra 1
Final set of non-enabled packages:  Teuchos RTOp Triutils EpetraExt Thyra 5
```

As shown, what you end of with is just the enabled package `Epetra` which does not have a required dependency on the disabled package `Teuchos`. Developers of large complex TriBITS projects would be wise to set the default for ${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES to `ON`.

Explicit enable of an optional TPL:

ToDo: Set Trilinos_ENABLE_Thyra=ON and TPL_ENABLE_MPI=ON

Explicit disable of an optional TPL:

ToDo: Set Trilinos_ENABLE_Thyra=ON and TPL_ENABLE_MPI=OFF

Explicit disable of a required TPL

ToDo: Set Trilinos_ENABLE_Epetra=ON and Trilinos_ENABLE_BLAS=OFF

Explicit enable of a subpackage

ToDo: Enable ThyraEpetra and show how it enables other SE packages and at the end, enables the Thyra package (just for show).

Explicit enable of an optional package dependency

ToDo: Set Trilinos_ENABLE_EpetraExt=ON and EpetraExt_ENABLE_Triutils=ON and shows how it enables Trilinos_ENABLE_Triutils=ON even through ST code is not enabled.

Explicit disable of an optional package dependency

ToDo: Set Trilinos_ENABLE_EpetraExt=ON, Trilinos_ENABLE_Triutils=ON, and EpetraExt_ENABLE_Triutils=OFF. Discuss how EpetraExt's and ThyraEpetraExt's CMakeLists.txt files might turn off some features if they detects that EpetraExt/Triutils support is turned off.

Explicit enable of an optional TPL dependency

ToDo: The current ReducedMockTrilinos is not set up to give a good example of this. We should add an optional Boost dependency to say, Epetra. Then we could show the enable of Teuchos and Epetra and Epetra_ENABLE_Boost=ON. That would enable Boost and enable support for Boost in Epetra but would not provide support for Boost in Teuchos.

Explicit disable of an optional TPL dependency

ToDo: The current ReducedMockTrilinos is not set up to give a good example of this. We should add an optional Boost dependency to say, Epetra. Then we could show the enable of Teuchos and Epetra and TPL_ENABLE_Boost=ON but set Epetra_ENABLE_Boost=OFF. That would provide support for Boost in Teuchos but not in Epetra.

Explicit enable of a package and downstream packages and tests

ToDo: Set Trilinos_ENABLE_RTOp=ON, Trilinos_ENABLE_ALL_FORWARD_DEP_PACKAGES=ON, and Trilinos_ENABLE_TESTS=ON and show what packages and tests/examples get enabled. This is the use case for the checkin-test.py script for PT enabled code.

Enable all packages

The last use case to consider is enabling all defined packages. This configuration would be used for either doing a full test of all of the packages defined or to creating a distrubtion of the project.

Enabling all packages the with:

```
$ cmake -DTrilinos_ENABLE_ALL_PACKAGES:BOOL=ON \
   -DTrilinos_DUMP_PACKAGE_DEPENDENCIES:BOOL=ON \
   ${REDUCED_MOCK_TRILINOS}
```

produces the relevant dependency-related output:

```
Enabling all SE packages that are not currently disabled because of Trilinos_ENABLE_AL

-- Setting Trilinos_ENABLE_Teuchos=ON
-- Setting Trilinos_ENABLE_RTOp=ON
-- Setting Trilinos_ENABLE_Epetra=ON
-- Setting Trilinos_ENABLE_ThyraCoreLibs=ON
-- Setting Trilinos_ENABLE_ThyraEpetra=ON
-- Setting Trilinos_ENABLE_Thyra=ON

Enabling all remaining required TPLs for current set of enabled packages ...

-- Setting TPL_ENABLE_BLAS=ON because it is required by the enabled package Teuchos
-- Setting TPL_ENABLE_LAPACK=ON because it is required by the enabled package Teuchos

Final set of enabled packages:  Teuchos RTOp Epetra Thyra 4
Final set of enabled SE packages:  Teuchos RTOp Epetra ThyraCoreLibs ThyraEpetra Thyra
Final set of non-enabled packages:  Triutils EpetraExt 2
Final set of non-enabled SE packages:  Triutils EpetraExt ThyraGoodStuff ThyraCrazyStu
Final set of enabled TPLs:  BLAS LAPACK 2
```

```
Final set of non-enabled TPLs:  MPI Boost UMFPACK AMD PETSC 5

Getting information for all enabled TPLs ...

-- Processing enabled TPL: BLAS
-- Processing enabled TPL: LAPACK

Configuring individual enabled Trilinos packages ...

Processing enabled package: Teuchos (Libs)
Processing enabled package: RTOp (Libs)
Processing enabled package: Epetra (Libs)
Processing enabled package: Thyra (CoreLibs, Epetra)
```

As shown above,

5.4 <Project>PackageDependencies.xml

The TriBITS CMake configure system can write out the project's package dependenices into a file `<Project>Dependnecies.xml`. This file is used by a number of the SE tools. The structure of this file, showing one of the more interesting mock packages from the MockTrilinos project is shown below:

```
<PackageDependencies project="Trilinos">

  <Package name="Amesos" dir="packages/amesos" type="PT">
    <LIB_REQUIRED_DEP_PACKAGES value="Teuchos,Epetra"/>
    <LIB_OPTIONAL_DEP_PACKAGES value="EpetraExt"/>
    <TEST_REQUIRED_DEP_PACKAGES/>
    <TEST_OPTIONAL_DEP_PACKAGES value="Triutils,Galeri"/>
    <LIB_REQUIRED_DEP_TPLS/>
    <LIB_OPTIONAL_DEP_TPLS value="SuperLUDist,ParMETIS,UMFPACK,SuperLU,MUMPS"/>
    <TEST_REQUIRED_DEP_TPLS/>
    <TEST_OPTIONAL_DEP_TPLS/>
    <EmailAddresses>
      <Regression address="amesos-regression@repo.site.gov"/>
    </EmailAddresses>
    <ParentPackage value=""/>
  </Package>

</PackageDependencies>
```

This XML file contains the names, directories, and testing groups (`type`), the CDash email address and all of the SE package and TPL dependencies. There are several python tools under `tribits/python/` that read in this file and use the created data-structure for various tasks. A TriBITS project configure can create this file as a byproduct (see ???), or the CMake -P script `TribitsDumpDepsXmlScript.cmake` can be used to create this file on the fly without having to configure a TriBITS project.

6 TriBITS Automated Testing

Much of the value provided by the TriBITS system is related to the support of testing of a complex project. Many different types of testing are required in a complex project and development effort. In addition a large project with lots of repositories and packages provides a number of testing and development challenges but also provides a number of opportunities to do testing in an efficient way; especially pre-push and post-push continuous integration (CI) testing. In addition, a number of post-push automated nightly test cases must be managed. TriBITS takes full advantage of the features of raw CMake, CTest, and CDash in support of testing and where gaps exist, TriBITS provides tools and customizations.

The following subsections describe several aspects to the TriBITS support for testing.

6.1 Test Classifications for Repositories, Packages, and Tests

TriBITS defines a few different testing-related classifications for a TriBITS project. These different classifications are used to select subsets of the project's repositories, packages (and code within these packages), and tests to be included in a given project build and test definition. These different classification are:

- Repository Test Classification
- SE Package Test Group
- Test Test Category

These different test-related classifications are used to defined several different Nested Layers of TriBITS Project Testing. First, the Repository Test Classification determines what SE packages are defined to even consider being enabled. Second, if a repository is selected, then the SE Package Test Group determines what SE packages (and optional code in those packages) are even enabled such that their <packageDir>/CMakeLists.txt files are even processed. Lastly, if an SE package gets enabled, then the Test Test Category determines what test exectuables and test cases get defined using the functions TRIBITS_ADD_EXECUTABLE(), TRIBITS_ADD_TEST() and TRIBITS_ADD_ADVANCED_TEST().

More detailed descriptions of Repository Test Classifications , SE Package Test Groups, and Test Test Categories are given in the following subsections.

Repository Test Classification

The first type of test-related classification is for extra repositories defined in the file <projectDir>/cmake/ExtraRepositoriesList.cmake using the `REPO_CLASSIFICATION` field in the macro call TRIBITS_PROJECT_DEFINE_EXTRA_REPOSITORIES(). These classifications map to the standard CTest dashboard types `Continuous`, `Nightly`, and `Experimental` (see CTest documentation and TriBITS Package-by-Package CTest/Dash Driver for details).

- Repositories marked **Continuous** match the standard CTest dashboard type `Continuous`. These repositories are pulled in when `${PROJECT_NAME}_ENABLE_KNOWN_EXTERNAL_REPOS_TYPE=Continuous`, or `Nightly`. Repositories marked as `Continuous` are cloned, updated, and processed by default in all project automated testing described in Nested Layers of TriBITS Project Testing. NOTE: One should not confuse this with the Test Test Category CONTINUOUS.

- Repositories marked **Nightly** match the standard CTest dashboard type `Nightly`. These repositories are pulled in by default when `${PROJECT_NAME}_ENABLE_KNOWN_EXTERNAL_REPOS_TYPE=Nightly`. Repositories marked as `Nightly` are not processed by default as part of either Pre-Push CI Testing or Post-Push CI Testing. One would mark a repository as `Nightly` for few reasons. First, an extra repo may be marked as `Nightly` if it may not be available to all developers to clone and only the the nightly testing processes and machines may have access. Also, an extra repo may also be marked as `Nightly` if it does not contain any packages that the project wants to pay the cost to include in even Post-Push CI Testing. NOTE: One should not confuse this with the Test Test Category NIGHTLY.

- Repositories marked **Experimental** match the standard CTest dashboard type `Experimental`. Repositories marked as `Experimental` are not processed by default as part on any automated testing described in Nested Layers of TriBITS Project Testing (except for perhaps some experimental builds). The main reason that an extra repo may be marked as `Experimental` is that it may only contain `EX` SE packages and therefore none of these packages would be enabled by default anyway. Also, a repo may be marked as `Experimental` if it is developed in a very sloppy way such that one cannot even assume that the repository's <repoDir>/PackagesList.cmake, <repoDir>/TPlsList.cmake, and <packageDir>/cmake/Dependencies.cmake files are not without errors. Since these files are always processed if the repository is included, then any errors in these files will cause the entire configure to fail!

SE Package Test Group

Once a set of TriBITS repositories are selected in accordance with their Repository Test Classification, that determines the set of SE packages and TPLs defined for the TriBITS project. Given the set of defined SE packages and TPLs, the set of SE packages that get enabled is determined by the **SE Package Test Group** which is defined and described here.

Every TriBITS SE Package and TriBITS TPL is assigned a test group. These test groups are for *Primary Tested* (PT) code, *Secondary Tested* (ST) code, and *Experimental* (EX) code. The test group defines *what* SE packages and TPL get selected (or are excluded from being selected) to include in a given build for testing-related purposes. SE packages may

also conditionally build in additional code based on the testing group. The detailed rules for when an SE package or TPL are selected or excluded from the build based on the test group is given in TriBITS Dependency Handling Behaviors. We only summarize those rules here.

More detailed descriptions of the test groups are given below.

- **Primary Tested (PT)** Code is of the highest priority to keep working for the current development effort. SE packages and TPLs are selected to be `PT` for one of a number of reasons. First, if the capability provided by the code is mature and if a regression would cause major harm to a customer, the code should likely be marked as `PT`. Also, if the build and correct functioning of the code is needed by other development team members to support their day-to-day development activities, then the code should be marked as `PT` as well. A TPL, on the other hand, is marked as `PT` if it is required by a `PT` SE package. Every project developer is expected to have every `PT` TPL installed on every machine where they do development on and from which they push to the global repo (see checkin-test.py script). `PT` SE packages and TPLs are the the foundation for Pre-Push CI Testing.

- **Secondary Tested (PT)** Code is still very important code for the project and represents important capability to maintain but is excluded from the `PT` set of code for one of a couple of possible reasons. First, code may be marked as `ST` if is not critical to drive most day-to-day development activities. If `ST` code breaks, it usually will not cause immediate and major harm to most developers. Also, code may be marked as `ST` if it has required dependencies on `ST` TPLs which are either hard to install or may not be available on all platforms where developers do their development and where they push changes to the global repo . In addition, code my be marked as `ST` if the project is just too big and developers can't be expected to build and test all of this code with every push. `ST` code can be included in the TriBITS auto-enable algorithms by setting the variable `${PROJECT_NAME}_SECONDARY_TESTED_CODE=ON` (see TriBITS Dependency Handling Behaviors). Otherwise, `ST` code is not enabled by auto-enable algorithms. Typically, `ST` code is excluded from the default builds in Pre-Push CI Testing but *ST*' code is typically tested in Post-Push CI Testing, Nightly Testing as well as in other non-CI testing cases.

- **Experimental (EX)** Code is usually too unstable, buggy, or non-portable to be maintained as part of the projects automated testing processes. Or the code just may not be important enough to the project to bother paying the cost to test it in the project's automated testing processes. The ability to mark some code as `EX` allows the developers of that code to include their code in the VC repos with the rest of the project's code and be able to take advantage of all of the development tools provided by TriBITS while not having to "sign up" for all of the responsibilities of maintaining working software that every developer has to take a part in helping to keep working.

The test group for each type of entity is assigned in the following places:

- The top-level TriBITS Package's test group is assigned using the `CLASSIFICATION` field in the the macro call TRIBITS_DEFINE_REPOSITORY_PACKAGES() in its parent repository's <repoDir>/PackagesList.cmake file.
- A TriBITS Subpackage's test group is assigned using the `CLASSIFICATIONS` field of the `SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS` argument in the macro call TRIBITS_DEFINE_PACKAGE_DEPENDENCIES() in its parent's <packageDir>/cmake/Dependencies.cmake file.
- A TriBITS TPLs's test group is assigned using the `CLASSIFICATION` field in the macro call TRIBITS_DEFINE_REPOSITORY_TPLS() in its parent repository's <repoDir>/TPLsList.cmake file.

After these files are processed, the variable ${PACKAGE_NAME}_TESTGROUP gives the test group for each defined SE Package while the variable `${TPL_NAME}_TESTGROUP` gives the test group for each defined TPL.

Note that the test group classification `PT`/`ST`/`EX` is *not* to be confused with the *maturity level* of the SE package as discussed in the TriBITS Lifecycle Model. test group classification in no way implies the maturity of the given TriBITS SE Package or piece of code. Instead, the test group is just used to sub-select packages (and pieces of code within those packages) that are the most important to sustain for the various current development group's activities. While more mature code would typically never be classified as Experimental `EX`, there are cases were immature packages may be classified as `PT`. For example, a very important research project may be driving the development of a very new algorithm with *maturity level Research Stable* `RS` or even *Exploratory* `EP` because keeping that code working may be critical to keeping the research project on track.

In addition to just selecting `PT` and `ST` SE packages as a whole, a TriBITS `PT` SE package can also contain conditional code and test directories that get enabled when `${PROJECT_NAME}_SECONDARY_TESTED_CODE=ON`. The

package's <packageDir>/CMakeLists.txt files can contain simple if statements and can use the TRIBITS_SET_ST_FOR_DEV_MODE() function to automatically select extra code to enable when ST is enabled or when the project is in release mode.

Test Test Category

Once a package is even defined due to its parent repository's Repository Test Classification and is enabled consistent with its SE Package Test Group, then the set of individual test executables and test cases that are included or not depends on the CATEGORIES argument in the functions TRIBITS_ADD_EXECUTABLE(), TRIBITS_ADD_TEST() and TRIBITS_ADD_ADVANCED_TEST() . This **Test Test Category** defines the last "knob" that the development team has in selecting what tests get run in a particular test scenario described in the section Nested Layers of TriBITS Project Testing. Each of the currently allowed test test categories are described below.

The currently allowed values for the *Test Test Category* are BASIC, CONTINUOUS, NIGHTLY, WEEKLY, and PERFORMANCE. The test test categories BASIC, CONTINUOUS, NIGHTLY, and WEEKLY are subsets of each other. That is, a BASIC test automatically is included in the set of CONTINUOUS, NIGHTLY, and WEEKLY tests. Tests are enabled based on their assigned test test category matching the categories set in the CMake cache variable ${PROJECT_NAME}_TEST_CATEGORIES.

- Tests marked **BASIC** represent key functionality that is needed by nearly every developer that works on the project and so must be protected at all times and are therefore included in Pre-Push CI Testing. Tests marked as BASIC are enabled for the values of ${PROJECT_NAME}_TEST_CATEGORIES of BASIC, CONTINUOUS, NIGHT, and WEEKLY. The category BASIC is the default test test category given to all test executables and tests that don't specify the CATEGORIES argument.

- Tests marked **CONTINUOUS** also represent importantly functionality but are typically not run in Pre-Push CI Testing testing but instead are run in Post-Push CI Testing and Nightly Testing. Tests marked as CONTINUOUS are enabled for the values of ${PROJECT_NAME}_TEST_CATEGORIES equal to CONTINUOUS, NIGHT, and WEEKLY. A test may be marked CONTINUOUS and not BASIC for a few different reasons. For example, the code needed to run the test may take too long to build or the test itself may take too long to run in order to afford including it in Pre-Push CI Testing.

- Tests marked **NIGHTLY** usually take even longer to build and/or run than CONTINUOUS tests and therefore are too expensive to include in Post-Push CI Testing. Tests may also be marked as NIGHTLY even if they might run relatively fast if there is a desire to not cause the CI server to fail if these tests fail. In this case, the decision is to take the testing and maintenance of these tests and the capabilities they represent "offline" so that they don't influence the daily development cycle for the project but instead are addressed in a "secondary feedback loop". Tests marked as NIGHTLY are enabled for the values of ${PROJECT_NAME}_TEST_CATEGORIES equal to NIGHT, and WEEKLY.

- Tests marked **WEEKLY** are usually reserved for very expensive tests that are too expensive to run nightly. WEEKLY tests may only be run once a week (see Weekly Testing) but may be run on shorter or longer time intervals depending on circumstances (e.g. the availability of test machines and free processes, just how expensive all of the tests actually are, etc.). Tests marked as WEEKLY are enabled only for the value of ${PROJECT_NAME}_TEST_CATEGORIES equal WEEKLY.

- Tests marked **PERFORMANCE** are a special category of tests that are specially designed to measure the run-time performance of parts of the software (see Performance Testing). Tests marked as PERFORMANCE are enabled only for the value of ${PROJECT_NAME}_TEST_CATEGORIES equal PERFORMANCE.

Every TriBITS project has a default setting for ${PROJECT_NAME}_TEST_CATEGORIES that is set for a basic cmake configure of the project (see ${PROJECT_NAME}_TEST_CATEGORIES_DEFAULT for more details). In addition, the different testing processes described in the section Nested Layers of TriBITS Project Testing set this to different values.

6.2 Nested Layers of TriBITS Project Testing

Now that the different types of Test Classifications for Repositories, Packages, and Tests have been defined, this section describes how these different test-related classifications are used to select repositories, packages (and code) and tests to run in the standard project testing processes. More than any other section in this document, this section will describe and assume a certain class of software development processes (namely agile processes) where testing and *continuous*

integration (CI) are critical components. However, detailed descriptions of these processes are deferred to the later sections Pre-push Testing using checkin-test.py and TriBITS Package-by-Package CTest/Dash Driver.

The standard TriBITS project testing processes are:

- Pre-Push CI Testing
- Post-Push CI Testing
- Nightly Testing
- Weekly Testing
- Performance Testing

These standard testing processes are outlined in more detail below and show how the different test-related categories are used to define each of these.

Pre-Push CI Testing

The first level of testing is *Pre-Push CI Testing* that is performed before changes to the project are pushed to the master branch(es) in the global repository(s). With TriBITS, this type of testing and push is typically done using the checkin-test.py script and this category of testing is described in much more detail in Pre-push Testing using checkin-test.py. All of the "default builds" used with the `checkin-test.py` script select repositories, SE packages and code, and individual tests using the following test-related classifications:

Classification Type	**Classification**	**(See Reference)**
Repository Test Classif.	`Continuous`	(Repository Test Continuous)
SE Package Test Group	`PT`	(PT)
Test Test Category	`BASIC`	(see Test Test Category BASIC)

Typically a TriBITS project will define a "standard development environment" which is comprised of a standard development compiler (e.g. GCC 4.6.1), TPL versions (e.g. OpenMPI 1.4.2, Boost 4.9, etc.), and other tools (e.g. cmake 2.8.5, git 1.8.2, etc.). This standard development environment is expected to be used to test changes to the project's code before any push. By using a standard development environment, if the code builds and all the tests pass for the "default" pre-push builds for one developer, then that maximizes the probability that the code will also build and all tests will pass for every other developer. This is critical to keep the development team maximally productive. Portability is also important for most projects but portability testing is best done in a secondary feedback look using Nightly Testing builds. TriBITS has some support for helping to set up a standard software development environment as described in section TriBITS Development Toolset.

The basic assumption of all CI processes (including the one described here) is that if anyone pulls the project's development sources at any time, then all of the code will build and all of the tests will pass for the "default" pre-push build cases. For a TriBITS project, that means that the project's --default-builds (see above) will all pass for every package. All of these software development processes make that basic assumption and agile software development methods fall apart if this is not true.

Post-Push CI Testing

After changes are pushed to the master branch(es) in the global repository(s), *Post-Push CI Testing* is performed where a CI server detects the changes and immediately fires off a CI build using CTest to test the changes and the results are posted to a CDash server (in the "Continuous" section on the project's dashboard page). This process is driven by CTest driver code that calls TRIBITS_CTEST_DRIVER() as described in the section TriBITS Package-by-Package CTest/Dash Driver. Various types of specific CI builds can be constructed and run (see CTest/CDash CI Server) but these post-push CI builds typically select repositories, SE packages and code, and individual tests using the following test-related classifications:

Classification Type	**Classification**	**(See Reference)**
Repository Test Classif.	`Continuous`	(Repository Test Continuous)
SE Package Test Group	`PT` & `ST`	(PT and ST)

... continued on next page

Classification Type	Classification	(See Reference)
Test Test Category	CONTINUOUS	(see Test Test Category CONTINUOUS)

Post-push CI testing would assume to use the same standard development environment as used for Pre-Push CI Testing. Also, the project may also choose to run additional automated post-push CI builds that exactly match the pre-push CI default builds to help check on the health of these builds continuously and not just rely on the development team to always perform the pre-push CI builds correctly before pushing.

Nightly Testing

In addition to pre-push and post-push CI testing, a typical TriBITS project will set up multiple *Nightly Testing* builds (or once-a-day builds, they don't need to only run at night). These builds are also driven by code CTest driver scripts described in the section TriBITS Package-by-Package CTest/Dash Driver and post results to the project's CDash server (in the "Nightly" section on the project's dashboard page). Nightly builds don't run in a continuous loop but instead are run once a day (e.g. driven by a cron job) and there tends to be many different nightly build cases that test the project using different compilers (e.g. GCC, Intel, Microsoft, etc., and different versions of each), different TPL versions (e.g. different OpenMPI versions, different MPICH versions, etc.), different platforms (e.g. Linux, Windows, etc.), and varying many other options and settings on many different platforms. What all nightly builds have in common is that they tend to select repositories, SE packages and code, and individual tests using the following test-related classifications:

Classification Type	Classification	(See Reference)
Repository Test Classif.	Nightly	(Repository Test Nightly)
SE Package Test Group	PT & ST	(PT and ST)
Test Test Category	NIGHTLY	(see Test Test Category NIGHTLY)

The nightly builds comprise the basic "heart beat" for the project.

Weekly Testing

Weekly Testing builds are just an extension to the Nightly Testing builds that add on more expensive tests marked using the Test Test Category WEEKLY. For projects that define weekly tests and weekly builds, individual test cases can typically take 24 hours or longer to run so they can't even be run every day in nightly testing. What standard weekly builds have in common is that they tend to select repositories, SE packages and code, and individual tests using the following test-related classifications:

Classification Type	Classification	(See Reference)
Repository Test Classif.	Nightly	(Repository Test Nightly)
SE Package Test Group	PT & ST	(PT and ST)
Test Test Category	WEEKLY	(see Test Test Category WEEKLY)

Project developer teams should strive to limit the number of test cases that are marked as WEEKLY since these tests will *not* get run nightly and developers will tend to never enable them when doing more extensive testing using `--st-extra-builds` with the the checkin-test.py script in extended pre-push testing.

Performance Testing

Performance Testing builds are a special class of builds that have tests that are specifically designed to test the runtime performance of a particular piece of code or algorithm. These tests tend to be sensitive to loads on the machine and therefore typically need to be run on an unloaded machine for reliable results. Details on how to write good performance tests with hard pass/fail time limits is beyond the scope of this document. All TriBITS does is to define the special Test Test Category PERFORMANCE to allow TriBITS packages to declare these tests in a consistent way so that they can be run along with performance tests defined in other TriBITS packages. From a TriBITS standpoint, all performance testing builds would tend to select repositories, SE packages and code, and individual tests using the following test-related classifications:

Classification Type	Classification	(See Reference)
Repository Test Classif.	Nightly	(Repository Test Nightly)
SE Package Test Group	PT & ST	(PT and ST)
Test Test Category	PERFORMANCE	(see Test Test Category PERFORMANCE)

6.3 Pre-push Testing using checkin-test.py

CMake provides the integrated tool CTest (executable `ctest`) which is used to define and run different tests. However, a lot more needs to be done to effectively test changes for a large project before pushing to the master branch(es) in the main repository(s). Things get especially complicated and tricky when multiple version-control (VC) repositories are involved. The TriBITS system provides the tool `checkin-test.py` for automating the process of:

1) Determining if the local VC repo(s) are ready to integrate with the remote master branch(es)
2) Pulling and integrating the most current changes from the remote VC repo(s)
3) Figuring out what TriBITS packages need to be enabled and testing (by examining VC diffs)
4) Configuring only the necessary TriBITS packages and tests (and their downstream dependencies by default) and building, running tests, and reporting results (via email).
5) Only if all specified builds and tests pass, amending the last commit with the test results, then pushing local commits to the remove VC repos and sending out summary emails.

Every TriBITS project defines one or more "default builds" (specified through the `--defualt-builds` argument) for pre-push CI testing that form the criteria for if it is okay to push code changes or not. The "default builds" select repositories, SE packages and code, and individual test as described in Pre-Push CI Testing. A TriBITS project defines its default pre-push builds using the file <projectDir>/project-checkin-test-config.py. For an example, the file TribitsExampleProject/`project-checkin-test-config.py` is shown below:

```
#
# Define project-specific options for the checkin-test script for
# TribitsExampleProject.
#

configuration = {

    # Default command line arguments
    'defaults': {
        '--send-email-to-on-push': 'trilinos-checkin-tests@software.sandia.gov',
        },

    # CMake options (-DVAR:TYPE=VAL) cache variables.
    'cmake': {

        # Options that are common to all builds.
        'common': [],

        # Defines --default-builds, in order.
        'default-builds': [
            # Options for the MPI_DEBUG build.
            ('MPI_DEBUG', [
                '-DTPL_ENABLE_MPI:BOOL=ON',
                '-DCMAKE_BUILD_TYPE:STRING=RELEASE',
                '-DTribitsExProj_ENABLE_DEBUG:BOOL=ON',
                '-DTribitsExProj_ENABLE_CHECKED_STL:BOOL=ON',
                '-DTribitsExProj_ENABLE_DEBUG_SYMBOLS:BOOL=ON',
                ]),
            # Options for the SERIAL_RELEASE build.
            ('SERIAL_RELEASE', [
```

```
                '-DTPL_ENABLE_MPI:BOOL=OFF',
                '-DCMAKE_BUILD_TYPE:STRING=RELEASE',
                '-DTribitsExProj_ENABLE_DEBUG:BOOL=OFF',
                '-DTribitsExProj_ENABLE_CHECKED_STL:BOOL=OFF',
                ]),
            ], # default-builds

        }, # cmake

    } # configuration
```

This gives the default builds `--default-builds=MPI_DEBUG,SERIAL_RELEASE`. As shown, typically two default builds are defined so that various options can be toggled between the two builds. Typical options to toggle include enabling/disabling MPI and enabling/disabling runtime debug mode checking (i.e. toggle `${PROJECT_NAME}_ENABLE_DEBUG`). Typically, other important options will also be toggled between these two builds. For example, Trilinos toggles the enable/disable of C++ explicit template instantiation support.

Note that both of the default builds shown, including the `MPI_DEBUG` build, actually set optimized compiler flags with `-DCMAKE_BUILD_TYPE:STRING=RELEASE`. What makes this a "debug" build is turning on optional runtime debug-mode checking, not disabling optimized code. This is important so that the defined tests run fast. These `checkin-test.py` builds are **not** designed to support debugging efforts. Instead, they are designed to test changes to the project's code efficiently before pushing changes. A development team should not have to test the compiler's ability to generate non-optimized debug code.

Note that turning on `-DTribitsExProj_ENABLE_CHECKED_STL=ON` as shown above can only be used when the TPLs have no C++ code using the C++ STL or if that particular build points to C++ TPLs compiled checked STL enabled. The TribitsExampleProject default builds do not depend on any C++ TPLs that might use the C++ STL so enabling this option adds additional debug-mode checking for C++ code.

The `checkin-test.py` script is a sophisticated piece of software that is well tested and very robust. The level of testing of this tool is greater than any of the software that it is testing. This is needed so as to provide confidence in the developers that the tool will only push if everything checks out as it should. There are a *lot* of details and boundray cases that one has to consider and a number of use cases that need to be supported. For more detailed documentation, see checkin-test.py --help.

6.4 TriBITS Package-by-Package CTest/Dash Driver

The TriBITS system uses a sophisticated and hightly customized CTest driver script to test TriBITS projects and submit results to a CDash server. The primary code for driving this is contained in the CTest function TRIBITS_CTEST_DRIVER() contained in the file `TribitsCTestDriverCore.cmake`. This script loops through all of the specified TriBITS packages for a given TriBITS project and does a configure, built, and test and submitts results to the specified CDash server incrementally. If the configure or library build of any upstream TriBITS package fails, that TriBITS package is disabled in all downstream TriBITS package builds so as to not propoate sensless failures. Each TriBITS top-level package is assigned its own CDash regression email (see CDash regression email addresses) and each package configure/build/test is given its own row in the CDash build. A CTest script using TRIBITS_CTEST_DRIVER() run in one of two modes. First, it can run standard daily from-scratch builds as described in CTest/CDash Nightly Testing. Second, it can run as a CI server as described in CTest/CDash CI Server. Third, it can run in experimental mode testing a local repository.

CTest/CDash Nightly Testing

When a TriBITS CTest script using TRIBITS_CTEST_DRIVER() is run in "Nightly" testing mode, it builds the project from scratch package-by-package and submits results to the TriBITS project's CDash project on the designated CDash server.

CTest/CDash CI Server

When a TriBITS ctest driver script is used in continuous integration (CI) mode, it starts every day with a clean from-scratch build and then performs incremental rebuilds as new commits are pulled from the master branch in the

main repository. In this mode, a continuous loop is performed after the initial baseline build constantly pulling commits from the master git repository(s). If any package changes are detected (looking at git diffs), then the tests and examples for those packages and all downstream packages are performed using a reconfigure/rebuild. Since all of the upstream package libraries are already built, this rebuild and retest can take place in a fraction of the time of a complete from-scratch build and test of the project.

6.5 TriBITS CDash Customizations

CDash was not really designed to accommodate multi-package, multi-repository VC projects like are supported by TriBITS. However, CDash provides some ability to customize a CDash project and submittals to address missing features. Each TriBITS package is given a CTest/CDash "Label" with the name of the TriBITS package. CDash will then aggregate the different package configure/build/test roles into aggregated "builds". The commits pulled for each of the extra VC repo listed in the <projectDir>/cmake/ExtraRepositoriesList.cmake file, are shown in an uploaded CDash "Notes" file for each TriBITS package confgiure/build/test submit. This uploaded "Notes" file also contains a cleaned up version of the CMakeCache.txt file.

CDash offers numerous features such as the ability to construct a number of different types of queries and is extremely helpful in using past test data.

CDash regression email addresses

Every TriBITS Package has a regression email address associated with it that gets uploaded to a CDash project on a CDash server that is used to determine what email address to use when a package has configure, build, or test failures. Because of the complex organizational nature of different projects and different integration models, a single static email address for a given package in every project build is not practical.

The TriBITS system allows for a Package's regression email to be specified in the following order of precedence:

1) **${REPOSITORY_NAME}_REPOSITORY_OVERRIDE_PACKAGE_EMAIL_LIST** (typically defined in <projectDir>/cmake/ProjectDependenciesSetup.cmake): Defines a single email address for all packages for the repository `${REPOSITORY_NAME}` and overrides all other package email regression specification varaibles. This is typically used by a meta-project to redefined the regression email addresses for the packages in an externally developed repository.

2) **REGRESSION_EMAIL_LIST** (defined in <packageDir>/cmake/Dependencies.cmake): Package-specific email address specified in the packages's `Dependencies.cmake` file using TRIBITS_DEFINE_PACKAGE_DEPENDENCIES().

3) **${REPOSITORY_NAME}_REPOSITORY_EMAIL_URL_ADDRESSS_BASE** (set in <repoDir>/cmake/RepositoryDependenciesSetup.cmake): A base email address specified at the Repository level creating package-specific email addresses (e.g. `<lower-case-package-name>-regression@some.repo.gov`, where `${REPOSITORY_NAME}_REPOSITORY_EMAIL_URL_ADDRESSS_BASE=some.repo.gov`). This variable is used, for example, by the Trilinos project to provide automatic regression email addresses for packages.

4) **${REPOSITORY_NAME}_REPOSITORY_MASTER_EMAIL_ADDRESSS** (set in <repoDir>/cmake/RepositoryDependenciesSetup.cmake): A single email address for all packages specified at the Repository level (e.g. `my-repo-regression@some.repo.gov`). This varaible is used for smaller repositories with smaller development groups who just want all regression emails for the repository's packages going to a single email address. This reduces the overhead of managing a bunch of individual package email addresses but at the expense of spamming too many people with CDash failure emails.

5) **${PROJECT_NAME}_PROJECT_EMAIL_URL_ADDRESSS_BASE** (set in <projectDir>/cmake/ProjectDependenciesSetup.cmake): A base email address specified at the Project level creating package-specific email addresses (e.g. `<lower-case-package-name>-regression@some.project.gov`, where `${PROJECT_NAME}_PROJECT_EMAIL_URL_ADDRESSS_BASE=some.project.gov`). If not already set, this variable will be set to `${REPOSITORY_NAME}_REPOSITORY_EMAIL_URL_ADDRESSS_BASE` for the first repostory processed that has this set. This behavior is used, for example by the Trilinos project to automatically assign email addresses for add-on packages and was added to maintain backward compatibility.

6) **${PROJECT_NAME}_PROJECT_MASTER_EMAIL_ADDRESSS** (set in <projectDir>/cmake/ProjectDependenciesSetup.cmake): A single default email address for all packages specified at the Project level (e.g. `my-project-regression@some.project.gov`). If not already set, this variable will be set to `${REPOSITORY_NAME}_REPOSITORY_MASTER_EMAIL_ADDRESSS` for the first repostory processed that has

this set. Every meta-project should set this varaible so that it will be the defualt email address for any new package added.

WARNING: If any of the email lists or URL string variables listed above are set to `"OFF"` or `"FALSE"` (or some other value that CMake interprests as false, see `CMake Language Overivew and Gotchas`) then the varaibles are treated as empty and not set.

If a TriBITS project does not use CDash, then no email address needed to be assigned to packages at all (which will be the case if none of the above variables are set).

As a general rule, repository-level settings override project-level settings and package-level settings override both. Also, a project can redefine a reposiotry's regression email list settings by resetting the variables in the project's <projectDir>/cmake/ProjectDependenciesSetup.cmake file.

All of the email dependency managment logic must be accessable by just running the macro:

```
TRIBITS_READ_PACKAGES_PROCESS_DEPENDENCIES_WRITE_XML()
```

The above email address configuration variables are read from the Repository and Project files <repoDir>/cmake/RepositoryDependenciesSetup.cmake and <projectDir>/cmake/ProjectDependenciesSetup.cmake, respectively. The `RepositoryDependenciesSetup.cmake` files are read first in the specified repository order followed up by reading the `ProjectDependenciesSetup.cmake` file. In this way, the project can override any of the repository settings.

Here is a short review of the precedence order for how regression email addresses are selected for a given package:

1) Package-specific email list is selected if defined (unless an override is in place).

2) Repository-level option is selected over a project-level option.

3) Default email form with repository or project address base is selected over single repository or project email address.

4) If none of the above are selected, then no email address is assigned.

What the above setup does is it results in the TriBITS system (in the TRIBITS_CTEST_DRIVER() function called in `ctest -S` script) creating a file called `CDashSubprojectDependencies.xml` that gets sent to the CDash server. CDash then takes this file and creates, or updates, a set of CDash users and sets up a mapping of Labels (which are used for TriBITS package names) to CDash user emails addresses. CDash is automatically set up to process this XML file and create and updates CDash users. It is not, however, set up to remove labels from existing users. Therefore, if you change a TriBITS package's CDash regression email list (using one of the methods described above), then you need to manually remove the associated labels from the old email address. CDash will not remove them for you.

Therefore, to change the mapping of CDash regression email addresses to TriBITS packages, you must perform the actions:

1) Change the TriBITS CMake files as described above that will result in the desired email addresses in the `CDashSubprojectDependeinces.xml` file. You can debug this by running the `checkin-test.py` script and seeing what gets written in the generated <Project>PackageDependencies.xml file in the `CHECKIN` directory.

2) Log onto the CDash server using an administrator account and then remove the auto-generated account for the CDash user email address for which labels are being removed (i.e. no longer associated with a TriBITS package). This is needed since CDash seems to be unable to remove labels from an existing CDash user (however this might be fixed in a current version of CDash).

3) The next time a CDash submit is performed by a CTest driver script calling TRIBITS_CTEST_DRIVER(), the CDash user associated with the mail list with labels being removed will get automatically recreated with the right list of labels (according to the current `CDashSubprojectDependencies.xml` file). Also, any new CDash users for new email addresses will be created.

Hopefully that should be enough clues to manage the mapping of CDash regression email lists to TriBITS packages.

7 Multi-Repository Support

TriBITS has built-in support for projects involving multiple TriBITS Repositories which contain multiple TriBITS Packages. The basic configuration, build, and test of such projects requires only raw CMake/CTest, just like any other

CMake project (see TriBITS System Project Dependencies). Every TriBITS project automatically supports tacking on add-on TriBITS packages and TPL through the `${PROJECT_NAME}_EXTRA_REPOSITORIES` cmake cache variable as described in Enabling extra repositories with add-on packages. In addition, a TriBITS project can be set up from the start to pull in other TriBITS Repositories using the <projectDir>/cmake/ExtraRepositoriesList.cmake file. A special form of this is a TriBITS Meta-Project that contains no TPLs or packages of its own. The ability to create meta-projects is what allows TriBITS to be used to provide the build of large aggregations of software.

To help set up a development environment for multi-repository TriBITS projects, TriBITS provides some extra development tools based on Python. The primary tools supporting multi-repostory project are the checkin-test.py tool and the egdist tool.

For projects with a standard set of extra repositories defined in the <projectDir>/cmake/ExtraRepositoriesList.cmake file, the `checkin-test.py` script only requires passing in the option `--extra-repos-file=projec` and `--extra-repos-type=Continuous` (or `Nightly`, see Repository Test Classification) and it will automatically perform all of the various actions for all of the selected repositories. See checkin-test.py and checkin-test.py --help for more details.

The tool **egdist** is a simple Python script that just distributes the listed eg/git command across a set of git repos. It is not specific to TriBITS at all. It ony requires that a base git repo and a set of zero or more git repos cloned under it. For example, consider the TriBITS meta-project given in the ExtraRepositoriesList.cmake file:

```
TRIBITS_PROJECT_DEFINE_EXTRA_REPOSITORIES(
  ExtraRepo1  ""  GIT  someurl.com:/ExtraRepo1  ""  Continuous
  ExtraRepo2  packages/SomePackage/Blah  GIT  someurl2.com:/ExtraRepo2  NOPACKAGES  Ni
  ExtraRepo3  ""  HG  someurl3.com:/ExtraRepo3 ""  Continuous
  ExtraRepo4  ""  SVN  someurl4.com:/ExtraRepo4 ""  Nightly
  )
```

This would be layed out a directories as:

```
BaseRepo/
  .git/
  .gitignore
  ExtraRepo1/
    .git/
  packages/SomePackage/Blah/
    .git/
  ExtraRepo3/
    .git/
  ExtraRepo4/
    .git/
```

to use `egdist` with this aggregate project, one would first set up the file `BaseRepo/.egdist` to contain just:

```
ExtraRepo1
packages/SomePackage/Blah
ExtraRepo3
ExtraRepo4
```

and one would set up the tracked ignore file `BaseRepo/.gitignore` which contains the lines:

```
/ExtraRepo1/
/packages/SomePackage/Blah/
/ExtraRepo3/
/ExtraRepo4/
```

Common aggregate commands then run under the `BaseRepo/` directory are:

```
# See status of all repos at once
egdist status
```

```
# Pull updates to all
egdist pull

# Push local commits to tracking branches
egdist push
```

See *egdist --help* for more details.

The TriBITS approach to managing multiple VC repos described above works well for order 10 or so VC repos but will not scale well to order 30 or more. For larger numbers of VC repos, one should consider nested integration creating snapshot git repos that aggregate several related repositories. Another approach might be to use git submodules. However, note that these tools and processes described here are currently **not** set up to support aggregate VC repos that use git submodules. The design decision with TriBITS was to explicitly handle the different git VC repos. There are advantages and disadvantages to use git submodules verses the approach currently supported in TriBITS using egdist and <projectDir>/cmake/ExtraRepositoriesList.cmake. It is possible that TriBITS will add support for aggregate git repos that use git submodules but only if there are important projects that choose to use them. The discussion of these various approaches and strategies to dealing with aggregate repos is beyond the scope of this document.

8 Development Workflows

In this section, the typical development workflows for a TriBITS project are described. First, the Basic Development Workflow for a sinlge-repository TriBITS project is described. This is followed up with a slightly more complex Multi-Repository Development Workflow.

8.1 Basic Development Workflow

The basic development workflow of a TriBITS project is not much different than with any CMake project that uses CTest to define and run tests. One pulls updates from the master VC repo then configures with `cmake`, and iteratively builds, runs tests, adds files, changes files, etc. The major difference is that a well constructed development process will use the checkin-test.py script to test and push all changes that affect the build or the tests. The basic steps in configuring, building, running tests, etc., are given in the project's <Project>BuildQuickRef. file (see Project-Specific Build Quick Reference).

8.2 Multi-Repository Development Workflow

The development workflow for a project with multiple VC repos is very similar to a project with just a single VC repo if the project provides a standard <projectDir>/cmake/ExtraRepositoriesList.cmake file. The major difference is in making changes, creating commits, etc. The egdist tool makes these steps easier and has been shown to work fairly well for up to 20 extra VC repos (as used in the CASL VERA project). The checkin-test.py script automatically handles all of the details of pulling, diffing, pushing etc. to all the VC repos.

9 Howtos

While the rest of this document provides all of the information one would need to construct and maintain a TriBITS project, this section provides short, succinct lists of the steps to accomplish a few common tasks.

9.1 How to Add a new TriBITS Package

To add a new TriBITS package, it is recommended to take the template from one of the TribitsExampleProject packages that most closely fits the needs of the new package and modify it for the new package. For example, the files for the `SimpleCxx` package can be copied one at a time and modified for the new package.

To add a new TriBITS package (with no subpackages), do the following:

1) Chose a name `<packageName>` for the new package and which TriBITS repository (`<repoDir>`) to put the package into. **WARNING!** The choosen name `<packageName>` must be unique across all TriBITS repositories (see Globally unique TriBITS package names).

2) Create the directory `<repoDir>/<packageDir>` for the new package and put in skeleton files for <packageDir>/cmake/Dependencies.cmake and <packageDir>/CMakeLists.txt. Set the desired upstream TPL and SE package dependenices in the new `Dependencies.cmake` file but initially comment out everything in the `CMakeLists.txt` file except for the TRIBITS_PACKAGE() and TRIBITS_PACKAGE_POSTPROCESS() commands.

3) Add the line for the new package to the <repoDir>/PackagesList.cmake file after all of its upstream dependent packages. If a mistake is made and it is listed before one of its upstream dependent packages, the TriBITS CMake code will catch this and issue an error.

4) Configure the TriBITS project enabling the new empty package `<packageName>`. This will enabled the listed dependencies.

5) Incrementally fill in the package's `CMakeLists.txt` files defining libraries, executables, tests and examples. The project should be built and tests run as new pieces are added.

Once the new package is defined, downstream SE packages can define dependencies on this new package.

9.2 How to Add a new TriBITS Package with Subpackages

Adding a new package with subpackages is similary to adding a new regular packages described in How to Add a new TriBITS Package. Again, it is recommended that one copies an example package from TribitsExampleProject; this time the `PackageWithSubpackages` package files and directories.

To add a new TriBITS package with packages, do the following:

1) Chose a name `<packageName>` for the new package and which TriBITS repository (`<repoDir>`) to put the package into. **WARNING!** The choosen name `<packageName>` must be unique across all TriBITS repositories (see Globally unique TriBITS package names).

2) Create the directory `<repoDir>/<packageDir>` for the new package and put in skeleton files for <packageDir>/cmake/Dependencies.cmake and <packageDir>/CMakeLists.txt. Initially don't define any subpackages yet and comment out everything in the `CMakeLists.txt` file except for the TRIBITS_PACKAGE() and TRIBITS_PACKAGE_POSTPROCESS() commands.

3) Add the line for the new package to the <repoDir>/PackagesList.cmake file after all of the upstream dependencies of its to-be-defined subpackages.

4) Configure the TriBITS project enabling the new empty package `<packageName>`.

5) Incrementally add the subpackages as described in How to Add a new TriBITS Subpackage, filling out the various `CMakeLists.txt` files defining libraries, executables, tests and examples.

Once the new SE packages are defined, downstream SE packages can define dependencies on these.

9.3 How to Add a new TriBITS Subpackage

Given an existing top-level TriBITS package that is already broken down into subpackages, adding a new subpackage does not require changing any project- or repository-level files. One only needs to add the declaration for the new subpackages in its parent's <packageDir>/cmake/Dependencies.cmake file then fill out the pieces of the new subpackage defined in the section TriBITS Subpackage Core Files. It is recommended to copy files from one of the TribitsExampleProject subpackages in the `PackagesWithSubpackages` package.

To add a new TriBITS subpackage to a top-level package that already has subpackages, do the following:

1) Chose a name `<spkgName>` for the new subpackage which only has to be different than the other subpackages in the parent package. This name gets appended to the parent package's name `<packageName>` to form the SE package name `<packageName><spkgName>`.

2) Create the directory *<packageDir><spkgDir>*‘ for the new package and put in skeleton files for <packageDir>/<spkgDir>/cmake/Dependencies.cmake and <packageDir>/<spkgDir>/CMakeLists.txt. Set the desired upstream TPL and SE package dependenices in the new `Dependencies.cmake` file but initially comment out everything in the `CMakeLists.txt` file except for the TRIBITS_SUBPACKAGE() and TRIBITS_SUBPACKAGE_POSTPROCESS() commands.

3) Add the line for the new subpackage to the argument `SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS` in the macro call TRIBITS_DEFINE_PACKAGE_DEPENDENCIES() in the parent package's <packageDir>/cmake/Dependencies.cmake file after all of its upstream dependent subpackages. If a mistake is made and it is listed before one of its upstream dependent subpackages, the TriBITS CMake code will catch this and issue an error.

4) Configure the TriBITS project enabling the new empty subpackage `<packageName><spkgName>`. This will enabled the listed dependencies.

5) Incrementally fill in the subpackage's `CMakeLists.txt` files defining libraries, executables, tests and examples. The project should be built and tests run as new pieces are added.

9.4 How to Add a new TriBITS TPL

In order for an SE package to define a dependency on a new TPL (i.e. one that has not already been declared in the current repo's or an upstream repo's <repoDir>/TPLsList.cmake file), one must add and modify a few repository-level files.

To add a new TriBITS TPL, do the following:

1) Chose a name `<tplName>` for the new TPL (must be globally unique across all TriBITS repos, see Globally unique TriBITS TPL names) and which TriBITS repository (`<repoDir>`) to define the new TPL in. The right repo is usually the one where the package exists that needs the new TPL dependnecy.

2) Create the TPL find module for the new, e.g. `<repoDir>/tpls/FindTPL<tplName>.cmake` (see TriBITS TPL and TRIBITS_TPL_DECLARE_LIBRARIES() for details). List the default required header files and/or libraries that must be provided by the TPL.

3) Add the line for the new TPL to the <repoDir>/TPLsList.cmake file after any TPLs that this new TPL may depend on.

4) Configure the TriBITS project enabling the new TPL with `TPL_ENABLE_<tplName>=ON` and see that TriBITS finds the TPL correctly.

5) List the new TPL in the package(s) that need this dependency in the package's <packageDir>/cmake/Dependencies.cmake file.

6) If the TPL is an optional TPL for the package, then:

```
#cmakedefine HAVE_<PACKAGE_NAME_UC>_<TPLN_AME_UC>
```

should be added to the package's `<packageName>_config.h.in` file (see *TRIBITS_CONFIGURE_FILE()*) so that the code knows if the TPL is defined or not.

7) Use the TPL in the package's that define the dependnecy on the new TPL, configure, test, etc.

9.5 How to Add a new TriBITS Repository

To add a new TriBITS and/ git VC repository to a TriBITS project that already contains other extra repositories, do the following:

1) Add a row for the new git repo to <projectDir>/cmake/ExtraRepositoriesList.cmake. Commit this file.

2) Add an ignore for the extra repo name to the base project's git repo's `.gitignore` file. Commit this file.

3) Set up the new package dependencies in the existing packages for the new packages in the new repo.

4) Consider the potential for missing upstream repos and packages by using TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES().

See <projectDir>/cmake/ExtraRepositoriesList.cmake for more details and links.

9.6 How to insert a package into an upstream repo

Sometimes it is desired to insert a package into an upstream repo that is defined in a downstream VC repo for for package in that repo to define a dependency on that package. While TriBITS does not provide direct support for doing this, it is easy to set up. For example, say you want to add the package `PackageFromDownstreamRepo` who's source is provided in a downstream repo (i.e. listed later in the <projectDir>/cmake/ExtraRepositoriesList.cmake file). This is done in several different TriBITS projects such as Trilinos (with Dakota wrapped in the TriKota package) and in the MPACT project (with one of their sensitive packages). There are a few different ways to accomplish this that depend on a few details. One issue is if the downstream repo and the extra package is allowed to be missing. If it is, then all of the packages that define a dependency on the package must declare that the package might be missing by calling TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES() in the package's <packageDir>/cmake/Dependencies.cmake file.

10 Additional Topics

In this section, a number of miscellaneous topics and TriBITS features are discussed. These features and topics are either not considered primary fetures of TriBITS (but can be very useful in many situations) or don't neatly fit into one of the other sections.

10.1 TriBITS System Project Dependencies

The basic TriBITS system itself which is used to configure, built, test, create tarballs, and install software that uses the TriBITS system has no dependencies other than a basic installation of CMake (which typically includes the exectuables `cmake`, `ctest`, and `cpack`). Great effort has been expended to implement all of the core functionality of TriBITS just using raw CMake. That means that anyone who needs to configure, build, and install the software just needs a compable CMake implementation. TriBITS is purposfully maintained to require an older version of CMake. At the time of this writing, the mimimum required version of CMake needed to use TriBITS is CMake 2.8.1 (relasesed in March 2010, see CMake Release Wiki). CMake is becoming iniquitous enough that many clients will already have a current-enough version of CMake installed by default on their systems and will therefore not need to download or install any extra software when building and installing a project that uses TriBITS (assuming the necessary compilers etc. required by the project are also installed). If a current-enough version of CMake is not installed on a given system, it is easy to download the source code and all it needs is a basic C++ compiler to build and install.

However, note that a specific TriBITS project is free to use any newer CMake features it wants and therefore these projects will require newer versions of CMake than what is required by TriBITS (see discussion of `CMAKE_MINIMUM_REQUIRED()` in <projectDir>/CMakeLists.txt). But also note that specific TriBITS projects and packages will also require additional tools like compilers, Python, Perl, or many other such dependencies. It is just that TriBITS itself does not require any of these. The goal of TriBITS is not to amke the portability of software that uses it any worse than it already is but instead to make it easier in most cases (that after all is the whole goal of CMake).

While the core TriBITS functionality is just written using raw CMake, the more sophisticated development tools needed to implement the full TriBITS development environment requires Python 2.4 (or higher, but not Python 3.x). Python is needed for tools like `checkin-test.py` and `egdist`. In addition, these python tools are used in TRIBITS_CTEST_DRIVER() to drive automated testing and submittals to CDash. In addition, git is the chosen version control tool for TriBITS and all of the VC related functionality requires git support. But none of this is required for doing the most basic building, testing, or installation of a TriBITS project.

10.2 Project-Specific Build Quick Reference

If a project that uses TriBITS is going to have a significnat user base that will configure, build, and test the project, then having some documentation that explains how to do this would be useful. For this purpose, TriBITS provides a mechanism to quickly create a project-specific build quick reference document in restructured text (RST) format and with HTML and LaTeX/PDF outputs. These documents are generally created in the base project source tree and given then name `<Project>BuildQuickRef.[rst,html,pdf]`. This document consists of two parts. One part is a generic template document:

```
tribits/doc/build_quick_ref/TribitsBuildQuickRefBody.rst
```

provided in the TriBITS source tree that uses the place-holder `<Project>` for the for the real project name. The second part is a project-specific template file:

```
<projectBaseDir>/cmake/<Project>BuildQuickRefTemplate.rst
```

which provides the outer RST doucment (with title, authors, abstract, introduction, other introductory sections). From these two files, the script:

```
tribits/doc/build_quick_ref/create-project-build-quickref.py
```

is used to replace `<Project>` in the `TribitsBuildQuickRefBody.rst` file with the real project name (read from the project's `ProjectName.cmake` file by default) and then generates the read-only files:

```
<projectBaseDir>/
  <Project>BuildQuickRef.rst
  <Project>BuildQuickRef.html
  <Project>BuildQuickRef.pdf
```

To see a simple example of this, see:

```
tribits/doc/examples/TribitsExampleProject/cmake/create-build-quickref.sh
```

A project-indepenent version of this file is provided in the TribitsBuildQuickRef.[rsts,html,pdf] which is referred to many times in this developers guide.

10.3 Project and Repositiory Versioning and Release Mode

TriBITS has built-in support for project and repository versioning and release mode control. When the project contains the file <projectDir>/Version.cmake, it is used to define the project's offical version. The idea is that when it is time to branch for a release, the only file that needs to be changed is the file is <projectDir>/Version.cmake

Each TriBITS repository can also contain a <repoDir>/Version.cmake file that sets varaibles which TriBITS packages in that repository can use to derive development and release version information. If the TriBITS repository also contains a <repoDir>/Copyright.txt file, then the information in `<repoDir>/Version.cmake` and `<repoDir>/Copyright.txt` are used to configure a repository version header file:

```
${${REPOSITORY_NAME}_BINARY_DIR}/${REPOSITORY_NAME}_version.h
```

The configured header file `${REPOSITORY_NAME}_version.h` gives the repository version number in several formats, which allows C/C++ code (or any software that uses the C preprocessor) to write conditional code like:

```
#if Trilinos_MAJOR_MINOR_VERSION > 100200
  /* Contains feature X */
  ...
#else
  /* Does not contain feature X */
  ...
#endif
```

Of course when the TriBITS project and the TriBITS repository are the same directory, the `<projectDir>/Version.cmake` and `<repoDir>/Version.cmake` files are the same file, which works just fine.

10.4 TriBITS Environment Probing and Setup

Part of the TriBITS Framework is to probe the environment, set up the compilers, and get ready to compile code. This was mentioned in the step "Probe and set up the environment" in Full Processing of TriBITS Project Files. This is exectued by the TriBITS macro `TRIBITS_SETUP_ENV()`. Some of to things this macro does are:

Probe and set up the environment:

- Set `CMAKE_BUILD_TYPE`

- Set up for MPI (MPI compilers, etc.)
- Set up C, C++, and Fortran compiler
- Find Perl (sets `PERL_EXECUTABLE`)
- Determine mixed langauge C/Fortran linking
- Set up C++11, OpenMP, and Windows issues
- Find Doxygen
- Perform some other configure-time tests (see output)

At the completion of this part of the processing, the TriBITS CMake project is ready to compile code.

10.5 Configure-time System Tests

CMake has very nice support for defining configure-time checks of the system to help inconfiguring the project. One can check for whether a header file exists or not, if the compiler supports a given data-type or language feature or perform almost any other type of check that one can imagine that can be done using the configured compilers, libraries, system tools, etc. An example was given in TribitsExampleProject. Just following that example, looking at some of the built-in CMake configure-time test modules, and using provided online CMake documentation, one can see how to create a configure-time test for almost anyting.

10.6 Creating Source Distributions

The TriBITS system uses CMake's built-in CPack support to create source distributions in a varity of zipped and tarred formats. TriBITS will automatically add support for CPack when the variable ${PROJECT_NAME}_ENABLE_CPACK_PACKAGING is set to `ON`. The commands for creating a source distribution are described in Creating a tarball of the source tree using the built-in `package_source` build target. The value added by TriBITS is that TriBITS will automatically exclude the source for any pakcages that are not enabled. In addition, the source for non-enabled subpackages can also be excluded depending on the value of ${PROJECT_NAME}_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION. This allows one to create distributions for subsets of a larger project (even a single package in some cases).

Unlike other build systems (like autotools), CMake will put *everything* in the source distribution that it sitting in the source tree by default. Therefore, setting up for a source distribution usually means deciding what extra files and directories should be excluded. Further files can be selected to excluded on a package-by-package based and at the repository level.

Packages can list additional files to be excluded from the source tree using a call to TRIBITS_EXCLUDE_FILES() in their <packageDir>/CMakeLists.txt file.

Files can be excluded at the repostory level in the <repoDir>/cmake/CallbackDefineRepositoryPackaging.cmake which typically just appends the built-in CMake variable `CPACK_SOURCE_IGNORE_FILES`.

There are a number of project-level settings that need to be defined and these are defined in the file <projectDir>/cmake/CallbackDefineProjectPackaging.cmake.

The TribitsExampleProject is set up for creating source distributions and this is demonstrated in one of the tests defined in:

```
tribits/doc/examples/UnitTests/CMakeLists.txt
```

10.7 Multi-Repository Almost Continuous Integration

The checkin-test.py script can be used to implement stagged integration of the various repositories in a multi-repo TriBITS project. This is referred to here as Almost Continuous Integration (ACI). The basic concept of Almost Continuous Integration (ACI) is defined and described in the paper:

Bartlett, Roscoe. Integration Strategies for Computational Science &

Engineering Software. 2009-0655, Second International Workshop on Software Engineering for Computational Science and Engineering, 2009

which can be found here:

https://cfwebprod.sandia.gov/cfdocs/CCIM/docs/CSE_SoftwareIntegration_Strategies%5B2%5D.pdf

See:

- ACI Multi-Git/TriBITS Repo Integration Example
- ACI Local Sync Git Repo Setup
- ACI Integration Build Directory Setup
- ACI Sync Driver Script
- ACI Cron Job Setup
- Addressing ACI Failures and Summary

ACI Introduction

The TriBITS system allows for setting up composite meta-builds of large collections of software pulled in from many different git/TriBITS code repositories. The checkin-test.py script is a key tool to enable the testing of a set of packages in different git/TriBITS repos before pushing to a master 'origin' set of git repos; all in one robust command invocation.

While the `checkin-test.py` script was originally designed and its default behavior is to test a set of local commits created by a developer before pushing changes to one or more (public) git repos, it can also be used to set up an Almost Continuous Integration (ACI) process to keep these various git/TriBITS repos in sync thereby integrating the work of various disconnected development teams and projects. To use the checkin-test.py for ACI requires some setup and changing what the script does a little by passing in additional options that a developer typically never uses.

This section describes how to use the `checkin-test.py` script to implement an ACI process for a given set of git/TriBITS repositories and also provides a little background and context behind ACI.

ACI Multi-Git/TriBITS Repo Integration Example

In order to set up the context for the ACI process, let's take, for example, a simple TriBITS project with two extra repositories:

```
BaseProj/
    ExtraRepo1
    ExtraRepo2
```

Here, `BaseProj` is the base TriBITS project/repository and `ExtraRepo1` and `ExtraRepo2` are extra repositories that supply additional TriBITS packages that are appended to the TriBITS packages defined in `BaseProj` (see <projectDir>/cmake/ExtraRepositoriesList.cmake). Also, let's assume that `BaseProj`, `ExtraRepo1`, and `ExtraRepo2` are developed by three different development teams that all have different funding sources and priorities so they tend not to work together or consider the other efforts too much when developing the software. However, in this example, there is great value in combining all of this software into a single integrated TriBITS meta-project. This combined meta-build is driven by a 4th integration/development team. In this case, the core developers for each of these three different git/TriBITS repos do not test compatibility with the other git/TriBITS repos when pushing commits to their own git/TriBITS repos. This gives three different git repos on three different machines:

- `BaseProj` main repo: Pushed to by the core BaseProj team:

  `url1.gov:/git/BaseProj`

- `ExtraRepo1` main repo: Pushed to by the core ExtraRepo1 team:

  `url2.gov:/git/ExtraRepo1`

- `ExtraRepo2` main repo: Pushed to by the core ExtraRepo2 team:

```
url3.gov:/git/ExtraRepo2
```

Because of the independent development processes of these three teams, unless these development teams maintain 100% backward compatibility w.r.t. the interfaces and behavior of the combined software, one cannot expect at any time to be able to pull the code from these three different git repos and be able to successfully build all of the code and have all of the tests pass. Therefore, how does the 4th integration team expect to be able to build, test, and possibly extend the combined software? In this case, the integration team would set up their own clones of all three git/TriBITS repos on their own machine such as:

Integration project mirrored git repos:

```
url4.gov:/git/BaseProj
url4.gov:/git/ExtraRepo1
url4.gov:/git/ExtraRepo2
```

Once an initial collaboration effort between the integration team and the three other development teams is able to get a version of all three git/TriBITS repos to work correctly in the combined meta-project, these versions (assume the master branches) would be pushed to the git repos on the git integration server `url4.gov`. The state where the TriBITS packages in the three different git/TriBITS repos in the master branch on `url4.gov` all work together correctly constitutes the initial condition for the ACI process described below. From that initial condition, the ACI processes ensures that updates of the git/TriBITS repos on `url4.gov` do not break any builds or tests of the integrated software.

We will use the update of the git/TriBITS `ExtraRepo1` repo keeping the other two git/TriBITS repos `BaseProj` and `ExtraRepo2` constant as the ACI use case in order to describe how to set up an ACI process using the `checkin-test.py` script.

ACI Local Sync Git Repo Setup

In order to set up an ACI process for the multi-git/TriBITS repo example outlined above, first local repos are created by cloning the repos on the integration server url4.gov as follows (all of which become 'origin'):

```
$ cd $SYNC_BASE_DIR
$ git clone url4.gov:/git/BaseProj
$ cd BaseProj
$ git clone url4.gov:/git/ExtraRepo1
$ git clone url4.gov:/git/ExtraRepo2
```

where, `SYNC_BASE_DIR=~/sync_base_dir` for example, which must already be created.

An `.egdist` file can be created to aid in multi-repo git commands using the tool egdist. This file is located in the `BaseProj` directory and can be created as:

```
$ cd $SYNC_BASE_DIR/BaseProj
$ echo ExtraRepo1 > .egdist
$ echo ExtraRepo2 >> .egdist
$ cat .egdist
ExtraRepo1
ExtraRepo2
```

Next, the remotes that define the integration pattern are created as follows:

```
$ cd $SYNC_BASE_DIR/BaseProj
$ git remote add integrate-from url4.gov:/git/BaseProj
$ cd ExtraRep1
$ git remote add integrate-from url2.gov:/git/ExtraRepo1
$ cd ..
$ cd ExtraRepo2
$ git remote add integrate-from url4.gov:/git/ExtraRepo2
$ cd ..
```

This gives the remotes:

```
$ cd $SYNC_BASE_DIR/BaseProj
$ egdist remote -v | grep -v push | grep -v "^$"
** Base Git Repo: BaseProj
origin        url4.gov:/git/BaseProj (fetch)
integrate-from        url4.gov:/git/BaseProj (fetch)
** Git Repo: ExtraRepo1
origin        url4.gov:/git/ExtraRepo1 (fetch)
integrate-from        url2.gov:/git/ExtraRepo1 (fetch)
** Git Repo: ExtraRepo2
origin        url4.gov:/git/ExtraRepo2 (fetch)
integrate-from        url4.gov:/git/ExtraRepo2 (fetch)
```

Above, the remote `integrate-from` is used by the `checkin-test.py` wrapper script (see below) to pull and merge in additional changes that will be tested and pushed to the 'origin' repos on `url4.gov`. In this case, the `BaseProj` and `ExtraRepo2` repos have the remote `integrate-from` that points to the same repos as 'origin' (and will therefore not result in any merging) but the ExtraRepo1 remote `integrate-from` will result in updates being pulled from the main development repo on `url2.gov`, thereby facilitating the update of ExtraRepo1 in the integrated project.

ACI Integration Build Directory Setup

After the git repos are cloned and the remotes are set up, a build base directory is set up as:

```
$ cd $SYNC_BASE_DIR
$ mkdir BUILDS
$ mkdir BUILDS/CHECKIN
```

An ACI wrapper script for `checkin-test.py` is created to drive the clones. It is assumed that this script would be called only once a day and not continuously in a loop (but that is possible as well but is not documented here).

NOTE: Other build directory structures are possible, it all depends how one writes the the `checkin-test.py` wrapper scripts but the above directory structure is fairly standard in the usage of the `checkin-test.py` script.

ACI Sync Driver Script

The sync driver script for this example should be called something like `sync_ExtraRepo1.sh`, placed under version control, and would look something like:

```
#!/bin/bash -e

EXTRA_ARGS=$@

# Set up the environment (i.e. PATH; needed for cron jobs)
...

SYNC_BASE_DIR=~/sync_base_dir
CHECKIN_TEST_WRAPPER=$SYNC_BASE_DIR/BaseProj/sampleScripts/checkin-test-foo.sh

cd $SYNC_BASE_DIR/BUILDS/CHECKIN

$CHECKIN_TEST_WRAPPER \
  --extra-pull-from=integrate-from:master \
  --abort-gracefully-if-no-changes-to-push \
  --send-email-to=base-proj-integrators@url4.gov \
  --send-email-to-on-push=base-proj-integrators@url4.gov \
  --no-append-test-results --no-rebase \
  --do-all --push \
  -j16 \
  --wipe-clean \
  $EXTRA_ARGS
```

NOTE, in the above example `sync_ExtraRepo1.sh` script, the variable `CHECKIN_TEST_WRAPPER` is set to a wrapper script

> BaseProj/sampleScripts/checkin-test-foo.sh

which would be set up to call the `checkin-test.py` script with configure options for the specific machine. The location and the nature of the wrapper script will vary project to project. In some simple cases, `CHECKIN_TEST_WRAPPER` might just be set to be the raw machine-independent `checkin-test.py` script for the project.

A description of each option passed into this invocation of the checkin-test.py script is given below (see checkin-test.py --help):

`--extra-pull-from=integrate-from:master`

> This option instructs the `checkin-test.py` script to pull and merge in commits that define the integration. This sasme remote name and remote branch name has to be the same in all git repos (todo: remove this requirement). If it is not, then this option can't be used and instead the wrapper script should do the pulls up front manually before calling the checkin-test.py script. The disadvantage of doing the pulls manually is that if they fail for some reason, they will not be seen by the `checkin-test.py` script and no notification email would go out. However, integrating 'master' branches in different git repos is a very common use case when good Lean/Agile CI practices are used by all of the projects.

`--abort-gracefully-if-no-changes-to-push`

> The option `--abort-gracefully-if-no-changes-to-push` makes the `checkin-test.py` script gracefully terminate without sending out any emails if after all the pulls, there are no local changes to push to the 'origin' repos. This can happen, for example, if no commits were pushed to the main development git repo for `ExtraRepo1` at `url2.gov:/git/ExtraRepo1` since the last time this sync process was run. This avoids getting confusing and annoying emails like `"PUSH FAILED"`. The reason this option is not generally needed for local developer usage of the checkin-test.py script is that in general a developer will not run the `checkin-test.py` script with `--push` unless they have made local changes; it just does not make any sense at all to do that and if they do by accident, they should get an error email. However, for an automated CI sync process, there is no easy way to know a priori if changes needs to be synced so the script supports this option to deal with that case gracefully.

`--send-email-to=base-proj-integrators@url4.gov`

> The results of the builds will be sent this email address. If you only want an email sent when a push actually happens, you can set `--send-email-to=''` and rely on `--send-email-to-on-push`.

`--send-email-to-on-push=base-proj-integrators@url4.gov`

> A confirmation and summary email will be sent to this address if the push happens. This can be a different email address than set by the `--send-email-to` option. It is highly recommended that a mail list be used for this email address since this will be the only more permanent logging of the ACI process.

`--no-append-test-results --no-rebase`

> These options are needed to stop the `checkin-test.py` script from modifying the commits being tested and pushed from one public git repo to another. The option `--no-append-test-results` is needed to instruct the `checkin-test.py` script to *NOT* amend the last commit with the test results. The option `--no-rebase` is needed to avoid rebasing the new commits pulled from being rebased. While the default behavior of the `checkin-test.py` script is to amend the last commit message and rebase the local commits (which is considered best practice when testing local commits), this is a very bad thing to do when a ACI sync server is only testing and moving commits between public repos. Amending the last commit would change the SHA1 of the commit (just as a rebase would) and would fork the history and mess up a number of workflows that otherwise should work smoothly. Since an

email logging what was tested will go out if a push happens due to the `--send-email-to-on-push` argument, there is no value in appending the test results to the last commit pulled and merged (which will generally not be a merge commit but just a fast-forward). There are cases, however, where appending the test results in an ACI process might be acceptable but they are not discussed here.

`--do-all --push -j16`

These are standard options that always get used when invoking the `checkin-test.py` script and need no further explanation.

`--wipe-clean`

This option is added if you want to make the sync server more robust to changes that might require a clean configure from script. If you care more about using less computer resources and rebuild, remove this option.

Note that the option `--enable-packages` is not set in the above invocation of the `checkin-test.py` script, and therefore the `checkin-test.py` script will decide on its own what packages to test just based on what packages have changed files in the `ExtraRepo1` git/TriBITS extra repository. This is the preferred way to go since any affected packages will automatically be enabled as determined by the TriBITS package dependency structure and therefore this driver script will require no modifications if the the dependency structure changes over time. However, there are cases and various reasons where the exact list of packages to be tested (or not tested) should be specified using the `--enable-packages` option (or the `--disable-packages` option, respectively). However, these should not be needed with well structured portable TriBITS repos and packages.

The sync script can be created and tested locally to ensure that it works correctly first, before setting it as a cron job as described next.

Note, if using this in a continuous sync server that runs many times in a day in a loop, you also want to set the option `--abort-gracefully-if-no-updates` in addition to the option `--abort-gracefully-if-no-changes-to-push`. That is because if the updated repos are in a broken state such that there are always local changes at every CI iteration (becuase they have not been pushed to origin), you don't want to do a new CI build unless something has changed that would otherwise perhaps make the error go away. That allows the CI server to sit ready to try out any change that gets pulled that might allow the integrated build to work and then push the updates.

ACI Cron Job Setup

Once the sync script `sync_ExtraRepo1.sh` has been locally tested, then it should be committed to a version control git repo and then run automatically as a cron job. For example, the cron script shown below would fire off the daily ACI process at 8pm every night:

```
# ----------------- minute (0 - 59)
# |  -------------- hour (0 - 23)
# |  |  ------------ day of month (1 - 31)
# |  |  |  -------- month (1 - 12)
# |  |  |  |  ----- day of week (0 - 7) (Sunday=0 or 7)
# |  |  |  |  |
# *  *  *  *  *  command to be executed
  00 20  *  *  *  ~/sync_base_dir/sync_ExtraRepo1.sh &> ~/sync_base_dir/sync_ExtraRepo1
```

In the above crontab file (set with `'crontab -e'` or `'crontab my-crontab-file.txt'`), the script:

```
~/sync_base_dir/sync_ExtraRepo1.sh
```

is assumed to be a soft symbolic link to some version controlled copy of the ACI sync script. For example, it might make sense for this script to be version controlled in the `BaseProj` repo and therefore the symbolic link would be created with something like:

```
$ cd ~/sync_base_dir/
$ ln -s BaseProj/sampleScripts/sync_ExtraRepo1.sh .
```

Such a setup would ensure that sync scripts would always be up-to-date due to the git pulls part of the ACI process.

Addressing ACI Failures and Summary

After the above cron job starts running, it will send out emails to the email addresses passed into the underlying `checkin-test.py` script. If the emails report an update, configure, build, or test failure, then someone will need to log onto the machine where the ACI sync server is running and investigate what went wrong, just like they would if the were running the `checkin-test.py` script for testing locally modified changes before pushing.

In the above example, only a single git/TriBITS repo is integrated in this ACI sync scripts. For a complete system, other ACI sync scripts would be written to sync the two other git/TriBITS repos in order to maintain some independence. Or, a single ACI sync script that tries to update all three git/TriBITS repos at once would be written and used. The pattern of integrations chosen will depend many different factors.

In summary, the `checkin-test.py` script can be used to set up robust and effective Almost Continuous Integration (ACI) sync servers that can be used to integrate large collections of software in logical pieces at any logical frequency. Such an approach, together with the practice of Regulated Backward Compatibility and Deprecated Code, can allow very large collections of software to be kept integrated in short time intervals using ACI.

10.8 TriBITS Dashboard Driver

TriBITS also includes a system based on CMake/CTest/CDash to drive the builds of a TriBITS project and post results to another CDash project. This system is contained under the directory:

```
tribits/ctest/tdd/
```

If the TriBITS project name is `<projectName>`, the TDD driver CDash project is typically called `<projectName>Driver`. Using CTest to drive the Nightly builds of a TriBITS project makes sense because CTest can run different builds in parallel, can time-out builds that are taking too long, and will report results to a dashboard and submit notification emails when things fail. However, this is the most confusing and immature part of the TriBITS system and is completely independent from the TriBITS TriBITS Package-by-Package CTest/Dash Driver using the TRIBITS_CTEST_DRIVER().

10.9 Regulated Backward Compatibility and Deprecated Code

The motivation and ideas behind regulated backward compatibility and deprecated code are described in the TriBITS Lifecycle Model document. Here, the details of the implementation in the TriBITS system are given and how transitions between non-backward compatible versions is accomplished.

10.10 Wrapping Exterally Configured/Built Software

It is possible to take an external piece of software that uses an uses any arbitrary builds system and wrap it as a TriBITS package and have it integrate in with the package dependency infrastructure. The TribitsExampleProject package `WrapExternal` shows how this can be done.

While it is possible to wrap an exteranlly configured and built piece of software as a TriBITS package, is is usually must better to just go ahead and create a TriBITS build system for the software. For projects that use a raw CMake build system, a TriBITS build build can be created side by side using a number of approaches. The common approach that is not too invasive is to create a `CMakeLists.tribits.txt` file along side every native `CMakeLists.txt` file and have the native `CMakeList.txt` file defined like:

```
IF (DOING_A_TRIBITS_BUILD)
  INCLUDE("${CMAKE_CURRENT_SOURCE_DIR}/CMakeLists.tribits.txt")
  RETURN()
ENDIF()

# Rest of native CMakeLists.txt file ...
```

Experience from the CASL VERA project shows that, overall, there is less hassell and less work and better portability when creating a native TriBITS build, even it it is a secondary build system for a given piece of software.

10.11 TriBITS Development Toolset

Most TriBITS projects need a git, a compiler (e.g. GCC), MPI, and a number of otehr standard TPLs and tools in order to develop on the project sources. To this end, TriBITS contains some helper scripts for downloading, configuring, building, and installing packages like git, cmake, GCC, OpenMPI, and others needed to set up a development environment. These tools are used to set up development environments on new machines for projects like Trilinos and CASL VERA. Scrits with names like `install-gcc.py` are defined which pull sources from public git repos then configure, build, and install into specified installation directories.

11 References

SCALE http://scale.ornl.gov/

12 TriBITS Detailed Reference Documentation

The following subsections contain detailed reference documentation for the various TriBITS variables and functions and macros that are used by TriBITS.

12.1 TriBITS Global Project Settings

TriBITS defines a number of global project-level settings that can be set by the user and can have their default determined by each individual TriBITS project. If a given TriBITS project does not define its own default, a reasonble default is set by the TriBITS system automatically. These options are defined and are set, for the most part, in the internal TriBITS function `TRIBITS_DEFINE_GLOBAL_OPTIONS_AND_DEFINE_EXTRA_REPOS()` in the TriBITS CMake code file `TribitsGlobalMacros.cmake` which gets called inside of the `TRIBITS_PROJECT()` macro. That function and that file are the definitive source the options that a TriBITS project takes and what the default values are but we strive to document them here as well. Many of these global options (i.e. cache variables) such as `${PROJECT_NAME}_<SOME_OPTION>` allow the project to define a default by setting a local variable `${PROJECT_NAME}_<SOME_OPTION>_DEFAULT` as:

```
SET(${PROJECT_NAME}_<SOME_OPTION>_DEFAULT <someDefault>)
```

either in its top-level `CMakeLists.txt` file or in its `ProjectName.cmake` file. If `${PROJECT_NAME}_<SOME_OPTION>_DEFAULT` is not set by the project, then TriBITS provides a reasonable default value. The TriBITS code for this looks like:

```
IF ("${${PROJECT_NAME}_<SOME_OPTION>_DEFAULT}" STREQUAL "")
  SET(${PROJECT_NAME}_<SOME_OPTION>_DEFAULT <someDefault>)
ENDIF()

ADVANCED_SET( ${PROJECT_NAME}_<SOME_OPTION>
  ${PROJECT_NAME}_<SOME_OPTION>_DEFAULT}
  CACHE BOOL "[documentation]."
  )
```

where `<SOME_OPTION>` is the option name like `TEST_CATEGORIES` and `<someDefault>` is the default set by TriBITS if the project does not define a default. In this way, if the project sets the variable `${PROJECT_NAME}_<SOME_OPTION>_DEFAULT` before this code exeutates, then `${${PROJECT_NAME}_<SOME_OPTION>_DEFAULT}` will be used as the default for the cache variable `${PROJECT_NAME}_<SOME_OPTION>` which, of course, can be overridden by the user when calling `cmake` in a number of ways.

Most of these global options that can be overridden externally by setting the cache variable `${PROJECT_NAME}_<SOME_OPTION>` should be documented in the Project-Specific Build Quick Reference document. A generic version of this document is found in TribitsBuildQuickRef.[rsts,html,pdf]. Some of the more unusual options that might only be of interest to developers mentioned below may not be documented in `<Project>BuildQuickRef.[rst,html,pdf]`.

The global project-level TriBITS options for which defaults can be provided by a given TriBITS project are:

- ${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES
- ${PROJECT_NAME}_ENABLE_Fortran
- ${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS
- ${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES
- ${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES
- ${PROJECT_NAME}_GENERATE_EXPORT_FILE_DEPENDENCIES
- ${PROJECT_NAME}_ELEVATE_ST_TO_PT
- ${PROJECT_NAME}_ENABLE_CPACK_PACKAGING
- ${PROJECT_NAME}_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION
- ${PROJECT_NAME}_CPACK_SOURCE_GENERATOR
- ${PROJECT_NAME}_TEST_CATEGORIES
- MPI_EXEC_MAX_NUMPROCS

These options are described below.

${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES

If ${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES=ON (the TriBITS default value), then any explicitly enabled packages that have disabled upstream required packages or TPLs will be disabled. If ${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES=OFF, then an configure error will occur. For more details also see TribitsBuildQuickRef.*). A project can define a different default value by setting:

```
SET(${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES_DEFAULT FALSE)
```

${PROJECT_NAME}_ENABLE_Fortran

If `${PROJECT_NAME}_ENABLE_Fortran` is `ON`, then Fortran support for the project will be enabled and the Fortran compiler(s) must be found. By default, TriBITS sets this to `ON` for non-Windows systems (i.e. `WIN32` is not set by CMake) but is `OFF` for a Windows system. A project always requires Fortran, for example, it can set the default:

```
SET(${PROJECT_NAME}_ENABLE_Fortran_DEFAULT TRUE)
```

If a project does not have any native Fortran code a good default would be:

```
SET(${PROJECT_NAME}_ENABLE_Fortran_DEFAULT OFF)
```

NOTE: It is usually not a good idea to always force off Fortran, or any compiler, because extra repositories and packages might be added by someone that might require the compiler and we don't want to unnecessarily limit the generality of a given TriBITS build. Setting the default for all platforms should be sufficient.

${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS

If `${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS` is set to `ON`, then any defined libraries or header files that are listed in calls to TRIBITS_ADD_LIBRARY() will be installed (unless options are passed into TRIBITS_ADD_LIBRARY() that disable installs). If set to `OFF`, then headers and librareis will be installed by default and only `INSTALLABLE` executables added with TRIBITS_ADD_EXECUTABLE() will be installed. However, as described in TribitsBuildQuickRef.*, shared libraries will still be always be installed if enabled since they are needed by the installed executables. The TriBITS default is to set this to `ON`.

For a TriBITS project that primarily is delivering libraries (e.g. Trilinos), then it makes sense to leave the TriBITS default or explicitly set:

```
SET(${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS_DEFAULT ON)
```

For a TriBITS project that is primarily delivering executablers (e.g. VERA), then it makes sense to set the default to:

```
SET(${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS_DEFAULT OFF)
```

${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES

If `${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES` is `ON`, then `Makefile.export.<PACKAGE_NAME>` will get created at configure time in the build tree and installed into the install tree. See TribitsBuildQuickRef.* for details. The TriBITS default is `ON` but a project can decide to turn this off by default by setting:

```
SET(${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES_DEFAULT OFF)
```

A project might want to disable the generation of export makefiles by default if its main purpose is to provide executables. There is no reason to provide an export makefile if libraies and headers are not actaully installed (see ${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS)

${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES

If `${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES` is set to `ON`, then `<PACKAGE_NAME>Config.cmake` files are created at configure time in the build tree and installed into the install tree. These files are used by external CMkae projects to pull in the list of compilers, compiler options, include directories and libraries. The TriBITS default is `ON`. A project can change the default by setting, for example:

```
SET(${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES_DEFAULT OFF)
```

A project would want to turn off the creation and installation of `<PACKAGE_NAME>Config.cmake` files if it was only installing and providing executables. See TribitsBuildQuickRef.* for details.

${PROJECT_NAME}_GENERATE_EXPORT_FILE_DEPENDENCIES

If `${PROJECT_NAME}_GENERATE_EXPORT_FILE_DEPENDENCIES` is `ON`, then the data-structures needed to generate `Makefile.export.<PACKAGE_NAME>` and `<PACKAGE_NAEM>Config.cmake` are created. These data structures are also needed in order to generate export makefiles on demand using the function TRIBITS_WRITE_FLEXIBLE_PACKAGE_CLIENT_EXPORT_FILES(). The default in TriBITS is to turn this `ON` automatically by default if `${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES` or `${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES` are `ON`. Else, by default, TriBITS sets this to `OFF`. The only reason for the project to override the default is to set it to `ON` as with:

```
SET(${PROJECT_NAME}_GENERATE_EXPORT_FILE_DEPENDENCIES_DEFAULT ON)
```

is so that the necessary data-structures are generated in order to use the function TRIBITS_WRITE_FLEXIBLE_PACKAGE_CLIENT_EXPORT_FILES().

${PROJECT_NAME}_ELEVATE_ST_TO_PT

If `${PROJECT_NAME}_ELEVATE_ST_TO_PT` is set to `ON`, then all `ST` SE packages will be elevated to `PT` packages. The TriBITS default is obviously `OFF`. The default can be changed by setting:

```
SET(${PROJECT_NAME}_ELEVATE_ST_TO_PT_DEFAULT ON)
```

There are projects, especially meta-projects, where the distiction between `PT` and `ST` code is not helpful or the assignment of `PT` and `ST` packages in a repository is not appropriate. An example project like this CASL VERA. Changing the default to `ON` allows any packages to be considered in pre-push testing.

${PROJECT_NAME}_ENABLE_CPACK_PACKAGING

If ${PROJECT_NAME}_ENABLE_CPACK_PACKAGING is ON, then CPack support is enabled and some TriBITS code is avoided that is needed to set up data-structures that are used by the built-in CMake target package_source. The TriBITS default is OFF with the idea that the average developer or user will not be wanting to create source distributions with CPack. However, this default can be changed by setting:

```
SET(${PROJECT_NAME}_ENABLE_CPACK_PACKAGING ON)
```

${PROJECT_NAME}_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION

If ${PROJECT_NAME}_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION is TRUE, then the directories for subpackages that are not enabled are left out of the source tarball. This reduces the size of the tarball as much as possible but does require that the TriBITS packages and subpackages be properly set up to allow disabled subpackages from being excluded. The TriBITS default is TRUE but this can be changed by setting:

```
SET(${PROJECT_NAME}_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION_DEFAULT FALSE)
```

${PROJECT_NAME}_CPACK_SOURCE_GENERATOR

The variable ${PROJECT_NAME}_CPACK_SOURCE_GENERATOR determines the CPack source generation types that are created when the package_source target is run. The TriBITS default is set to TGZ. However, this default can be overridded by setting, for example:

```
SET(${PROJECT_NAME}_CPACK_SOURCE_GENERATOR_DEFAULT "TGZ;TBZ2")
```

This variable should generally be set in the file:

```
<projectDir>/cmake/CallbackDefineProjectPackaging.cmake
```

instead of in the base-level CMakeLists.txt file so that it goes along with rest of the project-specific CPack packaging options.

${PROJECT_NAME}_TEST_CATEGORIES

The cache varaible ${PROJECT_NAME}_TEST_CATEGORIES detemines what tests defined using TRIBITS_ADD_TEST() and TRIBITS_ADD_ADVANCED_TEST() will be added for ctest to run (see Test Test Category). The TriBITS default is NIGHTLY for a standard local build. The checkin-test.py script sets this to BASIC by default. A TriBITS project can override the default for a basic configure using, for example:

```
SET(${PROJECT_NAME}_TEST_CATEGORIES_DEFAULT BASIC)
```

The justification for having the default test category be NIGHTLY instead of BASIC is that when someone is enabling a package to develop on it or install it, we want them by default to be seeing the full version of the test suite (shy of the Test Test Category WEEKLY tests which can be very expensive) for the packages they are explictly enabling. Typically they will not be enabling forward/downstream dependent packages so the cost of running the test suite should not be too prohibitive. This all depends on how good of a job the development teams do in making their test suites run fast and keeping the cost of running the tests down. See the section TriBITS Automated Testing for a more detailed discussion.

${PROJECT_NAME}_ENABLE_DEVELOPMENT_MODE

The variable ${PROJECT_NAME}_ENABLE_DEVELOPMENT_MODE switches the TriBITS project from development mode to release mode. The default for this variable ${PROJECT_NAME}_ENABLE_DEVELOPMENT_MODE_DEFAULT should be set in the project's <projectDir>/Version.cmake file and switched from ON to OFF when creating a release (see Project and Repositiory Versioning and Release Mode). When ${PROJECT_NAME}_ENABLE_DEVELOPMENT_MODE = ON, several other variables are given defaults appropriate for development mode. For example, ${PROJECT_NAME}_ASSERT_MISSING_PACKAGES is set to ON by default in development mode but is set to OFF by default in release mode. In addition, strong compiler warnings are enabled by default in development mode but are disabled by default in release mode. Thsi variable also affects the behavior of TRIBITS_SET_ST_FOR_DEV_MODE().

MPI_EXEC_MAX_NUMPROCS

The varaible `MPI_EXEC_MAX_NUMPROCS` gives the maximum number of processes for an MPI test that will be allowed as defined by TRIBITS_ADD_TEST() and TRIBITS_ADD_ADVANCED_TEST(). The TriBITS default is set to be `4` (for no good reason really but it needs to stay that way for backward compatibility). This default can be changed by setting:

```
SET(MPI_EXEC_MAX_NUMPROCS_DEFAULT <newDefaultMax>)
```

While this default can be changed for the project as a whole on all platforms, it is likely better to change this default on a machine-by-machine basis to correspond to the loat that can be accomidated by a given machine (or class of machines). For example if a given machine has 64 cores, a reasonble number for `MPI_EXEC_MAX_NUMPROCS_DEFAULT` is 64.

12.2 TriBITS Macros and Functions

The following subsections give detailed documentation for the CMake macros and functions that make up the core TriBITS system. These are what are used by TriBITS project developers in their `CMakeLists.txt` and other files. All of these functions and macros should be available when processing the project's and package's variables files if used properly. Therefore, no explicit `INCLUDE()` statements should be needed other than the initial include of the `TribitsProject.cmake` file in the top-level <projectDir>/CMakeLists.txt file so the command TRIBITS_PROJECT() can be executed.

TRIBITS_ADD_ADVANCED_TEST()

Function that creates an advanced test defined by stringing together one or more executables and/or commands that is run as a separate CMake -P script with very flixible pass/fail criteria.

This function allows you to add a single CTest test as a single unit that is actually a sequence of one or more separate commands strung together in some way to define the final pass/fail. You will want to use this function to add a test instead of `TRIBITS_ADD_TEST()` when you need to run more than one command, or you need more sophisticated checking of the test result other than just greping STDOUT (i.e. by running programs to examine output files).

Usage:

```
TRIBITS_ADD_ADVANCED_TEST(
  <testName>
  TEST_0 (EXEC <execTarget0> | CMND <cmndExec0>) ...
  [TEST_1 (EXEC <execTarget1> | CMND <cmndExec1>) ...]
  ...
  [TEST_N (EXEC <execTargetN> | CMND <cmndExecN>) ...]
  [OVERALL_WORKING_DIRECTORY (<overallWorkingDir> | TEST_NAME)]
  [FAIL_FAST]
  [KEYWORDS <keyword1> <keyword2> ...]
  [COMM [serial] [mpi]]
  [OVERALL_NUM_MPI_PROCS <overallNumProcs>]
  [CATEGORIES <category0> <category1> ...]
  [HOST <host0> <host1> ...]
  [XHOST <host0> <host1> ...]
  [HOSTTYPE <hosttype0> <hosttype1> ...]
  [XHOSTTYPE <hosttype0> <hosttype1> ...]
  [FINAL_PASS_REGULAR_EXPRESSION <regex> | FINAL_FAIL_REGULAR_EXPRESSION <regex>]
  [ENVIRONMENT <var1>=<value1> <var2>=<value2> ...]
  )
```

Each atomic test case is either a package-built executable or just a basic command. An atomic test command block of arguments takes the form:

```
TEST_<idx>
   (EXEC <exeRootName> [NOEXEPREFIX] [NOEXESUFFIX] [ADD_DIR_TO_NAME]
          [DIRECTORY <dir>]
```

```
    | CMND <cmndExec>)
  [ARGS <arg1> <arg2> ... <argn>]
  [MESSAGE "<message>"]
  [WORKING_DIRECTORY <workingDir>]
  [NUM_MPI_PROCS <numProcs>]
  [OUTPUT_FILE <outputFile>]
  [NO_ECHO_OUTPUT]]
  [PASS_ANY
    | PASS_REGULAR_EXPRESSION "<regex>"
    | PASS_REGULAR_EXPRESSION_ALL "<regex1>" "<regex2>" ... "<regexn>"
    | FAIL_REGULAR_EXPRESSION "<regex>"
    | STANDARD_PASS_OUTPUT
    ]
```

By default, each and every atomic test or command needs to pass (as defined below) in order for the overall test to pass.

Sections:

Overall Arguments (TRIBITS_ADD_ADVANCED_TEST())

Below are given some overall arguments. Remaining overall arguments that control overall pass/fail are described in Overall Pass/Fail (TRIBITS_ADD_ADVANCED_TEST()). (NOTE: All of these arguments must be listed outside of the `TEST_<idx>` blocks, see Argument Parsing and Ordering (TRIBITS_ADD_ADVANCED_TEST())).

`<testName>`

> The name of the test (which will have `${PACKAGE_NAME}_` prepended to the name) that will be used to name the output CMake script file as well as the CTest test name passed into `ADD_TEST()`. This must be the first argument.

`OVERALL_WORKING_DIRECTORY <overallWorkingDir>`

> If specified, then the working directory `<overallWorkingDir>` will be created and all of the test commands by default will be run from within this directory. If the value `<overallWorkingDir>=TEST_NAME` is given, then the working directory will be given the name `${PACKAGE_NAME}_<testName>`. If the directory `<overallWorkingDir>` exists before the test runs, it will be deleted and created again. Therefore, if you want to preserve the contents of this directory between test runs you need to copy the files it somewhere else. This is a good option to use if the commands create intermediate files and you want to make sure they get deleted before a set of test cases runs again.

`FAIL_FAST`

> If specified, then the remaining test commands will be aborted when any test command fails. Otherwise, all of the test cases will be run.

`RUN_SERIAL`

> If specified then no other tests will be allowed to run while this test is running. This is useful for devices(like cuda cards) that require exclusive access for processes/threads. This just sets the CTest test property `RUN_SERIAL` using the built-in CMake function `SET_TESTS_PROPERTIES()`.

`COMM [serial] [mpi]`

If specified, selects if the test will be added in serial and/or MPI mode. See the `COMM` argument in the script TRIBITS_ADD_TEST() for more details.

`OVERALL_NUM_MPI_PROCS <overallNumProcs>`

If specified, gives the default number of processes that each executable command runs on. If `<numProcs>` is greater than `${MPI_EXEC_MAX_NUMPROCS}` then the test will be excluded. If not specified, then the default number of processes for an MPI build will be `${MPI_EXEC_DEFAULT_NUMPROCS}`. For serial builds, this argument is ignored.

`CATEGORIES <category0> <category1> ...`

Gives the test categories for which this test will be added. See TRIBITS_ADD_TEST() for more details.

`HOST <host0> <host1> ...`

The list of hosts for which to enable the test (see TRIBITS_ADD_TEST()).

`XHOST <host0> <host1> ...`

The list of hosts for which **not** to enable the test (see TRIBITS_ADD_TEST()).

`HOSTTYPE <hosttype0> <hosttype1> ...`

The list of host types for which to enable the test (see TRIBITS_ADD_TEST()).

`XHOSTTYPE <hosttype0> <hosttype1> ...`

The list of host types for which **not** to enable the test (see TRIBITS_ADD_TEST()).

`ENVIRONMENT <var1>=<value1> <var2>=<value2> ...`

If passed in, the listed environment varaibles will be set before calling the test. This is set using the built-in test property `ENVIRONMENT`.

TEST_<idx> Test Blocks and Arguments (TRIBITS_ADD_ADVANCED_TEST())

Each test command block `TEST_<idx>` runs either a package-built test executable or some general command executable and is defined as either `EXEC <exeRootName>` or `CMND <cmndExec>` with the arugments:

`EXEC <exeRootName> [NOEXEPREFIX] [NOEXESUFFIX] [ADD_DIR_TO_NAME] [DIRECTORY <dir>]`

If specified, then `<exeRootName>` gives the root name of an executable target that will be run as the command. The full executable name and path is determined in exactly the same way it is in the TRIBITS_ADD_TEST() function (see Determining the Exectuable or Command to Run (TRIBITS_ADD_TEST())). If this is an MPI build, then the executable will be run with MPI using `NUM_MPI_PROCS <numProcs>` or `OVERALL_NUM_MPI_PROCS <overallNumProcs>` (if `NUM_MPI_PROCS` is not set for this test case). If the number of maximum MPI processes allowed is less than this number of MPI processes, then the test will *not* be run. Note that `EXEC <exeRootName>` when `NOEXEPREFIX` and `NOEXESUFFIX` are specified is basically equivalent to `CMND <cmndExec>` except that in an MPI build, `<exeRootName>` is always run using MPI. In this case, you can pass in `<exeRootName>` to any command you would like and it will get run with MPI in MPI mode just link any other command.

`CMND <cmndExec>`

If specified, then `<cmndExec>` gives the executable for a command to be run. In this case, MPI will never be used to run the executable even when configured in MPI mode (i.e. TPL_ENABLE_MPI=ON). If you want to run an arbitrary command using MPI, use `EXEC <fullPathToCmndExec> NOPREFIX NOEXESUFFIX` instead.

By default, the output (stdout/stderr) for each test command is captured and is then echoed to stdout for the overall test. This is done in order to be able to grep the result to determine pass/fail.

Other miscellaneous arguments for each `TEST_<idx>` block include:

`DIRECTORY <dir>`

> If specified, then the executable is assumed to be in the directory given by relative <dir>. See TRIBITS_ADD_TEST().

`MESSAGE "<message>"`

> If specified, then the string in `"<message>"` will be print before this test command is run. This allows adding some documentation about each individual test invocation to make the test output more understandable.

`WORKING_DIRECTORY <workingDir>`

> If specified, then the working directory `<workingDir>` will be created and the test will be run from within this directory. If the value `<workingDir> = TEST_NAME` is given, then the working directory will be given the name `${PACKAGE_NAME}_<testName>`. If the directory <workingDir> exists before the test runs, it will be deleted and created again. Therefore, if you want to preserve the contents of this directory between test runs you need to copy it somewhere else. Using `WORKING_DIRECTORY` for individual test commands allows creating independent working directories for each test case. This would be useful if a single ``OVERALL_WORKING_DIRECTORY was not sufficient for some reason.

`NUM_MPI_PROCS <numProcs>`

> If specified, then <`numProcs`> is the number of processors used for MPI executables. If not specified, this will default to `<overallNumProcs>` from `OVERALL_NUM_MPI_PROCS <overallNumProcs>`.

`OUTPUT_FILE <outputFile>`

> If specified, then stdout and stderr for the test case will be sent to `<outputFile>`. By default, the contents of this file will **also** be printed to STDOUT unless `NO_ECHO_OUT` is passed as well.

`NO_ECHO_OUTPUT`

> If specified, then the output for the test command will not be echoed to the output for the entire test command.

By default, an atomic test line is assumed to pass if the executable returns a non-zero value. However, a test case can also be defined to pass based on:

`PASS_ANY`

> If specified, the test command 'i' will be assumed to pass reguardless of the return value or any other output. This would be used when a command that is to follow will determine pass or fail based on output from this command in some way.

`PASS_REGULAR_EXPRESSION "<regex>"`

> If specified, the test command 'i' will be assumed to pass if it matches the given regular expression. Otherwise, it is assumed to fail.

`PASS_REGULAR_EXPRESSION_ALL "<regex1>" "<regex2>" ... "<regexn>"`

> If specified, the test command 'i' will be assumed to pas if the output matches all of the provided regular expressions. Note that this is not a capability of raw ctest and represents an extension provided by TriBITS.

`FAIL_REGULAR_EXPRESSION "<regex>"`

> If specified, the test command 'i' will be assumed to fail if it matches the given regular expression. Otherwise, it is assumed to pass.

`STANDARD_PASS_OUTPUT`

If specified, the test command 'i' will be assumed to pass if the string expression "Final Result: PASSED" is found in the ouptut for the test.

All of the arguments for a test block `TEST_<idx>` must appear directly below their `TEST_<idx>` argument and before the next test block (see Argument Parsing and Ordering (TRIBITS_ADD_ADVANCED_TEST())).

Overall Pass/Fail (TRIBITS_ADD_ADVANCED_TEST())

By default, the overall test will be assumed to pass if it prints:

```
"OVERALL FINAL RESULT: TEST PASSED"
```

However, this can be changed by setting one of the following optional arguments:

`FINAL_PASS_REGULAR_EXPRESSION <regex>`

If specified, the test will be assumed to pass if the output matches <regex>. Otherwise, it will be assumed to fail.

`FINAL_FAIL_REGULAR_EXPRESSION <regex>`

If specified, the test will be assumed to fail if the output matches <regex>. Otherwise, it will be assumed to fail.

Argument Parsing and Ordering (TRIBITS_ADD_ADVANCED_TEST())

The basic tool used for parsing the arguments to this function is the macro PARSE_ARGUMENTS() which has a certain set of behaviors. The parsing using PARSE_ARGUMENTS() is actually done in two phases. There is a top-level parsing listing the "overall" arguments listed in Overall Arguments (TRIBITS_ADD_ADVANCED_TEST()) that also pulls out the test blocks and then there is a second level of parsing using *PARSE_ARGUMENTS()* for each of the `TEST_<idx>` blocks. Becuase of this usage, there are a few restructions that one needs to be aware of when using `TRIBITS_ADD_ADVANCED_TEST()`. This short sections tries to explain the behaviors and what is allowed and what is not allowed.

For the most part, the overall argument and the arguments inside of any individual `TEST_<idx>` block can be listed can appear in any order but there are restructions related to the grouping of overall arguments and `TEST_<idx>` blocks which are as follows:

- The `<testName>` argument must be the first listed (it is the only positional argument).
- The test cases `TEST_<idx>` must be listed in order (i.e. `TEST_0 ... TEST_1 ...`) and the test cases must be consecutive integers (i..e can't jump from `TEST_5` to `TEST_7`).
- All of the arguments for a test case must appear directly below its `TEST_<idx>` keyword and before the next `TEST_<idx+1>` keyword or before any trailing overall keyword arguments.
- None of the overall arguments (e.g. `CATEGORIES`) can be inside listed inside of a `TEST_<idx>` block but otherwise can be listed before or after all of the `TEST_<idx>` blocks.

Other than that, the keyword argumnets and options can appear in any order.

ToDo: Add some examples of bad argument ordering and what will happen.

Implementation Details (TRIBITS_ADD_ADVANCED_TEST())

Since raw CTest does not support the features provided by this function, the way an advanced test is implemented is that a CMake script with the name `${PACKAGE_NAME}_<testName>.cmake` gets created in the current binary directory that then gets added to CTest using:

```
ADD_TEST(${PACKAGE_NAME}_<testName>
  cmake [other options] -P ${PACKAGE_NAME}_<testName>.cmake)
```

This CMake script then runs the various test cases and checks the pass/fail for each case to determine overall pass/fail and implement other functionality.

Setting Additional Test Properties (TRIBITS_ADD_ADVANCED_TEST())

After this function returns, if the test gets added using `ADD_TEST()` then additional properties can be set and changed using `SET_TEST_PROPERTIES(${PACKAGE_NAME}_<testName> ...)`. Therefore, any tests properties that are not directly supported by this function and passed through the argument list to this wrapper function can be set in the outer `CMakeLists.txt` file after the call to `TRIBITS_ADD_ADVANCED_TEST()`.

Disabling Tests Externally (TRIBITS_ADD_ADVANCED_TEST())

The test can be disabled externally by setting the CMake cache variable `${FULL_TEST_NAME}_DISABLE=TRUE`. This allows tests to be disable on a case-by-case basis. This is the *exact* name that shows up in 'ctest -N' when running the test.

Debugging and Examining Test Generation (TRIBITS_ADD_ADVANCED_TEST())

In order to see if the test gets added and to debug some issues in test creation, one can set the cache variable `${PROJECT_NAME}_VERBOSE_CONFIGURE=ON`. This will result in the printout of some information about the test getting added or not.

Likely the best way to debugging test generation using this function is to examine the generated file `${PACKAGE_NAME}_<testName>.cmake` in the current binary directory (see Implementation Details (TRIBITS_ADD_ADVANCED_TEST())).

TRIBITS_ADD_DEBUG_OPTION()

Add the standard option `${PACKGE_NAME}_ENABLE_DEBUG` for the package.

Usage:

```
TRIBITS_ADD_DEBUG_OPTION()
```

This option is given the default `${${PROJECT_NAME}_ENABLE_DEBUG}` and if true, will set the variable `HAVE_${PACKAGE_NAME_UC}_DEBUG` (to be used in the package's configured header file).

TRIBITS_ADD_EXAMPLE_DIRECTORIES()

Macro called to conditionally add a set of example directories for an SE package.

Usage:

```
TRIBITS_ADD_EXAMPLE_DIRECTORIES(<dir1> <dir2> ...)
```

This macro only needs to be called from the top most CMakeList.txt file for which all subdirectories are all "examples".

This macro can be called several times within a package and it will have the right effect.

Currently, really all it does macro does is to call `ADD_SUBDIRECTORY(<diri>)` if `${PACKAGE_NAME}_ENABLE_EXAMPLES` or `${PARENT_PACKAGE_NAME}_ENABLE_EXAMPLES` are true. However, this macro may be extended in the futgure in order to modify behavior related to adding tests and examples in a uniform way..

TRIBITS_ADD_EXECUTABLE()

Function used to create an executable (typically for a test or example), using the built-in CMake command `ADD_EXECUTABLE()`.

Usage:

```
TRIBITS_ADD_EXECUTABLE(
  <exeRootName>  [NOEXEPREFIX]  [NOEXESUFFIX]  [ADD_DIR_TO_NAME]
  SOURCES <src0> <src1> ...
  [CATEGORIES <category0>  <category1> ...]
  [HOST <host0> <host1> ...]
```

```
  [XHOST <host0> <host1> ...]
  [HOSTTYPE <hosttype0> <hosttype1> ...]
  [XHOSTTYPE <hosttype0> <hosttype1> ...]
  [DIRECTORY <dir>]
  [DEPLIBS <lib0> <lib1> ...]
  [COMM [serial] [mpi]]
  [LINKER_LANGUAGE (C|CXX|Fortran)]
  [DEFINES -D<define0> -D<define1> ...]
  [INSTALLABLE]
  )
```

Sections:

Formal Arguments (TRIBITS_ADD_EXECUTABLE())

`<exeRootName>`

The root name of the exectuable (and CMake target) (see Executable and Target Name (TRIBITS_ADD_EXECUTABLE())).

`NOEXEPREFIX`

If passed in, then `${PACKAGE_NAME}_` is not added the beginning of the executable name (see Executable and Target Name (TRIBITS_ADD_EXECUTABLE())).

`NOEXESUFFIX`

If passed in, then `${${PROJECT_NAME}_CMAKE_EXECUTABLE_SUFFIX}` and not added to the end of the executable name (see Executable and Target Name (TRIBITS_ADD_EXECUTABLE())).

`ADD_DIR_TO_NAME`

If passed in, the directory path relative to the package base directory (with "/" replaced by "_") is added to the executable name (see Executable and Target Name (TRIBITS_ADD_EXECUTABLE())). This provides a simple way to create unique test exectuable names inside of a given TriBITS package. Only test executables in the same directory would need to have unique `<execRootName>` passed in.

`SOURCES <src0> <src1> ...`

Gives the source files that will be compiled into the built executable. By default, these sources are assumed to be in the current working directory or gives the relative path to the current working directory. If `<srci>` is an absolute path, then that full file path is used. This list of sources (with adjusted directory path) are passed into `ADD_EXECUTABLE(<fullExeName> ... )`. After calling this function, the properties of the source files can be altered using `SET_SOURCE_FILE_PROPERTIES()`.

`DIRECTORY <dir>`

If specified, then the soruces for the exectuable listed in `SOURCES <src0> <src1> ...` are assumed to be in the relative or absolute directory `<dir>` instead of the current source directory. This directrory path is prepended to each source file name `<srci>` unless `<srci>` is an absolute path.

`CATEGORIES <category0> <category1> ...`

Gives the test categories for which this test will be added. See TRIBITS_ADD_TEST() for more details.

`HOST <host0> <host1> ...`

The list of hosts for which to enable the test (see TRIBITS_ADD_TEST()).

`XHOST <host0> <host1> ...`

The list of hosts for which **not** to enable the test (see TRIBITS_ADD_TEST()).

`HOSTTYPE <hosttype0> <hosttype1> ...`

The list of host types for which to enable the test (see TRIBITS_ADD_TEST()).

`XHOSTTYPE <hosttype0> <hosttype1> ...`

The list of host types for which **not** to enable the test (see TRIBITS_ADD_TEST()).

`DEPLIBS <lib0> <lib1> ...`

Specifies extra libraries that will be linked to the executable using `TARGET_LINK_LIBRARY()`. Note that regular libraries (i.e. not "TESTONLY") defined in the current SE package or any upstream SE packages do **NOT** need to be listed! TriBITS automatically links these libraries to the executable! The only libraries that should be listed in this argument are either `TESTONLY` libraries, or other libraries that are built external from this CMake project and are not provided through a proper TriBITS TPL. The latter usage is not recommended. External TPLs should be handled as a declared TriBITS TPL. For a `TESTONLY` library, the include directories will automatically be added using:

```
INCLUDE_DIRECTORIES(${<libi>_INCLUDE_DIRS})
```

where `<libi>_INCLUDE_DIRS` was set by:

```
TRIBITS_ADD_LIBRARY(<libi> ... TESTONLY ...)
```

Therefore, to link to a defined `TESTONLY` library in any upstream enabled package, one just needs to pass in the library name through `DEPLIBS ... <libi> ...` and that is it!

`COMM [serial] [mpi]`

If specified, selects if the test will be added in serial and/or MPI mode. See the `COMM` argument in the script TRIBITS_ADD_TEST() for more details.

`LINKER_LANGUAGE (C|CXX|Fortran)`

If specified, overrides the linker language used by setting the target property `LINKER_LANGUAGE`. By default, CMake choses the compiler to be used as the linker based on file extensions. The most typical use case is when Fortran-only or C-only sources are passed in through `SOURCES` but a C++ linker is needed because there are upstream C++ libraries.

`DEFINES -D<define0> -D<define1> ...`

Add the listed defines using `ADD_DEFINITIONS()`. These should only affect the listed sources for the built executable and not other compiles in this directory due to the FUNCTION scoping.

`INSTALLABLE`

If passed in, then an install target will be added to install the built exectuable into the `${CMAKE_INSTALL_PREFIX}/bin/` directory (see Install Target (TRIBITS_ADD_EXECUTABLE())).

Executable and Target Name (TRIBITS_ADD_EXECUTABLE())

By default, the full name of the executable and target name `<fullExecName>` =

```
${PACKAGE_NAME}_<exeRootName>
```

If `ADD_DIR_TO_NAME` is set, then the directory path relative to the package base directory (with "/" replaced with "_"), or `<relDirName>`, is added to the executable name to form `<fullExecName>` =

```
${PACKAGE_NAME}_<relDirName>_<exeRootName>
```

If the option NOEXEPREFIX is pased in, the prefix ${PACKAGE_NAME}_ is removed.

CMake will add the executable suffix ${${PROJECT_NAME}_CMAKE_EXECUTABLE_SUFFIX} the actual executable file if the option NOEXESUFFIX is not passed in but this suffix is never added to the target name.

The reason that a default prefix is prepended to the executable and target name is because the primary reason to create an executable is typically to create a test or an example that is private to the package. This prefix helps to namespace the exexutable and its target so as to avoid name clashes with targets in other packages. It also helps to avoid clashes if the executable gets installed into the install directory (if INSTALLABLE is specified).

Additional Executable and Source File Properties (TRIBITS_ADD_EXECUTABLE())

Once ADD_EXECUTABLE(<fullExeName> ...) is called, one can set and change properties on the <fullExeName> executable target using SET_TARGET_PROPERTIES() as well as properties on any of the source files listed in SOURCES using SET_SOURCE_FILE_PROPERTIES() just like in any CMake project.

Install Target (TRIBITS_ADD_EXECUTABLE())

If INSTALLABLE is passed in, then an install target INSTALL(TARGETS <fullExeName> ...) is added to install the built executable into the ${CMAKE_INSTALL_PREFIX}/bin/ directory (actual install directory path is determined by ${PROJECT_NAME}_INSTALL_RUNTIME_DIR) .

TRIBITS_ADD_EXECUTABLE_AND_TEST()

Add an executable and a test (or several tests) all in one shot.

Usage:

```
TRIBITS_ADD_EXECUTABLE_AND_TEST(
  <exeRootName>  [NOEXEPREFIX]  [NOEXESUFFIX]  [ADD_DIR_TO_NAME]
  SOURCES <src0> <src1> ...
  [NAME <testName> | NAME_POSTFIX <testNamePostfix>]
  [CATEGORIES <category0>  <category1> ...]
  [HOST <host0> <host1> ...]
  [XHOST <xhost0> <xhost1> ...]
  [XHOST_TEST <xhost0> <xhost1> ...]
  [HOSTTYPE <hosttype0> <hosttype1> ...]
  [XHOSTTYPE <xhosttype0> <xhosttype1> ...]
  [XHOSTTYPE_TEST <xhosttype0> <xhosttype1> ...]
  [DIRECTORY <dir>]
  [DEFINES -DS<someDefine>]
  [DEPLIBS <lib0> <lib1> ... ]
  [COMM [serial] [mpi]]
  [ARGS "<arg0> <arg1> ..." "<arg2> <arg3> ..." ...]
  [NUM_MPI_PROCS <numProcs>]
  [LINKER_LANGUAGE (C|CXX|Fortran)]
  [STANDARD_PASS_OUTPUT
    | PASS_REGULAR_EXPRESSION "<regex0>;<regex1>;..."]
  [FAIL_REGULAR_EXPRESSION "<regex0>;<regex1>;..."]
  [WILL_FAIL]
  [ENVIRONMENT <var0>=<value0> <var1>=<value1> ...]
  [INSTALLABLE]
  [TIMEOUT <maxSeconds>]
  )
```

This function takes a fairly common set of arguments to TRIBITS_ADD_EXECUTABLE() and TRIBITS_ADD_TEST() but not the full set passed to TRIBITS_ADD_TEST(). See the documentation for TRIBITS_ADD_EXECUTABLE() and TRIBITS_ADD_TEST() to see which arguments are accpeted by which functions.

Arguments that are specific to this function and not contained in TRIBITS_ADD_EXECUTABLE() or TRIBITS_ADD_TEST() include:

```
XHOST_TEST <xhost0> <xhost1> ...
```

When specified, this disables just running the tests for the named hosts `<xhost0>`, `<xhost0>` etc. but still builds the executable for the test.

`XHOSTTYPE_TEST <xhosttype0> <hosttype1> ...`

When specified, this disables just running the tests for the named host types `<hosttype0>`, `<hosttype0>`, ..., but still builds the executable for the test.

This is the function to use for simple test executbles that you want to run that either takes no arguments or just a simple set of arguments passed in through `ARGS`.

TRIBITS_ADD_LIBRARY()

Function used to add a CMake library and target using `ADD_LIBRARY()`.

Usage:

```
TRIBITS_ADD_LIBRARY(
  <libName>
  [HEADERS <h0> <h1> ...]
  [NOINSTALLHEADERS <nih0> <hih1> ...]
  [SOURCES <src0> <src1> ...]
  [DEPLIBS <deplib0> <deplib1> ...]
  [IMPORTEDLIBS <ideplib0> <ideplib1> ...]
  [TESTONLY]
  [NO_INSTALL_LIB_OR_HEADERS]
  [CUDALIBRARY]
  )
```

Sections:

Formal Arguments (TRIBITS_ADD_LIBRARY())

`<libName>`

Required name of the library. This is the name passed to `ADD_LIBRARY(<libName> ...)`. The name is *not* prefixed by the packae name. CMake will of course add any standard prefix or post-fix to the library file name appropriate for the platform and if this is a static or shared library build.

`HEADERS <h0> <h1> ...`

List of public header files for using this library. By default, these header files are assumed to be in the current source directory. They can also contain the relative path or absolute path to the files if they are not in the current source directory. List list of headers is passed into `ADD_LIBRARY(...)` as well (which is not strictly needed but is helpful for some build tools, like MS Visual Stuido). By default, these headers will be installed as well (see Include Directories (TRIBITS_ADD_LIBRARY())).

`NOINSTALLHEADERS <nih0> <hih1> ...`

List of private header files which are used by this library. These headers are not installed and do not needed to be passed in for any purpose other than to pass them into `ADD_LIBRARY()` as some build tools like to have these listed (e.g. MS Visual Studio).

`SOURCES <src0> <src1> ...`

List of source files passed into `ADD_LIBRARY()` that are compiled into header files and included in the library. The compiler used to compile the files is determined automatically based on the file extension (see CMake documentation).

`DEPLIBS <deplib0> <deplib1> ...`

List of dependent libraries that are built in the current SE package that this library is dependent on. These libraries are passed into `TARGET_LINK_LIBRARIES(<libName> ...)` so that CMake knows about the dependency. You should **not** list libraries in other upstream SE packages or libraries built externally from this TriBITS CMake project. The TriBITS system automatically handles linking to libraries in uptream TriBITS packages and external libraries need to be listed in `IMPORTEDLIBS` instead.

`IMPORTEDLIBS <ideplib0> <ideplib1> ...`

List of dependent libraries built exteranlly from this TriBITS CMake project. These libraries are passed into `TARGET_LINK_LIBRARIES(<libName> ...)` so that CMake knows about the dependency. These libraries are added the `${PACKAGE_NAME}_LIBRARIES` so that downstream SE packages will also have these libraries and the link line also and these libraries will show up in the generated `Makefile.export.${PACKAGE_NAME}` and `${PACKAGE_NAME}Config.cmake` files if they are generated.

`TESTONLY`

If passed in, then `<libName>` will **not** be added to `${PACKAGE_NAME}_LIBRARIES` and an install target for the library will not be added. In this case, the current include directories will be set in the global variable `<libName>_INCLUDE_DIR` which will be used in TRIBITS_ADD_EXECUTABLE() when a test-only library is linked in.

`NO_INSTALL_LIB_OR_HEADERS`

If specified, then no install targets will be added for the library `<libName>` or the header files listed in `HEADERS`.

`CUDALIBRARY`

If specified then `CUDA_ADD_LIBRARY()` is used instead of `ADD_LIBRARY()` where `CUDA_ADD_LIBRARY()` is assumed to be defined by the standard FindCUDA.cmake module as processed using the standard TriBITS FindTPLCUDA.cmake file. For this option to work, this SE package must have an enabled direct or indirect dependency on the TriBITS CUDA TPL or a configure-time error will occur about not finding `CUDA_ALL_LIBRARY()`.

Include Directories (TRIBITS_ADD_LIBRARY())

Any base directories for these header files listed in `HEADERS` or `NOINSTALLHEADERS` should be passed into `INCLUDE_DIRECTORIES()` *before* calling this function. These include directories will then be added to current packages list of include directories `${PACKAGE_NAME}_INCLUDE_DIRS`.

Install Targets (TRIBITS_ADD_LIBRARY())

By default, an install target for the library is created using `INSTALL(TARGETS <libName> ...)` to install into the directory `${CMAKE_INSTALL_PREFIX}/lib/` (actual install directory is given by `${PROJECT}_INSTALL_LIB_DIR`). However, this install target will not get created if `${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS=FALSE` and `BUILD_SHARD_LIBS=OFF`. But when `BUILD_SHARD_LIBS=ON`, the install target will get created. Also, this install target will *not* get created if `TESTONLY` or `NO_INSTALL_LIB_OR_HEADERS` are passed in.

By default, an install target for the headers listed in `HEADERS` will get created using `INSTALL(FILES <h1> <h2> ...)`, but only if `TESTONLY` and `NO_INSTALL_LIB_OR_HEADERS` are not passed in as well. These headers get installed into the flat directory `${CMAKE_INSTALL_PREFIX}/include/` (the actual install directory is given by `${PROJECT_NAME}_INSTALL_INCLUDE_DIR`). Note that an install target will *not* get created for the headers listed in `NOINSTALLHEADERS`.

Additional Library and Source File Properties (TRIBITS_ADD_LIBRARY())

Once ADD_LIBRARY(<libName> ... <src0> <src1> ...) is called, one can set and change properties on the <libName> library target using SET_TARGET_PROPERTIES() as well as properties on any of the source files listed in SOURCES using SET_SOURCE_FILE_PROPERTIES() just like in any CMake project.

Miscellaneous Notes (TRIBITS_ADD_LIBRARY())

WARNING: Do **NOT** use ADD_DEFINITIONS() to add defines -D<someDefine> to the compile command line that will affect a header file! These defines are only set locally in this directory and child directories. These defines will **NOT** be set when code in peer directories (e.g. a downstream TriBIS pacakge) compiles code that may include these header files. To add defines, please use a configured header file (see TRIBITS_CONFIGURE_FILE()).

TRIBITS_ADD_OPTION_AND_DEFINE()

Add an option and a define variable in one shot.

Usage:

```
TRIBITS_ADD_OPTION_AND_DEFINE( <userOptionName>  <macroDefineName>
  "<docStr>"  <defaultValue> )
```

This macro sets the user cache BOOL variable <userOptionName> and if it is true, then sets the global (internal cache) macro define variable <macroDefineName> to ON, and otherwise sets it to OFF. This is designed to make it easy to add a user-enabled option to a configured header file and have the define set in one shot. This would require that the package's configure file (see TRIBITS_CONFIGURE_FILE()) have the line:

```
#cmakedefine <macroDefineName>
```

TRIBITS_ADD_SHOW_DEPRECATED_WARNINGS_OPTION()

Add the standard option ${PACKAGE_NAME}_SHOW_DEPRECATED_WARNINGS for the package.

Usage:

```
TRIBITS_ADD_SHOW_DEPRECATED_WARNINGS_OPTION()
```

This option is given the default"${${PROJECT_NAME}_SHOW_DEPRECATED_WARNINGS}". This option is then looked for in TRIBITS_CONFIGURE_FILE() to add macros to add deprecated warnings to deprecated parts of a package.

TRIBITS_ADD_TEST()

Add a test or a set of tests for a single executable or command.

Usage:

```
TRIBITS_ADD_TEST(
  <exeRootName>  [NOEXEPREFIX]  [NOEXESUFFIX]
  [NAME <testName> | NAME_POSTFIX <testNamePostfix>]
  [DIRECTORY <directory>]
  [ADD_DIR_TO_NAME]
  [ARGS "<arg0> <arg1> ..." "<arg2> <arg3> ..." ...
    | POSTFIX_AND_ARGS_0 <postfix0> <arg0> <arg1> ...
      POSTFIX_AND_ARGS_1 ... ]
  [COMM [serial] [mpi]]
  [NUM_MPI_PROCS <numProcs>]
  [CATEGORIES <category0>  <category1> ...]
  [HOST <host0> <host1> ...]
  [XHOST <host0> <host1> ...]
  [HOSTTYPE <hosttype0> <hosttype1> ...]
  [XHOSTTYPE <hosttype0> <hosttype1> ...]
  [STANDARD_PASS_OUTPUT
    | PASS_REGULAR_EXPRESSION "<regex0>;<regex1>;..."]
```

```
  [FAIL_REGULAR_EXPRESSION "<regex0>;<regex1>;..."]
  [WILL_FAIL]
  [ENVIRONMENT <var0>=<value0> <var1>=<value1> ...]
  [TIMEOUT <maxSeconds>]
  )
```

Sections:

Formal Arguments (TRIBITS_ADD_TEST())

`<exeRootName>`

The name of the exectuble or path to the exectuable to run for the test (see Determining the Exectuable or Command to Run (TRIBITS_ADD_TEST())). This name is also the default root name for the test (see Determining the Full Test Name (TRIBITS_ADD_TEST())).

`NOEXEPREFIX`

If specified, then the prefix `${PACKAGE_NAME}_` is not assumed to be prepended to `<exeRootName>`.

`NOEXESUFFIX`

If specified, then the postfix `${${PROJECT_NAME}_CMAKE_EXECUTABLE_SUFFIX}` is not assumed to be post-pended to `<exeRootName>`.

`NAME <testRootName>`

If specified, gives the root name of the test. If not specified, then `<testRootName>` is taken to be `<exeRootName>`. The actual test name will always prefixed as `${PACKAGE_NAME}_<testRootName>` passed into the call to the built-in CMake command `ADD_TEST(...)`. The main purpose of this argument is to allow multiple tests to be defined for the same executable. CTest requires all test names to be globally unique in a single project.

`NAME_POSTFIX <testNamePostfix>`

If specified, gives a postfix that will be added to the standard test name based on `<exeRootName>` (appended as `_<NAME_POSTFIX>`). If the `NAME <testRootName>` argument is given, this argument is ignored.

`DIRECTORY <dir>`

If specified, then the executable is assumed to be in the directory given by by `<dir>`. The directory `<dir>` can either be a relative or absolute path. If not specified, the executable is assumed to be in the current bindary directory.

`ADD_DIR_TO_NAME`

If specified, then the directory name that this test resides in will be added into the name of the test after the package name is added and before the root test name (see below). The directory will have the package's base directory stripped off so only the unique part of the test directory will be used. All directory seperators will be changed into underscores.

`RUN_SERIAL`

If specified then no other tests will be allowed to run while this test is running. This is useful for devices(like cuda cards) that require exclusive access for processes/threads. This just sets the CTest test property `RUN_SERIAL` using the built-in CMake function `SET_TESTS_PROPERTIES()`.

`ARGS "<arg0> <arg1> ..." "<arg2> <arg3> ..." ...`

If specified, then a set of arguments can be passed in quotes. If multiple groups of arguments are passed in different quoted clusters of arguments then a different test will be added for each set of arguments. In this way, many different tests can be added for a single executable in a single call to this function. Each of these separate tests will be named `${TEST_NAME}_xy` where `xy` = `00`, `01`, `02`, and so on. **WARNING:** When defining multiple tests it is prefered to use the `POSTFIX_AND_ARGS_<IDX>` form instead. **WARNING:** Multiple arguments passed to a single test invocation must be quoted or multiple tests taking single arguments will be created instead! See Adding Multiple Tests (TRIBITS_ADD_TEST()) for more details and exmaples.

`POSTFIX_AND_ARGS_<IDX> <postfix> <arg0> <arg1> ...`

If specified, gives a sequence of sets of test postfix names and arguments lists for different tests (up to `POSTFIX_AND_ARGS_19`). For example, a set of three different tests with argument lists can be specified as:

```
POSTIFX_AND_ARGS_0 postfix0 --arg1 --arg2="dummy"
POSTIFX_AND_ARGS_1 postfix1  --arg2="fly"
POSTIFX_AND_ARGS_2 postfix2  --arg2="bags"
```

This will create three different test cases with the postfix names `postfix0`, `postfix1`, and `postfix2`. The indexes must be consecutive starting a `0` and going up to (currently) `19`. The main advantages of using these arguments instead of just 'ARGS' are that you can give meaningful name to each test case and you can specify multiple arguments without having to quote them and you can allow long argument lists to span multiple lines. See Adding Multiple Tests (TRIBITS_ADD_TEST()) for more details and exmaples.

`COMM [serial] [mpi]`

If specified, selects if the test will be added in serial and/or MPI mode. If the `COMM` argument is missing, the test will be added in both serial and MPI builds of the code.

`NUM_MPI_PROCS <numProcs>`

If specified, gives the number of processes that the test will be defined to run. If `<numProcs>` is greater than `${MPI_EXEC_MAX_NUMPROCS}` then the test will be excluded. If not specified, then the default number of processes for an MPI build will be `${MPI_EXEC_DEFAULT_NUMPROCS}`. For serial builds, this argument is ignored.

`HOST <host0> <host1> ...`

If specified, gives a list of hostnames where the test will be included. The current hostname is determined by the built-in CMake command `SITE_NAME(${PROJECT_NAME}_HOSTNAME)`. On Linux/Unix systems, this is typically the value returned by 'uname -n'. If this list is given, the value of `${${PROJECT_NAME}_HOSTNAME}` must equal one of the listed host names `<hosti>` or test will not be added. The value of `${PROJECT_NAME}_HOSTNAME` gets printed out in the TriBITS cmake output under the section `Probing the environment`.

`XHOST <host0> <host1> ...`

If specified, gives a list of hostnames (see `HOST` argument) where the test will *not* be added. This check is performed after the check for the hostnames in the `HOST` list if it should exist. Therefore, this list exclusion list overrides the 'HOST' inclusion list.

`CATEGORIES <category0> <category1> ...`

If specified, gives the specific categories of the test. Valid test categories include BASIC, CONTINUOUS, NIGHTLY, WEEKLY and PERFORMANCE. By default, the category is BASIC. When the test category does not match ${PROJECT_NAME}_TEST_CATEGORIES, then the test is not added. When the CATEGORIES is BASIC it will match ${PROJECT_NAME}_TEST_CATEGORIES eqaual to CONTINUOUS, NIGHTLY, and WEEKLY. When the CATEGORIES contains CONTINUOUS it will match ${PROJECT_NAME}_TEST_CATEGORIES equal to CONTINUOUS, NIGHTLY, and WEEKLY. When the CATEGORIES is NIGHTLY it will match ${PROJECT_NAME}_TEST_CATEGORIES equal to NIGHTLY and WEEKLY. When the CATEGORIES is PERFORMANCE it will match ${PROJECT_NAME}_TEST_CATEGORIES=PERFORMANCE only.

HOSTTYPE <hosttype0> <hosttype1> ...

If specified, gives the names of the host system type (given by CMAKE_HOST_SYSTEM_NAME which is printed in the TriBITS cmake confgiure output in the section Probing the environment) to include the test. Typical host system type names include Linux, Darwain, Windows, etc.

XHOSTTYPE <hosttype0> <hosttype1> ...

If specified, gives the names of the host system type to *not* include the test. This check is performed after the check for the host system names in the HOSTTYPE list if it should exist. Therefore, this list exclusion list overrides the HOSTTYPE inclusion list.

STANDARD_PASS_OUTPUT

If specified, then the standard test output End Result: TEST PASSED is greped for to determine success. This is needed for MPI tests on some platforms since the return value is unreliable. This is set using the built-in ctest property PASS_REGULAR_EXPRESSION.

PASS_REGULAR_EXPRESSION "<regex0>;<regex1>;..."

If specified, then a test will be assumed to pass only if one of the regular expressions <regex0>, <regex1> etc. match the output. Otherwise, the test will fail. This is set using the built-in test property PASS_REGULAR_EXPRESSION. Consult standard CMake documentation.

FAIL_REGULAR_EXPRESSION "<regex0>;<regex1>;..."

If specified, then a test will be assumed to fail if one of the regular expressions <regex0>, <regex1> etc. match the output. Otherwise, the test will pass. This is set using the built-in test property FAIL_REGULAR_EXPRESSION.

WILL_FAIL

If passed in, then the pass/fail criteria will be inverted. This is set using the built-in test property WILL_FAIL.

ENVIRONMENT <var0>=<value0> <var1>=<value1> ...

If passed in, the listed environment varaibles will be set before calling the test. This is set using the built-in test property ENVIRONMENT.

TIMEOUT <maxSeconds>

If passed in, gives maximum number of seconds the test will be allowed to run beforebeing timed-out. This sets the test property TIMEOUT. **WARNING:** Rather than just increasing the timeout for an expensive test, please try to either make the test run faster or relegate the test to being run less often (i.e. set CATEGORIES NIGHTLY or even WEEKLY for extremently expensive tests). Expensive tests are one of the worse forms of technical debt that a project can have!

In the end, this function just calls the built-in CMake commands ADD_TEST(${TEST_NAME} ...) and SET_TESTS_PROPERTIES(${TEST_NAME} ...) to set up a executable process for ctest to run, determine pass/fail criteria, and set some other test properties. Therefore, this wrapper funtion does not provide any fundamentally new features that are already avaiable in the basic usage if CMake/CTest. However, this wrapper function takes care of

many of the details and boiler-plate CMake code that it takes to add such a test (or tests) and enforces consistency across a large project for how tests are defined, run, and named (to avoid test name clashes).

If more flexibility or control is needed when defining tests, then the function `TRIBITS_ADD_ADVANCED_TEST()` should be used instead.

In the following subsections, more details on how tests are defined and run is given.

Determining the Exectuable or Command to Run (TRIBITS_ADD_TEST())

This funtion is primarily designed to make it easy to run tests for exectaubles built using the function TRIBITS_ADD_EXECUTABLE(). To set up tests to run arbitrary executables, see below.

By default, the command to run for the executable is determined by first getting the exectuable name which by default is assumed to be `<fullExeName>` =

```
${PACKAGE_NAME}_<exeRootName>${${PROJECT_NAME}_CMAKE_EXECUTABLE_SUFFIX}
```

which is (by no coincidence) idential to how it is selected in TRIBITS_ADD_EXECUTABLE(). This name can be alterned by passing in `NOEXEPREFIX`, `NOEXESUFFIX`, and `ADD_DIR_TO_NAME` as described in Executable and Target Name (TRIBITS_ADD_EXECUTABLE()).

By default, this executable is assumed to be in the current CMake binary directory `${CMAKE_CURRENT_BINARY_DIR}` but the directory location can be changed using the `DIRECTORY <dir>` argument.

If an arbitrary exectuable is to be run for the test, then pass in `NOEXEPREFIX` and `NOEXESUFFIX` and set `<exeRootName>` to the relative or absolute path of the exeutable to be run. If `<exeRootName>` is not an absolute path, then `${CMAKE_CURRENT_BINARY_DIR}/<exeRootName>` is set as the executable to run.

Whatever executable path is specified using this logic, if the executable is not found, then when `ctest` goes to run the test, it will mark it as `NOT RUN`.

Determining the Full Test Name (TRIBITS_ADD_TEST())

By default, the base test name is selected to be `<fullTestName>` =

```
${PACKAGE_NAME}_<exeRootName>
```

If `NAME <testRootName>` is passed in, then `<testRootName>` is used instead of `<exeRootName>`.

If `NAME_POSTFIX <testNamePostfix>` is passed in, then the base test name is selected to be `<fullTestName>` =

```
${PACKAGE_NAME}_<exeRootName>_<testNamePostfix>
```

If `ADD_DIR_TO_NAME` is passed in, then the directory name realtive to the package directory name is added to the name as well to help disambiguate the test name (see the above).

Let the test name determined by this process be `TEST_NAME`. If no arguments or one set of arguments are passed in through `ARGS`, then this is the test name actaully passed in to `ADD_TEST()`. If multiple tests are defined, then this name becomes the base test name for each of the tests. See below.

Finally, for any test that gets defined, if MPI is enabled (i.e. `TPL_ENABLE_MPI=ON`), then the terminal suffix *_MPI_${NUM_MPI_PROCS}* will be added to the end of the test name (even for multiple tests). No such prefix is added for the serial case (i.e. `TPL_ENABLE_MPI=OFF`).

Adding Multiple Tests (TRIBITS_ADD_TEST())

Using this function, one can add exectuable arguments and can even add multiple tests in one of two ways. One can either pass in 1 or more **quoted** clusters of arguments using:

```
ARGS "<arg0> <arg1> ..." "<arg2> <arg3> ..." ...
```

or can pass in an explicit test name postfix and arguments with:

```
POSTFIX_AND_ARGS_0 <postfix0> <arg0> <arg1> ...
POSTFIX_AND_ARGS_1 <postfix1> <arg2> ...
...
```

If only one short set of arguments needs to be passed in, then passing:

```
ARGS "<arg0> <arg1>"
```

may be preferable since it will not add any postfix name to the test. To add more than one test case using ARGS, you use more than one quoted set of arugments such as with:

```
ARGS "<arg0> <arg1>" "<arg2> <arg2>"
```

which creates 2 tests with the names <fullTestName>_00 passing arguments "<arg0> <arg1>" and <fullTestName>_01 passing arguments "<arg2> <arg3>". However, when passing multiple sets of arguments it is preferable to **not** use ARGS but instead use:

```
POSTFIX_AND_ARGS_0 test_a <arg0> <arg1>
POSTFIX_AND_ARGS_1 test_b <arg2> <arg2>
```

which also creates the same 2 tests but now with the improved names <fullTestName>_test_a passing arguments "<arg0> <arg1>" and <fullTestName>_test_b passing arguments "<arg2> <arg3>". In this way, the individual tests can be given more understandable names.

The other advantage of the POSTFIX_AND_ARGS_<IDX> form is that the arugments <arg0>, <arg1>, ... do not need to be quoted and can therefore be extended over multiple lines like:

```
POSTFOX_AND_ARGS_0 long_args --this-is-the-first-long-arg=very
  --this-is-the-second-long-arg=verylong
```

If you don't use quotes when using ARGS you actually get more than one test. For example, if you pass in:

```
ARGS --this-is-the-first-long-arg=very
  --this-is-the-second-long-arg=verylong
```

you actually get two tests, not one test. This is a common mistake that people make when using the ARGS form of passing arguments. This can't be fixed or it will break backward compatibility. If this could be designed fresh, the ARGS argument would only create a single test and the arguments would not be quoted.

Determining Pass/Fail (TRIBITS_ADD_TEST())

The only means to determine pass/fail is to use the built-in test properties PASS_REGULAR_EXPRESSION and FAIL_REGULAR_EXPRESSION which can only grep STDOUT/STDERR or to check for a 0 return value (or invert these using WILL_FAIL). For simple tests, that is enough. However, for more complex executables, one may need to examine the output files to determine pass fail. Raw CMake/CTest cant' do this. In this case, one should use TRIBITS_ADD_ADVANCED_TEST().

Setting additional test properties (TRIBITS_ADD_TEST())

After this function returns, any tests that get added using ADD_TEST() can have additional properties set and changed using SET_TEST_PROPERTIES(). Therefore, any tests properties that are not directly supported by this function and passed through this wrapper function can be set in the outer CMakeLists.txt file after the call to TRIBITS_ADD_TEST().

ToDo: Describe how to use new variable ADDED_TESTS_OUT to get the list of tests actually added (if they are added) in order to make it easy to set additional test properties.

Debugging and Examining Test Generation (TRIBITS_ADD_TEST())

In order to see what tests are getting added and to debug some issues in test creation, one can set the cache variable ${PROJECT_NAME}_VERBOSE_CONFIGURE=ON. This will result in the printout of some information about the test getting added or not.

Also, CMake writes a file CTestTestfile.cmake in the current binary directory which contains all of the added tests and test properties that are set. This is the file that is read by ctest when it runs to determine what tests to run. In that file, one can see the exact ADD_TEST() and SET_TEST_PROPERTIES() commands. The is the ultimate way to debug exactly what tests are getting added by this function.

Disabling Tests Externally (TRIBITS_ADD_TEST())

The test can be disabled externally by setting the CMake cache variable ${FULL_TEST_NAME}_DISABLE=TRUE. This allows tests to be disable on a case-by-case basis. This is the *exact* name that shows up in 'ctest -N' when running the test. If multiple tests are added in this funtion through multiple argument sets to ARGS or through multiple POSTFIX_AND_ARGS_<IDX> arguments, then ${FULL_TEST_NAME}_DISABLE=TRUE must be set for each test individually.

TRIBITS_ADD_TEST_DIRECTORIES()

Macro called to add a set of test directories for an SE package.

Usage:

```
TRIBITS_ADD_TEST_DIRECTORIES(<dir1> <dir2> ...)
```

This macro only needs to be called from the top most CMakeList.txt file for which all subdirectories are all "tests".

This macro can be called several times within a package and it will have the right effect.

Currently, really all it does macro does is to call `ADD_SUBDIRECTORY(<diri>)` if `${PACKAGE_NAME}_ENABLE_TESTS` or `${PARENT_PACKAGE_NAME}_ENABLE_TESTS` are true. However, this macro may be extended in the futgure in order to modify behavior related to adding tests and examples in a uniform way..

TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES()

Macro used in Dependencies.cmake files to allow some upstream dependent packages to be missing.

Usage:

```
TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES(<pack_1> <pack_2> ...)
```

If the missing upstream SE package <pack_i> is optional, then the effect will be to simply ignore the missing package and remove it from the dependency list. However, if the missing upstream SE package <pack_i> is required, then in addition to ignoring the missing package, the current SE (sub)package will also ee hard disabled, i.e. ${PROJECT_NAME}_ENABLE_{CURRENT_PACKAGE}=OFF.

This function is typically used in packages in external TriBITS repos that are depend on other packages in other exteral TriBITS repos that might be missing.

NOTE: Using this function effectively turns off error checking for misspelled package names so it is important to only use it when it absolutely is needed.

TRIBITS_CONFIGURE_FILE()

Macro that configures the package's main configured header file (typically called `${PACKAGE_NAME}_config.h` but any name can be used).

Usage:

```
TRIBITS_CONFIGURE_FILE(<packageConfigFile>)
```

This function requires the file:

```
${PACKAGE_SOURCE_DIR}/cmake/<packageConfigFile>.in
```

exists and it creates the file:

```
${CMAKE_CURRENT_BINARY_DIR}/<packageConfigFile>
```

by calling the built-in `CONFIGURE_FILE()` command:

```
CONFIGURE_FILE(
  ${PACKAGE_SOURCE_DIR}/cmake/<packageConfigFile>.in
  ${CMAKE_CURRENT_BINARY_DIR}/<packageConfigFile>
  )
```

which does basic sustitution of CMake variables (see documentation for built-in `CONFIGURE_FILE()` command for rules on how it performs substitutions).

In addition to just calling `CONFIGURE_FILE()`, this function also aids in creating configured header files adding macros for deprecating code.

Deprecated Code Macros

If `${PARENT_PACKAGE_NAME}_SHOW_DEPRECATED_WARNINGS` is `TRUE` (see TRIBITS_ADD_SHOW_DEPRECATED_WARNINGS_OPTION()), then the local CMake varible `${PARENT_PACKAGE_NAME_UC}_DEPRECATED_DECLARATIONS` adds a define `<PARENT_PACKAGE_NAME_UC>_DEPRECATED` (where `<PARENT_PACKAGE_NAME_UC>` is the package name in all upper-case letters) add the compiler-specific deprecated warning for an entity. To use this, just add the line:

```
@<PARENT_PACKAGE_NAME_UC>_DEPRECATED_DECLARATIONS@
```

to the <packageConfigFile>.in file and it will be expended.

Then C/C++ code can use this macro to deprecate functions, variables, classes, etc., for example, using:

```
<PARENT_PACKAGE_NAME_UC>_DEPRECATED class SomeDepreatedClass { ... }.
```

If the particular compiler does not support deprecated warnings, then this macro is defined to be empty. See Regulated Backward Compatibility and Deprecated Code for more details.

TRIBITS_COPY_FILES_TO_BINARY_DIR()

Function that copies a list of files from a soruce directory to a destination directory at configure time, typically so that it can be used in one or more tests. This sets up all of the custom CMake commands and targets to ensure that the files in the destiation directory are always up to date just by building the `ALL` target.

Usage:

```
TRIBITS_COPY_FILES_TO_BINARY_DIR(
  <targetName>
  [SOURCE_FILES <file1> <file2> ...]
  [SOURCE_DIR <sourceDir>]
  [DEST_FILES <dfile1> <dfile2> ...]
  [DEST_DIR <destDir>]
  [TARGETDEPS <targDep1> <targDep2> ...]
  [EXEDEPS <exeDep1> <exeDep2> ...]
  [NOEXEPREFIX]
  [CATEGORIES <category1>  <category2> ...]
  )
```

This function has a few valid calling modes:

1) Source files and destination files have the same name:

```
TRIBITS_COPY_FILES_TO_BINARY_DIR(
  <targetName>
  SOURCE_FILES <file1> <file2> ...
  [SOURCE_DIR <sourceDir>]
  [DEST_DIR <destDir>]
  [TARGETDEPS <targDep1> <targDep2> ...]
  [EXEDEPS <exeDep1> <exeDep2> ...]
  [NOEXEPREFIX]
  [CATEGORIES <category1>  <category2> ...]
  )
```

In this case, the names of the source files and the destination files are the same but just live in different directories.

2) Source files have a prefix different from the destination files:

```
TRIBITS_COPY_FILES_TO_BINARY_DIR(
  <targetName>
  DEST_FILES <file1> <file2> ...
  SOURCE_PREFIX <srcPrefix>
  [SOURCE_DIR <sourceDir>]
  [DEST_DIR <destDir>]
  [EXEDEPS <exeDep1> <exeDep2> ...]
  [NOEXEPREFIX]
  [CATEGORIES <category1>  <category2> ...]
  )
```

In this case, the source files have the same basic name as the destination files except they have the prefix 'srcPrefix' appended to the name.

3) Source files and destination files have completely different names:

```
TRIBITS_COPY_FILES_TO_BINARY_DIR(
  <targetName>
  SOURCE_FILES <sfile1> <sfile2> ...
  [SOURCE_DIR <sourceDir>]
  DEST_FILES <dfile1> <dfile2> ...
  [DEST_DIR <destDir>]
  [EXEDEPS <exeDep1> <exeDep2> ...]
  [NOEXEPREFIX]
  [CATEGORIES <category1>  <category2> ...]
  )
```

In this case, the source files and destination files have completely different prefixes.

The individual arguments are:

`SOURCE_FILES <file1> <file2> ...`

> Listing of the source files relative to the source directory given by the argument `SOURCE_DIR <sourceDir>`. If omited, this list will be the same as `DEST_FILES` with the argument `SOURCE_PREFIX <srcPrefix>` appended.

`SOURCE_DIR <sourceDir>`

> Optional argument that gives (absolute) the base directory for all of the source files. If omited, this takes the default value of `${CMAKE_CURRENT_SOURCE_DIR}`.

`DEST_FILES <file1> <file2> ...`

> Listing of the destination files relative to the destination directory given by the argument `DEST_DIR <destDir>` If omited, this list will be the same as given by the `SOURCE_FILES` list.

`DEST_DIR <destDir>`

> Optional argument that gives the (absolute) base directory for all of the destination files. If omited, this takes the default value of `${CMAKE_CURRENT_BINARY_DIR}`

`TARGETDEPS <targDep1> <targDep2> ...`

> Listing of general CMake targets that these files will be added as dependencies to.

`EXEDEPS <exeDep1> <exeDep2> ...`

> Listing of executable targets that these files will be added as dependencies to. By default the prefix `${PACKAGE_NAME}_` will is appended to the names of the targets. This ensures that if the executable target is built that these files will also be copied as well.

`NOEXEPREFIX`

> Option that determines if the prefix `${PACKAGE_NAME}_` will be appended to the arguments in the `EXEDEPS` list.

TRIBITS_CTEST_DRIVER()

Platform-independent package-by-package CTest/CDash driver (run by `ctest` **NOT** `cmake`).

Usage:

```
TRIBITS_CTEST_DRIVER()
```

This is the driver code that is platform independent. This script drives the testing process by doing an update and then configuring and building the top-level TriBITS packages one at a time. This function gets called from inside of a platform and build-specific `ctest -S` driver script.

To understand this script, one must understand that it gets run in several different modes:

Mode 1: Run where there are already existing source and binary directories (i.e. `CTEST_DASHBOARD_ROOT` is set empty before call). This is for when the ctest driver script is run on an existing source and binary tree. In this case, there is one project source tree and `CTEST_SOURCE_DIRECTORY` and `CTEST_BINARY_DIRECTORY` must be set by the user before calling this function.

Mode 2: A new binary directory is created and new sources are cloned (or updated) in a driver directory (`CTEST_DASHBOARD_ROOT` is set is *not* empty before call). In this case, there are always two (partial) project source tree's, i) a "driver" skeleton source tree (typically embedded with TriBITS directory) that bootstraps the testing process, and ii) a true full "source" that is (optionally) cloned and/or updated.

There are a few different directory locations are significant for this script:

`TRIBITS_PROJECT_ROOT`

> **The root directory to an existing source tree where the project's** ProjectName.cmake (defining PROJECT_NAME variable) and Version.cmake file's can be found.

`${PROJECT_NAME}_TRIBITS_DIR`

> The base directory for the TriBITS system's various CMake modules, python scripts, and other files. By default this is assumed to be in the source tree under ${TRIBITS_PROJECT_ROOT} (see below) but it can be overridden to point to any location.

`CTEST_DASHBOARD_ROOT`

> If set, this is the base directory where this script runs that clones the sources for the project. If this directory does not exist, it will be created. If empty, then has no effect on the script.

`CTEST_SOURCE_DIRECTORY`

> Determines the location of the sources that are used to define packages, dependencies and configure and build the software. This is a varaible that CTest directly reads and must therefore be set. This is used to set PROJECT_SOURCE_DIR which is used by the TriBITS system. If CTEST_DASHBOARD_ROOT is set, then this is hard-coded to ${CTEST_DASHBOARD_ROOT}/${CTEST_SOURCE_NAME}.

`CTEST_BINARY_DIRECTORY`

> Determines the location of the binary tree where output from CMake/CTest is put. This is used to set to PROJECT_BINARY_DIR which is used by the TriBITS system. If CTEST_DASHBOARD_ROOT is set, then this is hard-coded to ${CTEST_DASHBOARD_ROOT}/BUILD.

ToDo: Document input variables that have defaults, be be set before, and can be overridden from the env.

ToDo: Finish Documentation!

TRIBITS_DEFINE_PACKAGE_DEPENDENCIES()

Define the dependenices for a given TriBITS SE package (i.e. a top-level package or a subpackage) in the package's <packageDir>/cmake/Dependencies.cmake file.

Usage:

```
TRIBITS_DEFINE_PACKAGE_DEPENDENCIES(
   [LIB_REQUIRED_PACKAGES <pkg1> <pkg2> ...]
   [LIB_OPTIONAL_PACKAGES <pkg1> <pkg2> ...]
   [TEST_REQUIRED_PACKAGES <pkg1> <pkg2> ...]
   [TEST_OPTIONAL_PACKAGES <pkg1> <pkg2> ...]
   [LIB_REQUIRED_TPLS <tpl1> <tpl2> ...]
   [LIB_OPTIONAL_TPLS <tpl1> <tpl2> ...]
   [TEST_REQUIRED_TPLS <tpl1> <tpl2> ...]
   [TEST_OPTIONAL_TPLS <tpl1> <tpl2> ...]
   [REGRESSION_EMAIL_LIST  <regression-email-address>
   [SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS
     <spkg1_name>  <spkg1_dir>  <spkg1_classifications>  <spkg1_optreq>
     <spkg2_name>  <spkg2_dir>  <spkg2_classifications>  <spkg2_optreq>
     ...
     ]
   )
```

Every argument in this macro is optional. The arguments that apply a package itself are:

- **LIB_REQUIRED_PACKAGES:** List of upstream packages that must be enabled in order to build and use the libraries (or capabilities) in this package.
- **LIB_OPTIONAL_PACKAGES:** List of additional optional upstream packages that can be used in this package if enabled. These upstream packages need not be enabled in order to use this package but not enabling one or more of these optional upstream packages will result in diminished capabilities of this package.
- **TEST_REQUIRED_PACKAGES:** List of additional upstream packages that must be enabled in order to build and/or run the tests and/or examples in this packages. If any of these upstream packages is not enabled, then there will be no tests or examples defined or run for this package.
- **TEST_OPTIONAL_PACKAGES:** List of additional optional upstream packages that can be used by the tests in this package. These upstream packages need not be enabled in order to run basic tests for this package. Typically, extra tests that depend on optional test packages involve integration testing of some type.
- **LIB_REQUIRED_TPLS:** List of upstream TPLs that must be enabled in order to build and use the libraries (or capabilities) in this package.
- **LIB_OPTIONAL_TPLS:** List of additional optional upstream TPLs that can be used in this package if enabled. These upstream TPLs need not be enabled in order to use this package but not enabling one or more of these optional upstream TPLs will result in diminished capabilities of this package.
- **TEST_REQUIRED_TPLS:** List of additional upstream TPLs that must be enabled in order to build and/or run the tests and/or examples in this packages. If any of these upstream TPLs is not enabled, then there will be no tests or examples defined or run for this package.
- **TEST_OPTIONAL_TPLS:** List of additional optional upstream TPLs that can be used by the tests in this package. These upstream TPLs need not be enabled in order to run basic tests for this package. Typically, extra tests that depend on optional test TPLs involve integration testing of some type.

Only upstream SE packages can be listed (as defined by the order the packages are listed in TRIBITS_DEFINE_REPOSITORY_PACKAGES() in the <repoDir>/PackagesList.cmake file). Otherwise an error will occur and processing will stop. Also, mispelled SE package names are caught as well.

Only direct package dependenices need to be listed. Indirect package dependencies are automatically handled. For example, if this SE package directly depends on PKG2 which depends on PKG1 (but this SE package does not directly depend on anything in PKG1) then this package only needs to list a dependency on PKG2, not PKG1. The dependnecy on PKG1 will be taken care of automatically by the TriBITS dependency tracking system.

However, currently, all TPL dependendies must be listed, even the indirect ones. This is a requirement that will be dropped in a future version of TriBITS.

The packages listed in LIB_REQUIRED_PACKAGES are implicitly also dependenices in TEST_REQUIRED_PACKAGES. Likewise LIB_OPTIONAL_PACKAGES are implicitly also dependenices in TEST_OPTIONAL_PACKAGES. Same goes for TPL dependencies.

The upstream dependencies within a single list do not need to be listed in any order. For example if `PKG2` depends on `PKG1`, and this given SE package depends on both, one can list:

```
LIB_REQUIRED_PACKAGES PKG2 PKG1
```

or:

```
"LIB_REQUIRED_PACKAGES PKG1 PKG2
```

Likewise the listing of TPLs order is not important.

If some upstream packages are allowed to be missing, this can be specified by calling the macro TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES().

A top-level package can also have subpackages. In this case, the following varible must be set:

- **SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS:** 2D array with rows listing the subpackages and the columns:
  - **SUBPACKAGE:** The name of the subpackage <spkg_name>. The full SE package name is "${PARENT_PACKAGE_NAME}<spkg_name>". The full SE package name is what is used in listing dependenices in other SE packages.
  - **DIRS:** The subdirectory <spkg_dir> relative to the parent package's base directory. All of the contents of the subpackage should be under this subdirectory. This is assumed by the TriBITS testing support software when mapping modified files to SE packages that need to be tested.
  - **CLASSIFICATIONS**: The test group PT, ST, EX and the maturity level EP, RS, PG, PM, GRS, GPG, GPM, and UM, separated by a coma ',' with no spaces in between (e.g. "PT,GPM"). These have exactly the name meaning as for full packages (see TRIBITS_DEFINE_REPOSITORY_PACKAGES()).
  - **OPTREQ:** Determines if the outer parent package has an OPTIONAL or REQUIRED dependence on this subpackage.

Other variables that this macro handles:

- **REGRESSION_EMAIL_LIST:** The email list that is used to send CDash error messages. If this is missing, then the email list that CDash errors go to is determined by other means (see CDash regression email addresses).

NOTE: All this macro really does is to just define the variables:

- LIB_REQUIRED_DEP_PACKAGES
- LIB_OPTIONAL_DEP_PACKAGES
- TEST_REQUIRED_DEP_PACKAGES
- TEST_OPTIONAL_DEP_PACKAGES
- LIB_REQUIRED_DEP_TPLS
- LIB_OPTIONAL_DEP_TPLS
- TEST_REQUIRED_DEP_TPLS
- TEST_OPTIONAL_DEP_TPLS
- REGRESSION_EMAIL_LIST
- SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS

which are then read by the TriBITS cmake code to build the package dependency graph. The advantage of using this macro instead of just directly setting the varibles is that you only need to list the dependencies you have. Otherwise, you need to set all of these varibles, even those that are empty. This is a error checking property of the TriBITS system to avoid misspelling the names of these variables.

TRIBITS_DEFINE_REPOSITORY_PACKAGES()

Define the set of packages for a given TriBIT repo. This macro is typically called from inside of a <repoDir>/PackagesList.cmake file for a given TriBITS repo.

Usage:

```
TRIBITS_DEFINE_REPOSITORY_PACKAGES(
   <pkg0>  <pkg0_dir>  <pkg0_classif>
   <pkg1>  <pkg1_dir>  <pkg1_classif>
   ...
   )
```

This macro sets up a 2D array of NumPackages by NumColumns listing out the packages for a TriBITS repository. Each row (with 3 entries) specifies a package which contains the 3 columns (ordered 0-2):

0. **PACKAGE** (`<pkgi>`): The name of the TriBITS package. This name must be unique across all other TriBITS packages in this or any other TriBITS repo that might be combined into a single TriBITS project meta-build. The name should be a valid identifier (e.g. matches the regex `[a-zA-Z_][a-zA-Z0-9_]*`).
1. **DIR** (`<pkgi_dir>`)): The relative directory for the package. This is relative to the TriBITS repository base directory. Under this directory will be a package-specific 'cmake/' directory with file 'cmake/Dependencies.cmake' and a base-level CMakeLists.txt file. The entire contents of the package including all of the source code and all of the tests should be contained under this directory. The TriBITS testing infrastructure relies on the mapping of changed files to these base directories when deciding what packages are modified and need to be retested (along with downstream packages).
2. **CLASSIFICATION** (`<pkgi_classif>`)): Gives the testing group PT, ST, EX and the maturity level EP, RS, PG, PM, GRS, GPG, GPM, UM. These are seprated by a coma with no space in between such as "RS,PT" for a "Research Stable", "Primary Tested" package. No spaces are allowed so that CMake treats this a one field in the array. The maturity level can be left off in which case it is assumed to be UM for "Unspecified Maturity". This classification for individual packages can be changed to `EX` for specific platforms by calling TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS().

IMPORTANT: The packages must be listed in increasing order of package dependencies; there are no cyclic package dependencies allowed. That is, package `i` can only list dependencies (in <packageDir>/cmake/Dependencies.cmake) for packages listed before this package in this list (or in upstream TriBITS repositories). This avoids an expensive package sorting algorithm and makes it easy to flag packages with circular dependencies or misspelling of package names.

NOTE: This macro just sets the varaible:

```
${REPOSITORY_NAME}_PACKAGES_AND_DIRS_AND_CLASSIFICATIONS
```

in the current scope. The advantages of using this macro instead of directly setting this varible include:

- Asserts that the varible `REPOSITORY_NAME` is defined and set
- Avoids having to hard-code the assumed repository name `${REPOSITORY_NAME}`. This provides more flexibility for how other TriBITS project name a given TriBITS repo (i.e. the name of repo subdirs).
- Avoid mispelling the name of the varible `${REPOSITORY_NAME}_PACKAGES_AND_DIRS_AND_CLASSIFICATIONS`. If you misspell the name of the macro, it is an immediate error in CMake.

TRIBITS_DEFINE_REPOSITORY_TPLS()

Define the list of TPLs, find modules, and classifications for a given TriBITS repository. This macro is typically called from inside of a TPLsList.cmake fil for a given TriBITS repo.

Usage:

```
TRIBITS_DEFINE_REPOSITORY_TPLS(
  <tpl0_name>   <tpl0_findmod>  <tpl0_classif>
  <tpl1_name>   <tpl1_findmod>  <tpl1_classif>
  ...
  )
```

This macro sets up a 2D array of NumTPLS by NumColumns listing out the TPLs for a TriBITS repository. Each row (with 3 entries) specifies a package which contains the 3 columns (ordered 0-2):

0. **TPL** (`<tpli_name>`)): The name of the TriBITS TPL `<TPL_NAME>`. This name must be unique across all other TriBITS TPLs in this or any other TriBITS repo that might be combined into a single TriBITS project meta-build. However, a TPL can be redefined (see below). The name should be a valid identifier (e.g. matches the regex `[a-zA-Z_][a-zA-Z0-9_]*`).
1. **FINDMOD** (`<tpli_findmod>`): The relative path for the find module, usually with the name `FindTPL<TPL_NAME>.cmake`. This path is relative to the repository base directory. If just the base path for the find module is given, ending with `"/"` (e.g. `"cmake/tpls/"`) then the find module will be assumed to be under that this directory with the standard name (e.g. `cmake/tpls/FindTPL<TPL_NAME>.cmake`). A standard way to write a `FindTPL<TPL_NAME>.cmake` module is to use the function TRIBITS_TPL_DECLARE_LIBRARIES().
2. **CLASSIFICATION** (`<tpl0_classif>`): Gives the testing group `PT`, `ST`, `EX` and the maturity level `EP`, `RS`, `PG`, `PM`, `GRS`, `GPG`, `GPM`, `UM`. These are seprated by a coma with no space in between such as `"RS,PT"` for a "Research Stable", "Primary Tested" package. No spaces are allowed so that CMake treats this a one field in the array. The maturity level can be left off in which case it is assumed by default to be `UM` for "Unspecified Maturity".

A TPL defined in a upstream repo can listed again, which allows redefining the find module that is used to specificy the TPL. This allows downstream repos to add additional requirements on a given TPL. However, the downstream repo's find module file must find the TPL components that are fully compatible with the upstream's find module.

This macro just sets the varaible:

```
${REPOSITORY_NAME}_TPLS_FINDMODS_CLASSIFICATIONS
```

in the current scope. The advantages of using this macro instead of directly setting this varible include:

- Asserts that the varible `REPOSITORY_NAME` is defined and set
- Avoids having to hard-code the assumed repository name `${REPOSITORY_NAME}`. This provides more flexibility for how other TriBITS project name a given TriBITS repo (i.e. the name of repo subdirs).
- Avoid mispelling the name of the varible `${REPOSITORY_NAME}_TPLS_FINDMODS_CLASSIFICATIONS`. If you misspell the name of the macro, it is an immediate error in CMake.

TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS()

Disable a package automatically for a list of platforms.

Usage:

```
TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS( <packageName>
  <hosttype0> <hosttype1> ...)
```

If any of the host-type arguments `<hosttypei>` matches the `${PROJECT_NAME}_HOSTTYPE` variable for the current platform, then package `<packageName>` test group classification is changed to `EX`. Changing the package test group classification to `EX` results in the package being disabled by default. However, an explicit enable can still enable the package.

TRIBITS_EXCLUDE_FILES()

Exclude package files/dirs from the source distribution by appending `CPACK_SOURCE_IGNORE_FILES`.

Usage:

```
TRIBITS_EXCLUDE_FILES(<file0> <file1> ...)
```

This is called in the package's top-level <packageDir>/CMakeLists.txt file and each file or directory name `<filei>` is actually interpreted by CMake/CPack as a regex that is prefixed by the project's and packages source directory names so as to not exclude files and directories of the same name and path from other packages. If `<filei>` is an absolute path it it not prefixed but is appended to `CPACK_SOURCE_IGNORE_FILES` unmodified.

TRIBITS_INCLUDE_DIRECTORIES()

This function is to override the standard behavior of include_directories for a TriBITS package.

Usage:

```
TRIBITS_INCLUDE_DIRECTORIES(
  [REQUIRED_DURING_INSTALLATION_TESTING] <dir0> <dir1> ...
  )
```

If specified, `REQUIRED_DURING_INSTALLATION_TESTING` can appear anywhere in the argument list.

This function allows overriding the default behavior for installation testing, to ensure that include directories will not be inadvertently added to the build lines for tests during installation testing. Normally we want the include directories to be handled as cmake usually does. However during TriBITS installation testing we do not want most of the include directories to be used as the majority of the files should come from the installation we are building against. There is an exception to this and that is when there are test only headers that are needed. For that case we allow people to set `REQUIRED_DURING_INSTALLATION_TESTING` to tell us that this include directory does need to be set for instaltion testing.

TRIBITS_PACKAGE()

Macro called at the very beginning of a package's top-level CMakeLists.txt file.

Usage:

```
TRIBITS_PACKAGE(
  <packageName>
  [ENABLE_SHADOWING_WARNINGS]
  [DISABLE_STRONG_WARNINGS]
  [CLEANED]
  [DISABLE_CIRCULAR_REF_DETECTION_FAILURE]
  )
```

See TRIBITS_PACKAGE_DECL() for the documentation for the arguments and TRIBITS_PACKAGE_DECL() and TRIBITS_PACKAGE() for a description the side-effects (and varibles set) after calling this macro.

TRIBITS_PACKAGE_DECL()

Macro called at the very beginning of a package's top-level CMakeLists.txt file when a packages has subpackages.

If the package does not have subpackages, just call TRIBITS_PACKAGE() which calls this macro.

Usage:

```
TRIBITS_PACKAGE_DECL(
  <packageName>
  [ENABLE_SHADOWING_WARNINGS]
  [DISABLE_STRONG_WARNINGS]
  [CLEANED]
  [DISABLE_CIRCULAR_REF_DETECTION_FAILURE]
  )
```

The arguments are:

`<packageName>`

Gives the name of the Package, mostly just for checking and documentation purposes. This much match the name of the package provided in the PackagesLists.cmake or it is an error.

`ENABLE_SHADOWING_WARNINGS`

If specified, then shadowing warnings will be turned on for supported platforms/compilers. The default is for shadowing warnings to be turned off. Note that this can be overridden globally by setting the cache variable ${PROJECT_NAME}_ENABLE_SHADOWING_WARNINGS.

`DISABLE_STRONG_WARNINGS`

If specified, then all strong warnings will be turned off, if they are not already turned off by global cache variables. Strong warnings are turned on by default in development mode.

`CLEANED`

If specified, then warnings will be promoted to errors for all defined warnings.

`DISABLE_CIRCULAR_REF_DETECTION_FAILURE`

If specified, then the standard grep looking for RCPNode circular references that causes tests to fail will be disabled. Note that if these warnings are being produced then it means that the test is leaking memory and user like may also be leaking memory.

There are several side-effects of calling this macro:

- The the varibles listed the packages set of library targets `${PACKAGE_NAME}_LIB_TARGETS` and all targets `${PACKAGE_NAME}_ALL_TARGETS` and are initialized to emtpy.
- The local varibles `PACKAGE_SOURCE_DIR` and `PACKAGE_BINARY_DIR` are set for this package's use in its CMakeLists.txt files.
- Package-specific compiler options are set up in package-scoped (i.e., the package's subdir and its subdirs) in `CMAKE_<LANG>_FLAG`.
- This packages's cmake subdir `${PACKAGE_SOURCE_DIR}/cmake` is added to `CMAKE_MODULE_PATH` locally so that the package's try-compile modules can be read in with just a raw `INCLUDE()` leaving off the full path and the `*.cmake` extension.

TRIBITS_PACKAGE_DEF()

Macro called after subpackages are processed in order to handle the libraries, tests, and examples of the final package.

Usage:

```
TRIBITS_PACKAGE_DEF()
```

If the package does not have subpackages, just call TRIBITS_PACKAGE() which calls this macro.

This macro has several side effects:

- The varible `PACKAGE_NAME` is set in the local scope for usage by the package's CMakeLists.txt files.
- The intra-package dependency varibles (i.e. list of include directoires, list of libraries, etc.) are initialized to emtpy.

TRIBITS_PACKAGE_POSTPROCESS()

Macro called at the very end of a package's top-level CMakeLists.txt file. This macro performs some critical post-processing activities before downstream packages are processed.

Usage:

```
TRIBITS_PACKAGE_POSTPROCESS()
```

NOTE: It is unfortunate that a packages's CMakeLists.txt file must call this macro but limitations of the CMake language make it necessary to do so.

TRIBITS_PROCESS_SUBPACKAGES()

Macro that processes subpackages for packages that have them. This is called in the parent packages top-level CMakeLists.txt file.

Usage:

```
TRIBITS_PROCESS_SUBPACKAGES()
```

Must be called after TRIBITS_PACKAGE_DECL() but before TRIBITS_PACKAGE_DEF().

TRIBITS_PROJECT()

Defines and processes a TriBITS project.

Usage:

```
TRIBITS_PROJECT()
```

Requires that the project name variable `PROJECT_NAME` be defined before calling this macro. Also, all default values for project settings should be set before calling this (see TriBITS Global Project Settings). Also, the variable `${PROJECT_NAME}_TRIBITS_DIR` must be set as well.

This macro then adds all of the necssary paths to `CMAKE_MODULE_PATH` and then performs all processing of the TriBITS project files (see ???).

ToDo: Give documentation!

TRIBITS_PROJECT_DEFINE_EXTRA_REPOSITORIES()

Declare a set of extra extra repositories for a project (typically in the project's <projectDir>/cmake/ExtraRepositoriesList.cmake file).

Usage:

```
TRIBITS_PROJECT_DEFINE_EXTRA_REPOSITORIES(
  <repo0_name> <repo0_dir> <repo0_type> <repo0_url> <repo0_packstat> <repo0_classif>
  <repo1_name> <repo1_dir> <repo1_type> <repo1_url> <rep10_packstat> <repo1_classif>
  ...
 )
```

This macro takes in a 2D array with 6 columns, where each row defines an extra repository. The 6 columns (ordered 0-5) are:

0. **REPO_NAME** (`<repoi_name>`): The name given to the repository `REPOSITORY_NAME`.
1. **REPO_DIR** (`<repoi_dir>`): The relative directory for the repository under the project directory `${PROJECT_SOURCE_DIR}` (or `<projectDir>`). If this is set to empty quoted string `""` `', then the relative directory name is assumed to be same as the repository name `<repoi_name>`.

2. **REPO_TYPE** (<repoi_type>): The version control (VC) type of the repo. Value choses include GIT and SVN (i.e. Subversion). *WARNING:* Only VC repos of type GIT can fully participate in the TriBITS development tool workflows. The other VC types are supported for basic cloning and updating using the TribitsCTestDriverCore.cmake script.

3. **REPO_URL** (<repoi_url>): The URL of the VC repo. This info is used to initially obtain the repo source code using the VC tool listed in <repoi_type>. If the repos don't need to be cloned for needed use cases, then this can be the empty quoted string "".

4. **REPO_PACKSTAT** (<repoi_packstat>): Determines if the VC repository contains any TriBITS packages or if it just provides directories and files. If the VC repo contains TriBITS packages, then this field is set as the empty quoted string "", then this repository is considered to be a TriBITS Repository and must therefore contain the files described in TriBITS Repository Core Files. If the listed repository is **not** a TriBITS repository, and just provides directories and files, then this field is set as NOPACKAGES.

5. **REPO_CLASSIFICATION** (<repoi_classif>): Gives the testing classification of the repository which also happens to be the CTest/CDash testing mode and the default dashboard track. The valid values are Continuous, Nightly, and Experimental. See Repository Test Classification for a detailed description.

This command is used to put together one or more VC and/or TriBITS repositories to construct a larger project. Files that contain this macro call are what is passed in for the option <Project>_EXTRAREPOS_FILE). Repositories with <repoi_packstat>="" are **not** TriBITS Repositories and are technically not considered at all during the basic configuration of the a TriBITS project. They are only listed in this file so that they can be used in the version control logic for tools that perform version control with the repositories (such as cloning, updating, looking for changed files, etc.). For example, a non-TriBITS repo can be used to grab a set of directories and files that fill in the definition of a package in an upstream repository (see How to insert a package into an upstream repo). Also, non-TriBITS repos can be used to provide extra test data for a given pakage or a set of packages so that extra tests can be run.

NOTE: These repositories must be listed in the order of package dependencies. That is, all of the packages listed in repository i must have upstream TPL and SE package dependencies listed before this package in this repository or in upstream repositories i-1, i-2, etc.

NOTE: This module just sets the local varaible:

```
${PROJECT_NAME}_EXTRAREPOS_DIR_REPOTYPE_REPOURL_PACKSTAT_CATEGORY
```

in the current scope. The advantages of using this macro instead of directly setting this varible include:

- Asserts that the varible PROJECT_NAME is defined and set.
- Avoids having to hard-code the assumed project name ${PROJECT_NAME}. This provides more flexibility for how other TriBITS project name a given TriBITS repo (i.e. the name of repo subdirs).
- Avoid mispelling the name of the varible ${PROJECT_NAME}_EXTRAREPOS_DIR_REPOTYPE_REPOURL_PACKSTAT_CATEGORY. If you misspell the name of the macro, it is an immediate error in CMake.

TRIBITS_SET_ST_FOR_DEV_MODE()

Function that allows packages to easily make a feature ST for development builds and PT for release builds by default.

Usage:

```
TRIBITS_SET_ST_FOR_DEV_MODE(<outputVar>)
```

${<outputVar>} is set to ON or OFF based on the configure state. In development mode (i.e. ${PROJECT_NAME}_ENABLE_DEVELOPMENT_MODE==ON), ${<outputVar>} will be set to ON only if ST code is enabled (i.e. ${PROJECT_NAME}_ENABLE_SECONDARY_TESTED_CODE==ON), otherwise it is set to OFF. In release mode (i.e. ${PROJECT_NAME}_ENABLE_DEVELOPMENT_MODE==OFF) it is always set to ON. This allows some sections of a TriBITS package to be considered ST for development mode reducing testing time which includes only PT code., while still having important functionality available to users by default in a release.

TRIBITS_SUBPACKAGE()

Declare a subpackage.

Usage:

```
TRIBITS_SUBPACKAGE(<spkgName>)
```

Once called, the following local varibles are in scope:

`PARENT_PACKAGE_NAME`

The name of the parent package.

`SUBPACKAGE_NAME`

The local name of the subpackage (does not contain the parent package name).

`SUBPACKAGE_FULLNAME`

The full project-level name of the subpackage (which includes the parent package name at the beginning).

`PACKAGE_NAME`

Inside the subpackage, the same as `SUBPACKAGE_FULLNAME`.

TRIBITS_SUBPACKAGE_POSTPROCESS()

Postprocess after defining a subpackage.

Usage:

```
TRIBITS_SUBPACKAGE_POSTPROCESS()
```

NOTE: It is unfortunate that a Subpackages's CMakeLists.txt file must call this macro but limitations of the CMake language make it necessary to do so.

TRIBITS_TPL_DECLARE_LIBRARIES()

Function that sets up cache variables for users to specify where to find a TPL's headers and libraries. This function is typically called inside of a file `FindTPL<tpl_name>.cmake` file.

Usage:

```
TRIBITS_TPL_DECLARE_LIBRARIES(
  <tpl_name>
  [REQUIRED_HEADERS <header1> <header2> ...]
  [MUST_FIND_ALL_HEADERS]
  [REQUIRED_LIBS_NAMES <libname1> <libname2> ...]
  [MUST_FIND_ALL_LIBS]
  [NO_PRINT_ENABLE_SUCCESS_FAIL]
  )
```

This function can set up a with header files and/or libraries.

The input arguments to this function are:

- `<tpl_name>`: Name of the TPL that is listed in a TPLsList.cmake file. Below, this is referted to as the local CMake variable `TPL_NAME`.
- `REQUIRED_HEADERS`: List of header files that are searched for the TPL using `FIND_PATH()`.
- `MUST_FIND_ALL_HEADERS`: If set, then all of the header files listed in REQUIRED_HEADERS must be found in order for TPL_${TPL_NAME}_INCLUDE_DIRS to be defined.

- REQUIRED_LIBS_NAMES: List of libraries that are searched for when looked for the TPLs libraries with FIND_LIBRARY(...).
- MUST_FIND_ALL_LIBS: If set, then all of the library files listed in REQUIRED_LIBS_NAMES must be found or the TPL is considered not found!
- **NO_PRINT_ENABLE_SUCCESS_FAIL: If set, then the final success/fail** will not be printed

The following cache variables, if set, will be used by that this function:

- ${TPL_NAME}_INCLUDE_DIRS:PATH: List of paths to search first for header files defined in REQUIRED_HEADERS.
- ${TPL_NAME}_INCLUDE_NAMES:STIRNG: List of include names to be looked for instead of what is specified in REQUIRED_HEADERS.
- ${TPL_NAME}_LIBRARY_DIRS:PATH: The list of directories to search first for libraies defined in REQUIRED_LIBS_NAMES.
- ${TPL_NAME}_LIBRARY_NAMES:STIRNG: List of library names to be looked for instead of what is specified in REQUIRED_LIBS_NAMES.

This function sets global varibles to return state so it can be called from anywhere in the call stack. The following cache variables defined that are intended for the user to set and/or use:

- TPL_${TPL_NAME}_INCLUDE_DIRS: A list of common-separated full directory paths that contain the TPLs headers. If this varible is set before calling this function, then no headers are searched for and this variable will be assumed to have the correct list of header paths.
- TPL_${TPL_NAME}_LIBRARIES: A list of commons-seprated full library names (output from FIND_LIBRARY(...)) for all of the libraries found for the TPL. IF this varible is set before calling this function, no libraries are searched for and this varaible will be assumed to have the correct list of libraries to link to.

TRIBITS_WRITE_FLEXIBLE_PACKAGE_CLIENT_EXPORT_FILES()

Utility function for writing ${PACKAGE_NAME}Config.cmake and/or the Makefile.export.${PACKAGE_NAME} for package PACKAGE_NAME with some greater flexibility than TRIBITS_WRITE_PACKAGE_CLIENT_EXPORT_FILES()

Usage:

```
TRIBITS_WRITE_FLEXIBLE_PACKAGE_CLIENT_EXPORT_FILES(
  PACKAGE_NAME <pakageName>
  [EXPORT_FILE_VAR_PREFIX <exportFileVarPrefix>]
  [WRITE_CMAKE_CONFIG_FILE <cmakeConfigFileFullPath>]
  [WRITE_EXPORT_MAKLEFILE <exportMakefileFileFullPath>]
  [WRITE_INSTALL_CMAKE_CONFIG_FILE]
  [WRITE_INSTALL_EXPORT_MAKLEFILE]
  )
```

The arguments are:

```
PACKAGE_NAME <pakageName>
```

Gives the name of the TriBITS package for which the export files should be created for.

```
EXPORT_FILE_VAR_PREFIX <exportFileVarPrefix>
```

If specified, then all of the varibles in the generated export files will be prefixed with "<exportFileVarPrefix>_" instead of "${PACKAGE_NAME}_". This is to provide flexibility.

```
WRITE_CMAKE_CONFIG_FILE <cmakeConfigFileFullPath>
```

If specified, then the package <packageName>'s cmake configure export file for extenral CMake client projects will be created in the file <cmakeConfigFileFullPath>. NOTE: the argument should be the full path!

`WRITE_EXPORT_MAKLEFILE <exportMakefileFileFullPath>`

If specified, then the package <packageName>'s cmake configure export file for external Makefile client projects will be created in the file <exportMakefileFileFullPath>. NOTE: the argument should be the full path!

`WRITE_INSTALL_CMAKE_CONFIG_FILE`

If specified, then the package <packageName>'s install cmake configure export to be installed will be written. The name and location of this file is hard-coded.

`WRITE_INSTALL_EXPORT_MAKLEFILE`

If specified, then the package <packageName>'s install export makefile to be installed will be written. The name and location of this file is hard-coded.

NOTE: The arguments to this function may look strange but the motivation is to support versy speicalized use cases such as when a TriBITS package needs to generate an export makefile for a given package but name the export makefile differently and use different variable name prefixes. The particular driver use case is when wrapping an external autotools project that depends on Trilinos and needs to read in the Makefile.export.Trilinos file but this file needs to be generated for a subset of enabled packages on the fly during a one-pass configure.

NOTE: This function does *not* contain the the INSTALL() commands because CMake will not allow those to even be present in scripting mode that is used for unit testing this function.

12.3 General Utility Macros and Functions

The following subsections give detailed documentation for some CMake macros and functions which are *not* a core part of the TriBITS system but are included in the TriBITS system that are used inside of the TriBITS system and are provided as a convenience to TriBITS project developers. One will see many of these functions and macros used throughout the implementation of TriBITS and even in the `CMakeLists.txt` files for projects that use TriBITS.

These macros and functions are *not* prefixed with `TRIBITS_`. There is really not a large risk to defining and using these non-namespaces utility functions and macros. It turns out that CMake allows you to redefine any macro or function, even built-in ones, inside of your project so even if CMake did add new comamnds that clashed with these names, there would be no conflicit. When overridding a built-in command `some_builtin_command()`, you can always access the original built-in command as `_some_builtin_command()`.

ADD_SUBDIRECTORIES()

Macro that adds a list of subdirectories all at once (removed boiler-place code).

Usage:

```
ADD_SUBDIRECTORIES(<dir1> <dir2> ...)
```

ADVANCED_OPTION()

Macro that sets an option and marks it as advanced (removes boiler-plate and duplication).

Usage:

```
ADVANCED_OPTION(<varName> [other arguments])
```

This is identical to:

```
ADVANCED_OPTION(<varName> [other arguments])
MARK_AS_ADVANCED(<varName>)
```

ADVANCED_SET()

Macro that sets a variable and marks it as advanced (removes boiler-plate and duplication).

Usage:

```
ADVANCED_SET(<varName> [other arguments])
```

This is identical to:

```
ADVANCED_SET(<varName> [other arguments])
MARK_AS_ADVANCED(<varName>)
```

APPEND_CMNDLINE_ARGS()

Utility function that appends command-line arguments to a variable of command-line options.

Usage:

```
APPEND_CMNDLINE_ARGS(<var> "<extraArgs>")
```

This function just appends the command-line arguments in the string `"<extraArgs>"` but does not add an extra space if `<var>` is empty on input.

APPEND_GLOB()

Utility macro that does a `FILE(GLOB ...)` and appends to an existing list (removes boiler-plate code).

Usage:

```
APPEND_GLOB(<fileListVar> <glob0> <glob1> ...)
```

On output, `<fileListVar>` will have the list of glob files appended.

APPEND_GLOBAL_SET()

Utility macro that appends arguments to a global variable (reduces boiler-plate code and mistakes).

Usage:

```
APPEND_GLOBAL_SET(<varName> <arg0> <arg1> ...)
```

NOTE: The variable `<varName>` must exist before calling this function. To set it empty initially use GLOBAL_NULL_SET().

APPEND_SET()

Utility function to append elements to a varible (reduces boiler-plate code).

Usage:

```
APPEND_SET(<varName> <arg0> <arg1> ...)
```

Just calls:

```
LIST(APPEND <varName> <arg0> <arg1> ...)
```

APPEND_STRING_VAR()

Append strings to an existing string variable (reduces boiler-place code and reduces mistakes).

Usage:

```
APPEND_STIRNG_VAR(<stringVar> "<string1>" "<string2>" ...)
```

Note that the usage of the characters `'['`,`']'`,`'{'`,`'}'` are taken by CMake to bypass the meaning of ';' to separate string characters.

If you want to ignore the meaning of these special characters and are okay with just adding one string at a time use APPEND_STRING_VAR_EXT().

APPEND_STRING_VAR_EXT()

Append a single string to an existing string variable, ignoring ';' (reduces boiler-place code and reduces mistakes).

Usage:

```
APPEND_STIRNG_VAR_EXT(<stringVar> "<string>")
```

Simply sets `<stringVar> = "${<stringVar>}<string>"`.

APPEND_STRING_VAR_WITH_SEP()

Append strings to a given string varible, joining them using a seperator.

Usage:

```
APPEND_STRING_VAR_WITH_SEP(<stringVar> "<sepStr>" "<str0>" "<str>" ...)
```

Each of the strings `<stri>` are appended to `<stringVar>` using the separation string `<sepStr>`.

ASSERT_DEFINED()

Assert that a varaible is defined and if not call `MESSAGE(SEND_ERROR ...)`.

Usage:

```
ASSERT_DEFINED(<varName>)
```

This is used to get around the problem of CMake not asserting the defreferencing of undefined varibles. For example, how do you know if you did not mispell the name of a varible in an if statement like:

```
IF (SOME_VARBLE)
  ...
ENDIF()
```

? If you mispelled the varible `SOME_VARBLE` (which you likely did in this case), the the if statement will always be false. To avoid this problem when you always expect the explicitly set, instead do:

```
ASSERT_DEFINED(SOME_VARBLE)
IF (SOME_VARBLE)
  ...
ENDIF()
```

Now if you misspell the varible, it will asset and stop processing. This is not a perfect solution since you can mispell the varible name in the following if statemnt but typically you would always just copy and paste between the two statements so they are always the same. This is the best we can do in CMake unfortunately.

COMBINED_OPTION()

Set up a bool cache variable (i.e. an option) based on a set of dependent options.

Usage:

```
COMBINED_OPTION( <combinedOptionName>
  DEP_OPTIONS_NAMES <depOpName0> <depOptName1> ...
  DOCSTR "<docstr0>" "<docstr1" ...
  )
```

This sets up a bool cache variable `<combinedOptionName>` which is defaulted to `ON` if all of the listed dependent option varaibles `<depOpName0>`, `<depOptName1>`, ... are all `ON`. However, if `<combinedOptionName>` is set to `ON` by the user and not all of the dependent option varibles are also true, this results in a fatal eror and alll processing stops.

This is used by a CMake project to by default automatically turn on a feature that requires a set of other features to also be turned on but allows a user to disable the feature if desired.

CONCAT_STRINGS()

Concatenate a set of string argumnets.

Usage:

```
CONCAT_STRINGS(<outputVar> "<str0>" "<str1>" ...)
```

On output, `<outputVar>` is set to `"<str0><str1>..."`.

DUAL_SCOPE_APPEND_CMNDLINE_ARGS()

Utility function that appends command-line arguments to a variable of command-line options and sets the result in current scope and parent scope.

Usage:

```
DUAL_SCOPE_APPEND_CMNDLINE_ARGS(<var> "<extraArgs>")
```

Just calls APPEND_CMNDLINE_ARGS() and then `SET(<var> ${<var>} PARENT_SCOPE)`.

DUAL_SCOPE_PREPEND_CMNDLINE_ARGS()

Utility function that prepends command-line arguments to a variable of command-line options and sets the result in current scope and parent scope.

Usage:

```
DUAL_SCOPE_PREPEND_CMNDLINE_ARGS(<var> "<extraArgs>")
```

Just calls PREPEND_CMNDLINE_ARGS() and then `SET(<var> ${<var>} PARENT_SCOPE)`.

DUAL_SCOPE_SET()

Macro that sets a variable name both in the current scope and the parent scope.

Usage:

```
DUAL_SCOPE_SET(<varName> [other args])
```

It turns out that when you call ADD_SUBDIRECTORY(<someDir>) or enter a FUNCTION that CMake actaully creates a copy of all of the regular non-cache varaibles in the current scope in order to create a new set of variables for the CMakeLists.txt file in <someDir>. This means that if you call SET(SOMEVAR Blah PARENT_SCOPE) that it will not affect the value of SOMEVAR in the current scope. This macro therefore is designed to set the value of the variable in the current scope and the parent scope in one shot to avoid confusion.

Global variables are different. When you move to a subordinate CMakeLists.txt file or enter a function, a local copy of the variable is *not* created. If you set the value name locally, it will shadow the global variable. However, if you set the globlal value with SET(SOMEVAR someValue CACHE INTERNAL “”), then the value will get changed in the current subordinate scope and in all parent scopes all in one shot!

GLOBAL_NULL_SET()

Set a variable as a null internal global (cache) variable (removes boiler plate).

Usage:

```
GLOBAL_NULL_SET(<varName>)
```

This just calls:

```
SET(<varName> "" CACHE INTERNAL "")
```

GLOBAL_SET()

Set a variable as an internal global (cache) variable (removes boiler plate).

Usage:

```
GLOBAL_SET(<varName> [other args])
```

This just calls:

```
SET(<varName> [other args] CACHE INTERNAL "")
```

JOIN()

Join a set of strings into a single string using a join string.

Usage:

```
JOIN(<outputStrVar> <sepStr> <quoteElements>
  "<string0>" "<string1"> ...)
```

Arguments:

<outputStrVar>

The name of a variable that will hold the output string.

<sepStr>

A string to use to join the list of strings.

<quoteElements>

If TRUE, then each <stingi> is quoted using an escaped quote char \". If FALSE then no escaped quote is used.

On output the variable <outputStrVar> is set to:

```
"<string0><sepStr><string1><sepStr>..."
```

If <quoteElements>=TRUE, then it is set to:

```
"\"<string0>\"<sepStr>\"<string1>\"<sepStr>..."
```

For example, the latter can be used to set up a set of command-line arguments given a CMake array like:

```
JOIN(CMND_LINE_ARGS " " TRUE ${CMND_LINE_ARRAY})
```

WARNING: Be careful to quote string arguments that have spaces because CMake interpet those ase array boundaries.

MESSAGE_WRAPPER()

Function that wraps the standard CMake/CTest MESSAGE() function call in order to allow unit testing to intercept the call.

Usage:

```
MESSAGE_WRAPPER(<arg0> <arg1> ...)
```

This function takes exactly the same argumnets as built-in MESSAGE(). When the varible MESSAGE_WRAPPER_UNIT_TEST_MODE is set to TRUE, then this function will not call MESSAGE(<arg0> <arg1> ...) but instead will prepend set to global varible MESSAGE_WRAPPER_INPUT that input argumnets. To capture just this call's input, first call GLOBAL_NULL_SET(MESSAGE_WRAPPER_INPUT(MESSAGE_WRAPPER_INPUT) before calling this function.

This function allows one to unit test other user-defined CMake macros and functions that call this to catch error conditions wihtout stopping the CMake program. Otherwise, this is used to capture print messages to verify that they say the right thing.

MULTILINE_SET()

Function to set a single string by concatenating a list of separate strings

Usage:

```
MULTILINE_SET(<outputStrVar>
  "<string0>"
  "<string1>"
  ...
  )
```

On output, the local variables <outputStrVar> is set to:

```
"<string0><string1>..."
```

The purpose of this is to make it easier to set longer strings without going to far to the right.

PARSE_ARGUMENTS()

Parse a set of macro/functon input arguments into different lists. This allows the easy implementation of keyword-based user-defined macros and functions.

Usage:

```
PARSE_ARGUMENTS(
  <prefix>  <argNamesList>  <optionNamesList>
  <inputArgsList>
  )
```

Arguments to this macro:

`<prefix>`

Prefix `<prefix>_` added the list and option variables created listed in `<argNamesList>` and `<optionNamesList>`.

`<argNamesList>`

Quoted array of list arguments (e.g. "`<argName0>;<argName1>;...`"). For each varaible name `<argNamei>`, a local varible will be created in the current scope with the name `<prefix>_<argNamei>` which gives the list of varibles parsed out of `<inputArgsList>`.

`<optionNamesList>`

Quoted array of list arguments (e.g. "`<optName0>;<optName1>;...`"). For each varaible name `<optNamei>`, a local varible will be created in the current scope with the name `<prefix>_<optNamei>` that is either set to `TRUE` or `FALSE` depending if `<optNamei>` apears in `<inputArgsList>` or not.

`<inputArgsList>`

List of arguments keyword-based arguments passed in for the outer macro or function to be parsed out into the different argument and option lists.

What this macro does is very simple yet very useful. What it does is to allow you to create your own user-defined keyword-based macros and functions like is used by some built-in CMake comamnds..

For example, consider the following user-defined macro that uses both positional and keyword-based arguments using `PARSE_ARGUMENTS()`:

```
MACRO(PARSE_SPECIAL_VARS  BASE_NAME)

  PARSE_ARGUMENTS(
    #prefix
    ${BASE_NAME}
    #lists
    "ARG0;ARG1;ARG2"
    #options
    "OPT0;OPT1"
    ${ARGN}
    )

ENDMACRO()
```

Calling this macro as:

```
PARSE_SPECIAL_VARS(MyVar ARG0 a b ARG2 c OPT1)
```

sets the following varibles in the current scope:

```
MyVar_ARG0="a;b"
MyVar_ARG1=""
MyVar_ARG2="c"
MyVar_OPT0="FALSE"
MyVar_OPT1="TRUE"
```

This allows you to define user-defined macros and functions that have a mixture of positional arguments and keyword-based arguments like you can do in other languages. The keyword-based arguments can be passed in any order and those that are missing are empty (or false) by default.

Any initial arguments that are not recongnised as `<argNamesList>` or `<optionNamesList>` keyword arguments will be put into the local varible `<prefix>_DEFAULT_ARGS`. If no arguments in `${ARGN}` match any in `<argNamesList>`, then all non-option arguments are point into `<prefix>_DEFAULT_ARGS`. For example, if you pass in:

```
PARSE_SPECIAL_VARS(MyVar ARG5 a b c)
```

you will get:

```
MyVar_DEFAULT_ARGS="a;b;c"
MyVar_ARG0=""
MyVar_ARG1=""
MyVar_ARG2=""
MyVar_OPT0="FALSE"
MyVar_OPT1="FALSE"
```

Multiple occurances of keyword arguments in `${ARGN}` is allowed but only the last one listed will be recored. For example, if you call:

```
PARSE_SPECIAL_VARS(MyVar ARG1 a b ARG1 c)
```

then this wil set:

```
MyVar_ARG0=""
MyVar_ARG1="c"
MyVar_ARG2=""
MyVar_OPT0="FALSE"
MyVar_OPT1="FALSE"
```

This is actually consistent with the way that most arugment list parsers behave with respect to multiple instances of the same argument so hopefully this will not be a surprise to anyone.

If you put an option keyword in the middle of a keyword argument list, the option keyword will get pulled out of the list. For exmaple, if you call:

```
PARSE_SPECIAL_VARS(MyVar ARG0 a OPT0 c)
```

then this wil set:

```
MyVar_ARG0="a;c"
MyVar_ARG1=""
MyVar_ARG2=""
MyVar_OPT0="TRUE"
MyVar_OPT1="FALSE"
```

If `PARSE_ARGUMENTS_DUMP_OUTPUT_ENABLED``is set to ``TRUE`, then a bunch of detailed debug info will be printed. This should only lbe used in the most desparate of debug situations because it will print a *lot* of output!

PERFORMANCE: This function will scale as:

```
O( (len(<argNamesList>) * len(<optionNamesList>)) * len(<inputArgsList>) )
```

Therefore, this could scale very badly for large lests of argument and option names and input argument lists.

PREPEND_CMNDLINE_ARGS()

Utility function that prepends command-line arguments to a variable of command-line options.

Usage:

```
PREPEND_CMNDLINE_ARGS(<var> "<extraArgs>")
```

This function just prepends the command-line arguments in the string `"<extraArgs>"` but does not add an extra space if `<var>` is empty on input.

PREPEND_GLOBAL_SET()

Utility macro that prepends arguments to a global variable (reduces boiler-plate code and mistakes).

Usage:

```
PREPEND_GLOBAL_SET(<varName> <arg0> <arg1> ...)
```

NOTE: The variable `<varName>` must exist before calling this function. To set it empty initially use GLOBAL_NULL_SET().

APPEND_SET()

Utility function to append elements to a varible (reduces boiler-plate code).

Usage:

```
APPEND_SET(<varName> <arg0> <arg1> ...)
```

Just calls:

```
LIST(APPEND <varName> <arg0> <arg1> ...)
```

PRINT_NONEMPTY_VAR()

Print a defined variable giving its name then value only if it is not empty.

Usage:

```
PRINT_NONEMPTY_VAR(<varName>)
```

Calls `PRINT_VAR(<varName>)` if `${<varName>}` is not empty.

PRINT_NONEMPTY_VAR_WITH_SPACES()

Print a defined variable giving its name then value printed with spaces instead of ';' but only if it is not empty.

Usage:

```
PRINT_NONEMPTY_VAR_WITH_SPACES(<varName>)
```

Prints the variable as:

```
<varName>: <ele0> <ele1> ...
```

PRINT_VAR()

Unconditionally print a variable giving its name then value.

Usage:

```
PRINT_VAR(<varName>)
```

This prints:

```
MESSAGE("-- " "${VARIBLE_NAME}='${${VARIBLE_NAME}}'")
```

The variable `<varName>` can be defined or undefined or empty. This uses an explicit "-- " line prefix so that it prints nice even on Windows CMake.

REMOVE_GLOBAL_DUPLICATES()

Remove duplicate elements from a global list variable.

Usage:

```
REMOVE_GLOBAL_DUPLICATES(<globalVarName>)
```

This function is necessary in order to preserve the "global" nature of the variable. If you just call LIST(REMOVE_DUPLICATES ...) it will actually create a local variable of the same name and shadow the global variable! That is a fun bug to track down!

SET_AND_INC_DIRS()

Set a variable to an include dir and call `INCLUDE_DIRECTORIES()` (removes boiler plate).

Usage:

SET_AND_INC_DIRS(<dirVarName> <includeDir>)

On output, this justs `<dirVarName>` to `<includeDir>` in the local scope and calls `INCLUDE_DIRECTORIES(<includeDir>)`.

SET_CACHE_ON_OFF_EMPTY()

Usage:

```
SET_CACHE_ON_OFF_EMPTY(<varName> <initialVal> "<docString>" [FORCE])
```

Sets a special string cache variable with possible values "", "ON", or "OFF". This results in a nice dropdown box in the CMake cache manipulation GUIs.

SET_DEFAULT()

Give a local variable a default if a non-empty value is not already set.

Usage:

```
SET_DEFAULT(<varName> <arg0> <arg1> ...)
```

If on input `"${<varName>}"==""`, then `<varName>` is set to the given default. Otherwise, the existing non-empty value is preserved.

SET_DEFAULT_AND_FROM_ENV()

Set a default value for a local variable and override from an env var of the same name if it is set.

Usage:

```
SET_DEFAULT_AND_FROM_ENV(<varName> <defaultVal>)
```

First calls `SET_DEFAULT(<varName> <defaultVal>)` and then looks for an environment variable named `<varName>` and if non-empty, then overrides the value of `<varName>`.

This macro is primarily used in CTest code to provide a way to pass in the value of CMake variables. Older versions of `ctest` did not support the option `-D <var>:<type>=<value>` to allow varaibles to be set through the commandline like `cmake` always allowed.

SPLIT()

Split a string varible into a string array/list variable.

Usage:

```
SPLIT("<inputStr>" "<sepStr>" <outputStrListVar>)
```

The <sepStr> string is used with STRING(REGEX ...) to replace all occurrences of <sepStr>` in `<inputStr> with ";" and writing into <outputStrListVar>.

WARNING: <sepStr> is interpreted as a regular expression so keep that in mind when considering special regex chars like '*', '.', etc!

TIMER_GET_RAW_SECONDS()

Return the raw time in seconds since epoch, i.e., since 1970-01-01 00:00:00 UTC.

Usage:

```
TIMER_GET_RAW_SECONDS(<rawSecondsVar>)
```

This function is used along with TIMER_GET_REL_SECONDS(), and TIMER_PRINT_REL_TIME() to time big chunks of CMake code for timing and profiling purposes. See TIMER_PRINT_REL_TIME() for more details and an example.

NOTE: This function runs an external process to run the date command. Therefsore, it only works on Unix/Linux type systems that have a standard date command. Since this runs an external process, this function should only be used to time very course grained operations (i.e. that take longer than a second).

TIMER_GET_REL_SECONDS()

Return the relative time between start and stop seconds.

Usage:

```
TIMER_GET_REL_SECONDS(<startSeconds> <endSeconds> <relSecondsOutVar>)
```

This simple function computes the relative number of seconds between <startSeconds> and <endSeconds> (i.e. from TIMER_GET_RAW_SECONDS()) and sets the result in the local variable <relSecondsOutVar>.

TIMER_PRINT_REL_TIME()

Print the relative time between start and stop timers in <min>m<sec>s format.

Usage:

TIMER_PRINT_REL_TIME(<startSeconds> <endSeconds> "<messageStr>")

Differences the raw times <startSeconds> and <endSeconds> (i.e. gotten from TIMER_GET_RAW_SECONDS()) and prints the time in <min>m<sec>s format. This can only resolve times a second or greater apart. If the start and end times are less than a second then 0m0s will be printed.

This is meant to be used with TIMER_GET_RAW_SECONDS() to time expensive blocks of CMake code like:

```
TIMER_GET_RAW_SECONDS(REAL_EXPENSIVE_START)

REAL_EXPENSIVE(...)

TIMER_GET_RAW_SECONDS(REAL_EXPENSIVE_END)

TIMER_PRINT_REL_TIME(${REAL_EXPENSIVE_START} ${REAL_EXPENSIVE_END}
   "REAL_EXPENSIVE() time")
```

This will print something like:

```
REAL_EXPENSIVE() time: 0m5s
```

Again, don't try to time something that takes less than 1 second as it will be recored as `0m0s`.

UNITTEST_COMPARE_CONST()

Perform a single unit test equality check and update overall test statistics

Usage:

```
UNITTEST_COMPARE_CONST(<varName> <expectedValue>)
```

If `${<varName>} == <expectedValue>`, then the check passes, otherwise it fails. This prints the variable name and values and shows the test result.

This updates the global variables `UNITTEST_OVERALL_NUMRUN`, `UNITTEST_OVERALL_NUMPASSED`, and `UNITTEST_OVERALL_PASS` which are used by the unit test harness system to assess overall pass/fail.

UNITTEST_STRING_REGEX()

Perform a series regexes of given strings and update overall test statistics.

Usage:

```
UNITTEST_STRING_REGEX(
  <inputString>
  REGEX_STRINGS <str0> <str1> ...
  )
```

If the `<inputString>` matches all of the of the regexs `<str0>`, "<str1>", ..., then the test passes. Otherwise it fails.

This updates the global variables `UNITTEST_OVERALL_NUMRUN`, `UNITTEST_OVERALL_NUMPASSED`, and `UNITTEST_OVERALL_PASS` which are used by the unit test harness system to assess overall pass/fail.

UNITTEST_FILE_REGEX()

Perform a series regexes of given strings and update overall test statistics.

Usage:

```
UNITTEST_FILE_REGEX(
  <inputFileName>
  REGEX_STRINGS <str1> <str2> ...
  )
```

The contents of `<inputFileName>` are read into a string and then passed to UNITTEST_STRING_REGEX() to assess pass/fail.

UNITTEST_FINAL_RESULT()

Print final statstics from all tests and assert final pass/fail

Usage:

```
UNITTEST_FINAL_RESULT(<expectedNumPassed>)
```

If `${UNITTEST_OVERALL_PASS}==TRUE` and `${UNITTEST_OVERALL_NUMPASSED} == <expectedNumPassed>`, then the overall test program is determined to have passed and string:

```
"Final UnitTests Result: PASSED"
```

is printed. Otherwise, the overall tets program is determined to have failed, the string:

```
"Final UnitTests Result: FAILED"
```

is printed and `MESSAGE(SEND_ERROR "FAIL")` is called.

The reason that we require passing in the expected number of passed tests is an an extra precaution to make sure that important unit tests are not left out. CMake is a loosely typed language and it pays to be a little paranoid.

13 Appendix

13.1 History of TriBITS

TriBITS started development in November 2007 as a set of helper macros to provide a CMake build system for a small subset of packages in Trilinos. The initial goal was to just to support a native Windows build (using Visual C++) to compile and install these few Trilinos packages on Windows for usage by another project (the Sandia Titan project which included VTK). At that time, Trilinos was using a highly customized autotools build system. Initially, this CMake system was just a set of macros to streamline creating executables and tests. Some of the conventions started in that early effort (e.g. naming conventions of variables and macros where functions use upper case like old FORTRAN and variables are mixed case) were continued in later efforts and are reflected in the current. Then, stating in early 2008, a more detailed evaluation was performed to see if Trilinos should stitch over to CMake as the default (and soon only) supported build and test system (see "Why CMake?" in TriBITS Overview). This lead to the initial implementation of a scale-able package-based architecture (PackageArch) for the Trilinos CMake project in late 2008. This Trilinos CMake PackageArch system evolved over the next few years with development in the system slowing into 2010. This Trilinos CMake build system was then adopted as the build infrastructure for the CASL VERA effort in 2011 where CASL VERA packages were treated as add-on Trilinos packages (see Section Multi-Repository Support). Over the next year, there was significant development of the system to support larger multi-repo projects in support of CASL VERA. That lead to the decision to formally generalize the Trilinos CMake PackageArch build system outside of Trilinos and the name TriBITS was formally adopted in November 2011. Work to refactor the Trilinos CMake system into a general reusable stand-alone CMake-based build system started in October 2011 and an initial implementation was complete in December 2011 when it was used for the CASL VERA build system. In early 2012, the ORNL CASL-related projects Denovo and SCALE (see [SCALE]) adopted TriBITS as their native development build systems. Shortly after TriBITS was adopted the native build system for the the CASL-related University of Michigan code MPACT. In addition to being used in CASL, all of these codes also had a significant life outside of CASL. Because they used the same TriBITS build system, it proved relatively easy to keep these various codes integrated together in the CASL VERA code meta-build. At the same time, TriBITS well served the independent development teams and non-CASL projects independent from CASL VERA. Since the initial extraction of TriBITS from Trilinos, the TriBITS system was further extended and refined, driven by CASL VERA development and expansion. Independently, an early version of TriBITS from 2012 was adopted by the LiveV projectfootnote{https://github.com/lifev/cmake} which was forked and extended independently.

Note that a TriBITS "Package" is not the same thing as a "Package" in raw CMake terminology. In raw CMake, a "Package" is some externally provided bit of software or other utiltiy for which the current CMake project has an optional or required dependency. Therefore, a raw CMake "Package" actually maps to a TriBITS TPL. A raw CMake "Package" (e.g. Boost, CUDA, etc.) can be found using a standard CMake find module `Find<rawPackageName>.cmake` using the built-in command `FIND_PACKAGE(<rawPackageName>)`. It is unfortunate that the TriBITS and the raw CMake defintions of the term "Package" are not the same. However, the term "Package" was coined by the Trilinos project long ago before CMake was adopted as the Trilinos build system and Trilinos' definition of "Package" (going back to 1998) pre-dates the development of CMake and therefore dictated the terminology of TriBITS

13.2 Design Considerations for TriBITS

ToDo: Discuss design requirements.

ToDo: Discuss why it is a good idea to explicitly list packages instead of just searching for them. Hint: Think performance and circular dependencies!

13.3 checkin-test.py --help

Below is a snapshot of the output from `checkin-test.py --help`. This documentation contains a lot of information about the recommended development workflow (mostly related to pushing commits) and outlines a number of different use cases for using the script.

```
Usage: checkin-test.py [OPTIONS]

This tool does checkin testing with CMake/CTest and can actually do
the push itself using eg/git in a safe way.  In fact, it is
recommended that you use this script to push since it will amend the
last commit message with a (minimal) summary of the builds and tests
run with results.


Quickstart:
-----------

In order to do a solid checkin, perform the following recommended workflow
(different variations on this workflow are described below):

1) Commit changes in the local repo:

  # 1.a) See what files are changed, newly added, etc. that need to be committed
  # or stashed.
  $ eg status

  # 1.b) Stage the files you want to commit (optional)
  $ eg stage <files you want to commit>

  # 1.c) Create your local commits
  $ eg commit -- SOMETHING
  $ eg commit -- SOMETHING_ELSE
  ...

  # 1.d) Stash whatever changes are left you don't want to test/push (optional)
  $ eg stash

  NOTE: You can group your commits any way that you would like (see the basic
  eg/git documentation).

  NOTE: If not installed on your system, the eg script can be found at
  tribits/common_tools/git/eg.  Just add it to your path.

  NOTE: When multiple repos are involved, use egdist instead.  It is provided
  at tribits/common_tools/git/egdist.  See egdist --help for details.

2) Review the changes that you have made to make sure it is safe to push:

  $ cd $PROJECT_HOME
  $ eg local-stat                  # Look at the full status of local repo
  $ eg diff --name-status origin   # [Optional] Look at the files that have changed

  NOTE: The command 'local-stat' is a git alias that can be installed with the
  script tribits/common_tools/git/git-config-alias.sh.  It is highly
  recommended over just a raw 'eg status' or 'eg log' to review commits before
  attempting to test/push commits.

  NOTE: If you see any files/directories that are listed as 'unknown' returned
  from 'eg local-stat', then you will need to do an 'eg add' to track them or
  add them to an ignore list *before* you run the checkin-test.py script.
```

The eg script will not allow you to push if there are new 'unknown' files or uncommitted changes to tracked files.

3) Set up the checkin base build directory (first time only):

```
$ cd $PROJECT_HOME
$ echo CHECKIN >> .git/info/exclude
$ mkdir CHECKIN
$ cd CHECKIN
```

NOTE: You may need to set up some configuration files if CMake can not find the right compilers, MPI, and TPLs by default (see detailed documentation below).

NOTE: You might want to set up a simple shell driver script.

NOTE: You can set up a CHECKIN directory of any name in any location you want. If you create one outside of the main source dir, then you will not have to add the git exclude shown above.

4) Do the pull, configure, build, test, and push:

```
$ cd $PROJECT_HOME
$ cd CHECKIN
$ ../checkin-test.py -j4 --do-all --push
```

NOTE: The above will: a) pull updates from the global repo(s), b) automatically enable the correct packages, c) configure and build the right packages, d) run the tests, e) send you emails about what happened, f) do a final pull from the global repo, g) optionally amend the last local commit with the test results, and h) finally push local commits to the global repo(s) if everything passes.

NOTE: You must have installed the official versions of eg/git with the install-git.py script in order to run this script. If you don't, the script will die right away with an error message telling you what the problem is.

NOTE: The current branch will be used to pull and push to. A raw 'eg pull' is performed which will get all of the branches from 'origin'. This means that your current branch must be a tracking branch so that it will get updated correctly. The branch 'master' is the most common branch but release tracking branches are also common.

NOTE: You must not have any uncommitted changes or the 'eg pull && eg rebase --against origin' command will fail on the final pull/rebase before the push and therefore the whole script will fail. To run the script, you will may need to first use 'eg stash' to stash away your unstaged/uncommitted changes *before* running this script.

NOTE: You need to have SSH public/private keys set up to the remote repo machines for the git commands invoked in the script to work without you having to type a password.

NOTE: You can do the final push in a second invocation of the script with a follow-up run with --push and removing --do-all (it will remember the results from the build/test cases just ran). For more details, see detailed documentation below.

NOTE: Once you start running the checkin-test.py script, you can go off and do something else and just check your email to see if all the builds and

tests passed and if the push happened or not.

NOTE: The commands 'cmake', 'ctest', and 'make' must be in your default path before running this script.

For more details on using this script, see the detailed documentation below.

Detailed Documentation:

The following approximate steps are performed by this script:

---

1) Check to see if the local repo(s) are clean:

$ eg status

NOTE: If any modified or any unknown files are shown, the process will be aborted. The local repo(s) working directory must be clean and ready to push *everything* that is not stashed away.

2) Do a 'eg pull' to update the code (done if --pull or --do-all is set):

NOTE: If not doing a pull, use --allow-no-pull or --local-do-all.

3) Select the list of packages to enable forward/downstream based on the package directories where there are changed files (or from a list of packages passed in by the user).

NOTE: The automatic enable behavior can be overridden or modified using the options --enable-all-packages, --enable-packages, --disable-packages, and/or --no-enable-fwd-packages.

4) For each build/test case <BUILD_NAME> (e.g. MPI_DEBUG, SERIAL_RELEASE, extra builds specified with --extra-builds):

4.a) Configure a build directory <BUILD_NAME> in a standard way for all of the packages that have changed and all of the packages that depend on these packages forward/downstream. You can manually select which packages get enabled (see the enable options above). (done if --configure, --do-all, or --local-do-all is set.)

4.b) Build all configured code with 'make' (e.g. with -jN set through -j or --make-options). (done if --build, --do-all, or --local-do-all is set.)

4.c) Run all BASIC tests for enabled packages. (done if --test, --do-all, or --local-do-all is set.)

4.d) Analyze the results of the update, configure, build, and tests and send email about results. (emails only sent out if --send-emails-to != "")

5) Do final pull and rebase, append test results to last commit message, and push (done if --push is set)

5.a) Do a final 'eg pull' (done if --pull or --do-all is set)

```
  5.b) Do 'eg rebase --against origin/<current_branch>' (done if --pull or
  --do-all is set and --rebase is set)

    NOTE: The final 'eg rebase --against origin/<current_branch>' is
    required to avoid trivial merge commits that the global get repo
    will reject on the push.

  5.c) Amend commit message of the most recent commit with the summary of the
  testing performed.  (done if --append-test-results is set.)

  5.d) Push the local commits to the global repo (done if --push is set)


-------------------------------------------------------------------------------


The recommended way to use this script is to create a new base CHECKIN test
directory apart from your standard build directories such as with:

  $ $PROJECT_HOME
  $ mkdir CHECKIN
  $ echo CHECKIN >> .git/info/exclude

The most basic way to do pre-push testing is with:

  $ cd CHECKIN
  $ ../checkin-test.py --do-all [other options]

If your MPI installation, other compilers, and standard TPLs (i.e. BLAS and
LAPACK) can be found automatically, then this is all you will need to do.
However, if the setup cannot be determined automatically, then you can add a
set of CMake variables that will get read in the files:

  COMMON.config
  MPI_DEBUG.config
  SERIAL_RELEASE.config

(or whatever your standard --default-builds are).

Actually, for built-in build/test cases, skeletons of these files will
automatically be written out with typical CMake cache variables (commented
out) that you would need to set out.  Any CMake cache variables listed in
these files will be read into and passed on the configure line to 'cmake'.

WARNING: Please do not add any CMake cache variables than what are needed to
get the Primary Tested (PT) --default-builds builds to work.  Adding other
enables/disables will make the builds non-standard and break these PT builds.
The goal of these configuration files is to allow you to specify the minimum
environment to find MPI, your compilers, and the required TPLs (e.g. BLAS,
LAPACK, etc.).  If you need to fudge what packages are enabled, please use the
script arguments --enable-packages, --disable-packages,
--no-enable-fwd-packages, and/or --enable-all-packages to control this, not
the *.config files!

WARNING: Please do not add any CMake cache variables in the *.config files
that will alter what packages or TPLs are enabled or what tests are run.
Actually, the script will not allow you to change TPL enables in these
standard *.config files because to do so deviates from a consistent build
configuration for Primary Tested (PT) Code.
```

```
NOTE: All tentatively-enabled TPLs (e.g. Pthreads and BinUtils) are hard
disabled in order to avoid different behaviors between machines where they
would be enabled and machines where they would be disabled.

NOTE: If you want to add extra build/test cases that do not conform to
the standard build/test configurations described above, then you need
to create extra builds with the --extra-builds and/or
--st-extra-builds options (see below).

NOTE: Before running this script, you should first do an 'eg status' and 'eg
diff --name-status origin..' and examine what files are changed to make sure
you want to push what you have in your local working directory.  Also, please
look out for unknown files that you may need to add to the git repository with
'eg add' or add to your ignores list.  There cannot be any uncommitted changes
in the local repo before running this script.

NOTE: You don't need to run this script if you have not changed any files that
affect the build or the tests.  For example, if all you have changed are
documentation files, then you don't need to run this script before pushing
manually.

NOTE: To see detailed debug-level information, set
TRIBITS_CHECKIN_TEST_DEBUG_DUMP=ON in the env before running this script.


Common Use Cases (examples):
----------------------------

(*) Basic full testing with integrating with global repo(s) without push:

  ../checkin-test.py --do-all

  NOTE: This will result in a set of emails getting sent to your email address
  for the different configurations and an overall push readiness status email.

  NOTE: If everything passed, you can follow this up with a --push (see
  below).

(*) Basic full testing with integrating with local repo and push:

  ../checkin-test.py --do-all --push

(*) Push to global repo after a completed set of tests have finished:

  ../checkin-test.py [other options] --push

  NOTE: This will pick up the results for the last completed test runs with
  [other options] and append the results of those tests to the log of the most
  recent commit.

  NOTE: Take the action options for the prior run and replace --do-all with
  --push but keep all of the rest of the options the same.  For example, if
  you did:

    ../checkin-test.py --enable-packages=Blah --default-builds=MPI_DEBUG --do-all

  then follow that up with:

    ../checkin-test.py --enable-packages=Blah --default-builds=MPI_DEBUG --push
```

NOTE: This is a common use case when some tests are failing which aborted the initial push but you determine it is okay to push anyway and do so with --force-push.

(*) Test only the packages modified and not the forward dependent packages:

```
../checkin-test.py --do-all --no-enable-fwd-packages
```

NOTE: This is a safe thing to do when only tests in the modified packages are changed and not library code. This can speed up the testing process and is to be preferred over not running this script at all. It would be very hard to make this script automatically determine if only test code has changed because every package does not follow a set pattern for tests and test code.

(*) Run the the most important default (e.g. MPI_DEBUG) build/test only:

```
../checkin-test.py --do-all --default-builds=MPI_DEBUG
```

(*) The minimum acceptable testing when code has been changed:

```
../checkin-test.py \
  --do-all --enable-all-packages=off --no-enable-fwd-packages \
  --default-builds=MPI_DEBUG
```

NOTE: This will do only an MPI DEBUG build and will only build and run the tests for the packages that have directly been changed and not any forward packages. Replace "MPI_DEBUG" with whatever your most important default build is.

(*) Test only a specific set of packages and no others:

```
../checkin-test.py \
  --enable-packages=<P0>,<P1>,<P2> --no-enable-fwd-packages \
  --do-all
```

NOTE: This will override all logic in the script about which packages will be enabled and only the given packages will be enabled.

NOTE: You might also want to pass in --enable-all-packages=off in case the script wants to enable all the packages (see the output in the checkin-test.py log file for details) and you think it is not necessary to do so.

NOTE: Using these options is greatly preferred to not running this script at all and should not be any more expensive than the testing you would already do manually before a push.

(*) Test changes locally without pulling updates:

```
../checkin-test.py --local-do-all
```

NOTE: This will just configure, build, test, and send an email notification without updating or changing the status of the local git repo in any way and without any communication with the global repo. Hence, you can have uncommitted changes and still run configure, build, test without having to commit or having to stash changes.

NOTE: This is not a sufficient level of testing in order to push the changes to the global repo because you have not fully integrated your changes yet

with other developers. However, this would be a sufficient level of testing in order to do a commit on the local machine and then pull to a remote machine for further testing and a push (see below).

(*) Adding extra build/test cases:

Often you will be working on Secondary Tested (ST) Code or Experimental (EX) Code and want to include the testing of this in your pre-push testing process along with the standard --default-builds build/test cases which can only include Primary Tested (PT) Code. In this case you can run with:

```
../checkin-test.py --extra-builds=<BUILD1>,<BUILD2>,... [other options]
```

For example, if you have a build that enables the TPL CUDA you would do:

```
echo "
-DTPL_ENABLE_MPI:BOOL=ON
-DTPL_ENABLE_CUDA:BOOL=ON
" > MPI_DEBUG_CUDA.config
```

and then run with:

```
../checkin-test.py --extra-builds=MPI_DEBUG_CUDA --do-all
```

This will do the standard --default-builds (e.g. MPI_DEBUG and SERIAL_RELEASE) build/test cases along with your non-standard MPI_DEBUG_CUDA build/test case.

NOTE: You can disable the default build/test cases with --default-builds="". However, please only do this when you are not going to push because you need at least one default build/test case (the most important default PT case, e.g. MPI_DEBUG) to do a safe push.

(*) Including extra repos and extra packages:

You can also use the checkin-test.py script to continuously integrate multiple git repos containing add-on packages. To do so, just run:

```
../checkin-test.py --extra-repos=<REPO1>,<REPO2>,... [options]
```

NOTE: You have to create local commits in all of the extra repos where there are changes or the script will abort.

NOTE: Extra repos can be specified with more flexibility using the --extra-repos-file and --extra-repos-type arguments (also see --ignore-missing-extra-repos).

NOTE: Each of the last local commits in each of the changed repos will get amended with the appended summary of what was enabled in the build/test (if --append-test-results is set).

(*) Performing a remote test/push:

If you develop on a slow machine like your laptop, doing an appropriate level of testing can take a long time. In this case, you can pull the changes to another faster remote workstation and do a more complete set of tests and push from there. If you are knowledgeable with git, this will be easy and natural to do, without any help from this script. However, this script can still help and automate the steps and can do so in one command invocation on the part of the developer.

```
  On your slow local development machine 'mymachine', do the limited testing
  with:

    ../checkin-test.py --do-all --no-enable-fwd-packages

  On your fast remote test machine, do a full test and push with:

    ../checkin-test.py \
      --extra-pull-from=<remote-name>:master \
      --do-all --push

  where <remote-name> is a git repo pointing to
  mymachine:/some/dir/to/your/src:master (see 'git help remote').

  NOTE: You can of course adjust the packages and/or build/test cases that get
  enabled on the different machines.

  NOTE: Once you invoke the checkin-test.py script on the remote test machine
  and it has pulled the commits from mymachine, then you can start changing
  files again on your local development machine and just check your email to
  see what happens on the remote test machine.

  NOTE: If something goes wrong on the remote test machine, you can either
  work on fixing the problem there or you can fix the problem on your local
  development machine and then do the process over again.

  NOTE: If you alter the commits on the remote machine (such as squashing
  commits), you will have trouble merging back on our local machine.
  Therefore, if you have to to fix problems, make new commits and don't alter
  the ones you pulled from your local machine (but rebasing them should be
  okay as long as the local commits on mymachine are not pushed to other
  repos).

  NOTE: Git will resolve the duplicated commits when you pull the commits
  pushed from the remote machine.  Git knows that the commits are the same and
  will do the right thing when rebasing (or just merging).

(*) Check push readiness status:

  ../checkin-test.py

  NOTE: This will examine results for the last testing process and send out an
  email stating if the a push is ready to perform or not.

(*) See the default option values without doing anything:

  ../checkin-test.py --show-defaults

  NOTE: This is the easiest way to figure out what all of the default options
  are.

Hopefully the above documentation, the example use cases, the documentation of
the command-line arguments below, and some experimentation will be enough to
get you going using this script for all of your pre-push testing and pushes.
If that is not sufficient, send email to your development support team to ask
for help.


Handling of PT, ST, and EX Code in built-in and extra builds:
```

```
------------------------------------------------------------------

This script will only process PT (Primary Tested) packages in the
--default-builds (e.g. MPI_DEBUG and SERIAL_RELEASE) builds.  This is to avoid
problems of side-effects of turning on ST packages that would impact PT
packages (e.g. an ST package getting enabled that enables an ST TPL which
turns on support for that TPL in a PT package producing different code which
might work but the pure PT build without the extra TPL may actually be broken
and not know it).  Therefore, any non-PT packages that are enabled (either
implicitly through changed files or explicitly by listing in --enable-packages)
will be turned off in the --default-builds builds.  If none of the enabled
packages are PT, then they will all be disabled and the --default-builds
builds will be skipped.

In order to better support the development of ST and EX packages, this script
allows you to define some extra builds that will be invoked and used to
determine overall pass/fail before a potential push.  The option
--st-extra-builds is used to specify extra builds that will test ST packages
(and also PT packages if any are enabled).  If only PT packages are enabled
then the builds specified in --st-extra-builds will still be run.  The
reasoning is that PT packages may contain extra ST features and therefore if
the goal is to test these ST builds it is desirable to also run these builds
because they also my impact downstream ST packages.

Finally, the option --extra-builds will test all enabled packages, including
EX packages, regardless of their test group.  Therefore, when using
--extra-builds, be careful that you watch what packages are enabled.  If you
change an EX package, it will be enabled in --extra-builds builds.

A few use cases might help better demonstrate the behavior.  Consider
the following input arguments specifying extra builds

  --st-extra-builds=MPI_DEBUG_ST --extra-builds=INTEL_DEBUG

with the packages Techos, Phalanx, and Meros where Teuchos is PT, Phalanx is
ST, and Meros is EX.

Here is what packages would be enabled in each of the builds:

  --default-builds=MPI_DEBUG,SERIAL_RELEASE \
  --st-extra-builds=MPI_DEBUG_ST \
  --extra-builds=INTEL_DEBUG

and which packages would be excluded:

A) --enable-packages=Teuchos:
   MPI_DEBUG:       [Teuchos]
   SERIAL_RELEASE:  [Teuchos]
   MPI_DEBUG_ST:    [Teuchos]
   INTEL_DEBUG:     [Teuchos]     Always enabled!

B) --enable-packages=Phalanx:
   MPI_DEBUG:       []            Skipped, no PT packages!
   SERIAL_RELEASE:  []            Skipped, no PT packages!
   MPI_DEBUG_ST:    [Phalanx]
   INTEL_DEBUG:     [Phalanx]

C) --enable-packages=Meros:
   MPI_DEBUG:       []            Skipped, no PT packages!
   SERIAL_RELEASE:  []            Skipped, no PT packages!
```

```
   MPI_DEBUG_ST:    []            Skipped, no PT or ST packages!
   INTEL_DEBUG:     [Meros]

D) --enable-packages=Teuchos,Phalanx:
   MPI_DEBUG:       [Teuchos]
   SERIAL_RELEASE:  [Teuchos]
   MPI_DEBUG_ST:    [Teuchos,Phalanx]
   INTEL_DEBUG:     [Teuchos,Phalanx]

E) --enable-packages=Teuchos,Phalanx,Meros:
   MPI_DEBUG:       [Teuchos]
   SERIAL_RELEASE:  [Teuchos]
   MPI_DEBUG_ST:    [Teuchos,Phalanx]
   INTEL_DEBUG:     [Teuchos,Phalanx,Meros]

The --extra-builds=INTEL_DEBUG build is always performed with all of the
enabled packages.  This logic given above must be understood in order to
understand the output given in the script.


Conventions for Command-Line Arguments:
---------------------------------------

The command-line arguments are segregated into three broad categories: a)
action commands, b) aggregate action commands, and c) others.

a) The action commands are those such as --build, --test, etc. and are shown
with [ACTION] in their documentation.  These action commands have no off
complement.  If the action command appears, then the action will be performed.

b) Aggregate action commands such as --do-all and --local-do-all turn on sets
of other action commands and are shown with [AGGR ACTION] in their
documentation.  The sub-actions that these aggregate action commands turn on
cannot be disabled with other arguments.

c) Other arguments are those that are not marked with [ACTION] or [AGGR
ACTION] tend to either pass in data and turn control flags on or off.


Exit Code:
---------

This script returns 0 if the actions requested are successful.  This does not
necessarily imply that it is okay to do a push.  For example, if only --pull
is passed in and is successful, then 0 will be returned but that does *not*
mean that it is okay to do a push.  A 0 return value is a necessary but not
sufficient condition for readiness to push.



Options:
  -h, --help            show this help message and exit
  --project-configuration=PROJECTCONFIGURATION
                        Custom file to provide configuration defaults for the
                        project.
  --show-defaults       Show the default option values and do nothing at all.
  --project-name=PROJECTNAME
                        Set the project's name. This is used to locate various
                        files.
  --eg-git-version-check
```

```
                      Enable automatic check for the right versions of eg
                      and git. [default]
--no-eg-git-version-check
                      Do not check the versions of eg and git, just trust
                      they are okay.
--src-dir=SRCDIR      The source base directory for code to be tested.
--trilinos-src-dir=SRCDIR
                      [DEPRECATED] Use --src-dir instead. This argument is
                      for backwards compatibility only.
--default-builds=DEFAULTBUILDS
                      Comma separated list of builds that should always be
                      run by default.
--extra-repos-file=EXTRAREPOSFILE
                      File path to an extra repositories list file.  If set
                      to 'project', then
                      <project_dir>/cmake/ExtraRepositoriesList.cmake is
                      read.  See the argument --extra-repos for details on
                      how this list is used (default empty '')
--extra-repos-type=EXTRAREPOSTYPE
                      The test type of repos to read from
                      <extra_repos_file>. Choices = ('', 'Continuous',
                      'Nightly', 'Experimental').  [default = '']
--extra-repos=EXTRAREPOS
                      List of comma separated extra repositories containing
                      extra  packages that can be enabled.  The order these
                      repos is listed in not important.  This option
                      overrides --extra-repos-file.
--ignore-missing-extra-repos
                      If set, then extra repos read in from
                      <extra_repos_file> will be ignored and removed from
                      list.  This option is not applicable if
                      <extra_repos_file>==' or <extra_repos_type>==''.
--require-extra-repos-exist
                      If set, then all listed extra repos must exist or the
                      script will exit. [default]
--with-cmake=WITHCMAKE
                      CMake executable to use with cmake -P scripts
                      internally (only set by unit testing code).
--skip-deps-update    If set, skip the update of the dependency XML file.
                      If the package structure has not changed since the
                      last invocation, then it is safe to use this option.
--enable-packages=ENABLEPACKAGES
                      List of comma separated packages to test changes for
                      (example, 'Teuchos,Epetra').  If this list of packages
                      is empty, then the list of packages to enable will be
                      determined automatically by examining the set of
                      modified files from the version control update log.
--disable-packages=DISABLEPACKAGES
                      List of comma separated packages to explicitly disable
                      (example, 'Tpetra,NOX').  This list of disables will
                      be appended after all of the listed enables no mater
                      how they are determined (see --enable-packages
                      option).  NOTE: Only use this option to remove
                      packages that will not build for some reason.  You can
                      disable tests that run by using the CTest option -E
                      passed through the --ctest-options argument in this
                      script.
--enable-all-packages=ENABLEALLPACKAGES
                      Determine if all packages are enabled 'on', or 'off',
                      or let other logic decide 'auto'.  Setting to 'off' is
```

```
                      appropriate when the logic in this script determines
                      that a global build file has changed but you know that
                      you don't need to rebuild every package for a
                      reasonable test.  Setting --enable-packages
                      effectively disables this option.  NOTE: Setting this
                      to 'off' does *not* stop the forward enabling of
                      downstream packages for packages that are modified or
                      set by --enable-packages. Choices = ('auto', 'on',
                      'off').  [default = 'auto']
--enable-fwd-packages
                      Enable forward packages. [default]
--no-enable-fwd-packages
                      Do not enable forward packages.
--continue-if-no-updates
                      If set, then the script will continue if no updates
                      are pulled from any repo. [default]
--abort-gracefully-if-no-updates
                      If set, then the script will abort gracefully if no
                      updates are pulled from any repo.
--continue-if-no-changes-to-push
                      If set, then the script will continue if no changes to
                      push from any repo. [default]
--abort-gracefully-if-no-changes-to-push
                      If set, then the script will abort gracefully if no
                      changes to push from any repo.
--continue-if-no-enables
                      If set, then the script will continue if no packages
                      are enabled. [default]
--abort-gracefully-if-no-enables
                      If set, then the script will abort gracefully if no
                      packages are enabled.
--extra-cmake-options=EXTRACMAKEOPTIONS
                      Extra options to pass to 'cmake' after all other
                      options. This should be used only as a last resort.
                      To disable packages, instead use --disable-packages.
                      To change test categories, use --test-categories.
--test-categories=TESTCATEGORIES
                      . Change the test categories.  Can be 'BASIC',
                      'CONTINUOUS',  'NIGHTLY', or 'WEEKLY' (default
                      'BASIC').
-j OVERALLNUMPROCS    The options to pass to make and ctest (e.g. -j4).
--make-options=MAKEOPTIONS
                      The options to pass to make (e.g. -j4).
--ctest-options=CTESTOPTIONS
                      Extra options to pass to 'ctest' (e.g. -j2).
--ctest-timeout=CTESTTIMEOUT
                      timeout (in seconds) for each single 'ctest' test
                      (e.g. 180 for three minutes).
--show-all-tests      Show all of the tests in the summary email and in the
                      commit message summary (see --append-test-results).
--no-show-all-tests   Don't show all of the test results in the summary
                      email. [default]
--without-default-builds
                      Skip the default builds (same as --default-builds='').
                      You would use option along with --extra-
                      builds=BUILD1,BUILD2,... to run your own local custom
                      builds.
--st-extra-builds=STEXTRABUILDS
                      List of comma-separated ST extra build names.  For
                      each of the build names in --st-extra-
```

```
                      builds=<BUILD1>,<BUILD2>,..., there must be a file
                      <BUILDN>.config in the local directory along side the
                      COMMON.config file that defines the special build
                      options for the extra build.
--ss-extra-builds=SSEXTRABUILDS
                      DEPRICATED!  Use --st-extra-builds instead!.
--extra-builds=EXTRABUILDS
                      List of comma-separated extra build names.  For each
                      of the build names in --extra-
                      builds=<BUILD1>,<BUILD2>,..., there must be a file
                      <BUILDN>.config in the local directory along side the
                      COMMON.config file that defines the special build
                      options for the extra build.
--send-email-to=SENDEMAILTO
                      List of comma-separated email addresses to send email
                      notification to after every build/test case finishes
                      and at the end for an overall summary and push status.
                      By default, this is the email address you set for git
                      returned by 'git config --get user.email'.  In order
                      to turn off email notification, just set --send-email-
                      to='' and no email will be sent.
--skip-case-send-email
                      If set then if a build/test case is skipped for some
                      reason (i.e. because no packages are enabled) then an
                      email will go out for that case. [default]
--skip-case-no-email  If set then if a build/test case is skipped for some
                      reason (i.e. because no packages are enabled) then no
                      email will go out for that case. [default]
--send-email-for-all  If set, then emails will get sent out for all
                      operations. [default]
--send-email-only-on-failure
                      If set, then emails will only get sent out for
                      failures.
--send-email-to-on-push=SENDEMAILTOONPUSH
                      List of comma-separated email addresses to send email
                      notification to on a successful push.  This is used to
                      log pushes to a central list.  In order to turn off
                      this email notification, just set --send-email-to-on-
                      push='' and no email will be sent to these email
                      lists.
--force-push          Force the local push even if there are build/test
                      errors. WARNING: Only do this when you are 100%
                      certain that the errors are not caused by your code
                      changes.  This only applies when --push is specified
                      and this script.
--no-force-push       Do not force a push if there are failures. [default]
--do-push-readiness-check
                      Check the push readiness status at the end and send
                      email if not actually pushing. [default]
--skip-push-readiness-check
                      Skip push status check.
--rebase              Rebase the local commits on top of origin/master
                      before amending the last commit and pushing.  Rebasing
                      keeps a nice linear commit history like with CVS or
                      SVN and will work perfectly for the basic workflow of
                      adding commits to the 'master' branch and then syncing
                      up with origin/master before the final push. [default]
--no-rebase           Do not rebase the local commits on top of
                      origin/master before amending the final commit and
                      pushing.  This allows for some more  complex workflows
```

```
                      involving local branches with multiple merges.
                      However, this will result in non-linear history and
                      will allow for trivial merge commits with
                      origin/master to get pushed.  This mode should only be
                      used in cases where the rebase mode will not work or
                      when it is desired to use a merge commit to integrate
                      changes on a branch that you wish be able to easily
                      back out.  For sophisticated users of git, this may in
                      fact be the prefered mode.
--append-test-results
                      Before the final push, amend the most recent local
                      commit by appending a summary of the test results.
                      This provides a record of what builds and tests were
                      performed in order to test the local changes.  This is
                      only  performed if --push is also set.  NOTE: If the
                      same local commit is amended more than once, the prior
                      test summary sections will be overwritten with the
                      most recent test results from the current run.
                      [default]
--no-append-test-results
                      Do not amend the last local commit with test results.
                      NOTE: If you have uncommitted local changes that you
                      do not want this script to commit then you must select
                      this option to avoid this last amending commit.
--extra-pull-from=EXTRAPULLFROM
                      Optional extra git pull '<repository>:<branch>' to
                      merge in changes from after pulling in changes from
                      'origin'.  This option uses a colon with no spaces in
                      between <repository>:<branch>' to avoid issues with
                      passing arguments with spaces.  For example --extra-
                      pull-from=machine:/base/dir/repo:master.  This extra
                      pull is only done if --pull is also specified.  NOTE:
                      when using --extra-repo=REPO1,REPO2,... the
                      <repository> must be a named repository that is
                      present in all of the git repos or it will be an
                      error.
--allow-no-pull       Allowing for there to be no pull performed and still
                      doing the other actions.  This option is useful for
                      testing against local changes without having to get
                      the updates from the global repo.  However, if you
                      don't pull, you can't push your changes to the global
                      repo.  WARNING: This does *not* stop a pull attempt
                      from being performed by --pull or --do-all!
--wipe-clean          [ACTION] Blow existing build directories and
                      build/test results.  The action can be performed on
                      its own or with other actions in which case the wipe
                      clean will be performed before any other actions.
                      NOTE: This will only wipe clean the builds that are
                      specified and will not touch those being ignored (e.g.
                      SERIAL_RELEASE will not be removed if --default-
                      builds=MPI_DEBUG is specified).
--pull                [ACTION] Do the pull from the default (origin)
                      repository and optionally also merge in changes from
                      the repo pointed to by --extra-pull-from.
--configure           [ACTION] Do the configure step.
--build               [ACTION] Do the build step.
--test                [ACTION] Do the running of the enabled tests.
--local-do-all        [AGGR ACTION] Do configure, build, and test with no
                      pull (same as setting --allow-no-pull ---configure
                      --build --test).  This is the same as --do-all except
```

```
                        it does not do --pull and also allows for no pull.
  --do-all              [AGGR ACTION] Do update, configure, build, and test
                        (same as --pull --configure --build --test).  NOTE:
                        This will do a --pull regardless if --allow-no-pull is
                        set or not.  To avoid the pull, use --local-do-all.
  --push                [ACTION] Push the committed changes in the local repo
                        into to global repo 'origin' for the current branch.
                        Note: If you have uncommitted changes this command
                        will fail.  Note: You must have SSH public/private
                        keys set up with the origin machine (e.g.
                        software.sandia.gov) for the push to happen without
                        having to type your password.
  --execute-on-ready-to-push=EXECUTEONREADYTOPUSH
                        [ACTION] A command to execute on successful execution
                        and 'READY TO PUSH' status from this script.  This can
                        be used to do a remote SSH invocation to a remote
                        machine to do a remote pull/test/push after this
                        machine finishes.
```

13.4 egdist --help

Below is a snapshot of the output from `egdist --help`. For more details on the usage of `egdist`, see Multi-Repository Support and Multi-Repository Development Workflow.

```
Usage: egdist [egdist options] [OPTIONS]

Run eg/git recursively over extra repos

Instead of typing

   $ eg [OPTIONS]

type:

  $ egdist [egdist options] [OPTIONS]

This will distribute git options across all git repos listed, including the
base git repo.  The options in [egdist options] are prefixed with '--dist-'
and are are pulled out before running eg/git on the underlying executable.
See --help to see the egdist options.

If --dist-extra-repos="", then the list of extra repos will be read from the
file .egdist.  The format of this file is to have one repo name per line as
in:

   Repo1
   Repo2
   Repo3
   ...

NOTE: If any extra repository does not exist, then it will be ignored and no
output will be produced.  Therefore, be careful to manually verify that the
script recognizes the repositories that you list.  The best way to do that is
to type 'egdist status'.

NOTE: This script has no other dependencies so it can be copied and moved
anywhere and used.

TIPS:
```

- Use 'egdist --no-pager <command> ...' to get the full output from all extra repos in one contiguous stream which can then be piped to 'less' or to a file to be read with emacs or vi (may also want to use --dist-no-color as well).

- 'egdist --help' will run egdist help, not eg/git help. If you want eg/git help, run raw 'eg --help'.

- By default, egdist will use 'eg' in the environment. If it can't find 'eg' in the environment, it will try to use 'eg' in the same directory as 'egdist' (which is the case in the home tribits directory). If it can't find this 'eg' it will look for 'git' in the environment. If it can't find 'git' it will require that the user specify the eg/git command to run with --with-eg-git=<the command>.

- To exclude processing either the base git repo and/or git repos listed in .egdist, pass in --dist-not-base-repo and/or --dist-not-extra-repos=RepoX,RepoZ,... The provides complete control over what git repos the given command is run on.

REPO VERSION FILES:

This script also supports the options --dist-version-file=<versionFile> and --dist-version-file2=<versionFile2> which are used to provide different SHA1 versions for each repo. Each of these version files is expected to represent a compatible set of versions of the repos.

The format of these repo version files is a follows:

```
------------------------------------------------------
** Base Git Repo: SomeBaseRepo
e102e27 [Mon Sep 23 11:34:59 2013 -0400] <author1@someurl.com>
First summary message
** Git Repo: ExtraRepo1
b894b9c [Fri Aug 30 09:55:07 2013 -0400] <author2@someurl.com>
Second summary message
** Git Repo: ExtraRepo2
97cf1ac [Thu Dec 1 23:34:06 2011 -0500] <author3@someurl.com>
Third summary message
...
------------------------------------------------------
```

Each repository entry can have a summary message or not (i.e. use two or three lines per repo in the file). A compatible repo version file can be generated with this script using, for example, using:

```
$ egdist log -1 --pretty=format:"%h [%ad] <%ae>%n%s" \
  | grep -v "^$" &> RepoVersion.txt
```

using three lines per repo, or just:

```
$ egdist log -1 --pretty=format:"%h [%ad] <%ae>" \
  | grep -v "^$" &> RepoVersion.txt
```

using two lines per repo in the output file.

This allows checking out consistent versions of the repos, diffing two consistent versions of the repos, etc.

```
To checkout an older set of consistent versions of the set of repos, use:

  $ egdist fetch origin
  $ egdist --dist-version-file=RepoVersion.SomeDate.txt checkout _VERSION_

The '_VERSION_' string will be replaced with the SHA1 for each of the repos.

To tag and branch the set of repos using a consistent set of versions, use:

  $ egdist --dist-version-file=RepoVersion.SomeDate.txt \
    tag -a some_tag _VERSION_

To diff two sets of versions of the repos, for example, use:

  $ egdist --dist-version-file=RepoVersion.NewerDate.txt \
     --dist-version-file2=RepoVersion.OlderDate.txt \
     diff _VERSION_ ^_VERSION2_

Here, _VERSION_ is replaced by the SHA1s listed in RepoVersion.NewerDate.txt
and _VERSION2_ is replaced by the SHA1s listed in RepoVersion.OlderDate.txt.

One can construct any git commit taking one or two different repo version
arguments (SHA1s).

Note that the set of repos listed in the RepoVersion.txt file must be a
super-set of those processed by this script or an error will occur and the
script will stop.


Options:
  -h, --help            show this help message and exit
  --with-eg-git=EGGIT   The (path) to the eg/git executable to use for each
                        git repo command (default='eg')
  --dist-extra-repos=EXTRAREPOS
                        Comma seprated list of extra repos to forward eg/git
                        commands to.  If the list is empty, it will look for a
                        file called .egdist to get the list of extra repos
                        seprated by newlines.
  --dist-not-extra-repos=NOTEXTRAREPOS
                        Comma seprated list of extra repos to *not* forward
                        eg/git commands to.  This removes any repos from being
                        processed that would otherwise be.
  --dist-not-base-repo  If set, don't pass the eg/git conmmand on to the base
                        git repo.
  --dist-version-file=VERSIONFILE
                        Path to a file contains a list of extra repo
                        directories and git versions (replaces _VERSION_).
  --dist-version-file2=VERSIONFILE2
                        Path to a second file contains a list of extra repo
                        directories and git versions (replaces _VERSION2_).
  --dist-no-color       If set, don't use color in the output for egdist
                        (better for output to a file).
  --dist-debug          If set, then debugging info is printed.
  --dist-no-opt         If set, then no eg/git commands will be run but
                        instead will just be printed.
```