TriBITS Developers Guide

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

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.

TriBITS is a fairly extensive framework that is build on CMake/CTest/CPack/CDash which in of itself is a very extensive system of software and tools. The most important thing to remember is that software project that use TriBITS are really just CMake projects. TriBITS makes no attent to hide that 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. In particular, CMake is a Turning complete programming lanauge 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, 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.

2 TriBITS Developer and User Roles

There are approximately five different types roles with respect to TriBITS. These different roles require different levels of expertise and knowlege 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 knolwege 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 and running tests with `ctest`. The proper reference for a TriBITS Project User is the Project-Specific Build Quick Reference. Also, the TriBITS Overview document may be of some help also. A TriBITS project user does not need to know anything about the CMake langauge itself or any of the TriBITS macros or functions described in TriBITS Macros and Functions or really anything else described in this current document.

A **TriBITS Project Developer** is someone who contributes to a software project that uses TriBITS. They will add source files, libraries and exectuables, add 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 need to know a subset of

the TriBITS Macros and Functions defined in this document. A slightly more sophsiticated TriBITS Project Developer will also add new packages, add new package dependencies, and define new TPLs. This current TriBITS Developers Guide 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 scrach, 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 as is 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 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 functionlity while maintaining backward compatibility, how to add new unit tests to the TriBITS system, 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 consusted 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 TriBIITS 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. 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 System Archetects (i.e. those would make the decision to adopt TriBITS), and new TriBITS System Developers to help extend and maintain the system. As TriBITS matures and its development stabilizes, the need for a TriBITS System Architect will be diminished.

So depending on the particular role that a reader falls into, this documnet may or may not be necessary but instead the TriBITS Overview or the <Project>BuildQuickRef documents may be more appropriate.

3 Brief CMake Language Tutorial

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 sytem, are due to a basic lack of knowlege 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 offical CMake reference book. Therefore, 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 overivew of the CMake langauge and a description of its unique features 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 itself is implemented in) is a fairly simply programming languge with fairly simple rules (for the most part). However, compared to other programming lanuages, there are a few peculiar aspects to the CMake language like strange varible scoping rules, arguments to macros and function, 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.

The first thing to understand about the CMake language is that everthing 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 with elements separated by semi-colons `"<str0>;<str1>;..."`. CMake is a bit odd in how it deals with these arrays (which 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 escapes the interpetation of space characters as array element boundaries. Quotes around arguments with no spaces does nothing (as seen above). 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\"`].

Varibles are set using a built-in CMke function that just takes string arguments like:

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

In CMake, the above is idential, 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 varible 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 argumnets are interpreted as the names of variables.

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..

CMake offers a rich assortment of built-in functions for doing all sorts of things. As part of these functions are the built-in `MACRO()` and the `FUNCTION()` functions 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()).

Varible names are translated into their stored values using `${SOME_VARIABLE}`. The value that is extracted depends on if the varible 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 varibles that are created by macros in that base local scope are seen across an entire project but are *not* persistent across `cmake` configure invocations.

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

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

produces `SOME_VAR='a;c'` and *not* `'a;;c'`! The same thing occurs when an empty varible is dereferenced 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 varible 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 varibles that might be undefined or empty. For example, for the macro:

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

if someone trys to call it with:

```
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 varible 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 mispell 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 mispelled variable would be caught.

CMake langauge 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 bulit-in *SET()*' function. The convention in TriBITS is to use all caps for functions and macros (was adopted by following the conventions used in the early versions of TriBITS, see History of TriBITS). The convention in CMake literature from Kitware seems to use lower-case letters for functions and macros.

- The names of CMake varables (local or cache/global) are *case sensitive*! That is, `SOME_VAR` and `some_var` are *different* variables. Built-in CMake varibles tend use all caps with underscores (e.g. `CMAKE_CURRENT_SOURCE_DIR`) but other built-in CMake varibles tend to use mixed case wtih underscores (e.g. `CMAKE_Fortran_FLAGS`). TriBITS tends to use a similar naming convention where most varibles have mostly upper-case letters except for 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 senstivity rules for varibles verses functions. However, because we must parse macro and function arguments when writing user-defined macros and functions, it is a good thing that CMake varibles are case sensitive. Case insenstivity 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 defaults a global scope (i.e. "cache" varibles) 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. It is not just varibles 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 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.

4 Structure of a TriBITS Project

???

5 Processing of TriBITS Files

???

6 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.

ToDo: Document what parameters influence the entire TriBITS project, what parameters can have project-specific defaults, etc.

7 Automated testing

ToDo: Fill in!

8 TriBITS Project Development Workflow

ToDo: Fill in!

9 Project-Specific Build Quick Reference

ToDo: Describe <Project>BuildQuickRef document!

10 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. These are listed in approximately the order they will be encounted in a project or packages `CMakeLists.txt` and other files.

10.1 TRIBITS_PROJECT()

Defines and processes a TriBITS project.

Requires that `PROJECT_NAME` be defined before calling this macro.

Note, this is just a shell of a macro that calls the real implementation. This allows someone to set `${PROJECT_NAME}_TRIBITS_DIR` in the env and point to a different Tribits implementation to test before snapshoting.

ToDo: Give documentation!

10.2 TRIBITS_DEFINE_REPOSITORY_PACKAGES()

Define the set of packages for a given TriBIT repo. This macro is typically called from inside of a PackagesList.cmake file for a given TriBITS repo.

Usage:

```
TRIBITS_DEFINE_REPOSITORY_PACKAGES(
   <pkg0>  <pkg0_dir>  <pkg0_classifications>
   <pkg1>  <pkg1_dir>  <pkg1_classifications>
   ...
   )
```

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 three columns:

- **PACKAGE** (1st column): 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_]*`).
- **DIR** (2nd column): 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).

- **CLASSIFICATION** (3nd column): 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".

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_CLASSIFICATI`
  If you misspell the name of the macro, it is an immediate error in CMake.

10.3 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_classification>
  <tpl1_name>   <tpl1_findmod>  <tpl1_classification>
  ...
  )
```

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 three columns:

- **TPL** (1st column): 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_]*`).
- **FINDMOD** (2nd column): The relative directory for the find module, usually with the name `FindTPL<TPL_NAME>.cmake`. This 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().

- **CLASSIFICATION** (3rd column): 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".

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.

10.4 TRIBITS_DEFINE_PACKAGE_DEPENDENCIES()

Define the dependenices for a given TriBITS SE package (i.e. a top-level package or a subpackage).

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 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 the future.

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 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_PACKAGE
  - **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 ???).

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.

10.5 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.

10.6 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.

10.7 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.

10.8 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_W

`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.

10.9 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.

10.10 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().

10.11 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..

10.12 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..

10.13 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 it will be set to `ON` only if `ST` code is enabled, otherwise it is set to `OFF`. In release mode 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.

10.14 TRIBITS_CONFIGURE_FILE()

Macro that configures the package's main config.h file

ToDo: Document everything this macro does!

10.15 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.

10.16 TRIBITS_ADD_LIBRARY()

Function used to add a CMake library target using ADD_LIBRARY().

Usage:

```
TRIBITS_ADD_LIBRARY(
  <libName>
  [HEADERS <h1> <h> ...]
  [NOINSTALLHEADERS <nih1> <hih2> ...]
  [SOURCES <src1> <src2> ...]
  [DEPLIBS <deplib1> <deplib2> ...]
  [IMPORTEDLIBS <ideplib1> <ideplib2> ...]
  [DEFINES -D<define1> -D<define2> ...]
  [TESTONLY]
  [NO_INSTALL_LIB_OR_HEADERS]
  [CUDALIBRARY]
  )
```

ToDo: Document each argument!

This function has a number of side-effects after it finishes running:

- An install target for the library is created by default using INSTALL(TARGETS <libName> ...). However, this install target will not get created if ${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS=FALSE and BUILD_SHARD_LIBS=OFF. However, 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.
- An install target for the headers listed in HEADERS will get created using INSTALL(FILES <h1> <h2> ...). NOTE: An install target will *not* get created for the headers listed in NOINSTALLHEADERS.

ToDo: Document other side-effects!

NOTE: IF the library is added, a CMake library target <libName> gets created through calling the build-in command ADD_LIBRARY(<libName> ...).

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()).

10.17 TRIBITS_ADD_EXECUTABLE()

Function used to create an executable (typically for a test or example), using the built-in CMake comamnd `ADD_EXECUTABLE()`.

Usage:

```
TRIBITS_ADD_EXECUTABLE(
  <exeRootName>  [NOEXEPREFIX]  [NOEXESUFFIX]
  SOURCES <src1> <src2> ...
  [CATEGORIES <category1>  <category2> ...]
  [HOST <host1> <host2> ...]
  [XHOST <host1> <host2> ...]
  [HOSTTYPE <hosttype1> <hosttype2> ...]
  [XHOSTTYPE <hosttype1> <hosttype2> ...]
  [DIRECTORY <dir> ]
  [DEPLIBS <lib1> <lib2> ... ]
  [COMM [serial] [mpi] ]
  [LINKER_LANGUAGE [C|CXX|Fortran] ]
  [ADD_DIR_TO_NAME]
  [DEFINES -DS<someDefine>]
  [INSTALLABLE]
  )
```

Formal Arguments:

`<exeRootName>`

The base name of the exectuable and CMake target.

ToDo: Document other arguments!

Executable and Target Name:

By default, the actual name of the executable and target will be:

```
${PACKAGE_NAME}_<exeRootName>${${PROJECT_NAME}_CMAKE_EXECUTABLE_SUFFIX}
```

If the option `NOEXEPREFIX` is pased in, the prefix `${PACKAGE_NAME}_` is removed. If the option `NOEXESUFFIX` is passed in, the suffix `${${PROJECT_NAME}_CMAKE_EXECUTABLE_SUFFIX}` is removed.

The reason that a default prefix is appended to the executable 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. Also, if `INSTALLABLE` is set and this executable gets installed into the `<install>/bin/` directory, then this prefix helps to avoid clashing with executables installed by other packages.

Postcondition:

ToDo: Document post conditions!

10.18 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.

10.19 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 "<arg1> <arg2> ..." "<arg3> <arg4> ..." ...
    | POSTFIX_AND_ARGS_0 <postfix> <arg1> <arg2> ...
      POSTFIX_AND_ARGS_1 ... ]
  [COMM [serial] [mpi]]
  [NUM_MPI_PROCS <numProcs>]
  [CATEGORIES <category1>  <category2> ...]
  [HOST <host1> <host2> ...]
  [XHOST <host1> <host2> ...]
  [HOSTTYPE <hosttype1> <hosttype2> ...]
  [XHOSTTYPE <hosttype1> <hosttype2> ...]
  [STANDARD_PASS_OUTPUT
    | PASS_REGULAR_EXPRESSION "<regex1>;<regex2>;..."]
  [FAIL_REGULAR_EXPRESSION "<regex1>;<regex2>;..."]
  [WILL_FAIL]
  [ENVIRONMENT <var1>=<value1> <var2>=<value2> ...]
  )
```

Formal Arguments:

`<exeRootName>`

The name of the exectuble or path to the exectuable to run for the test (see Determining the Exectuable or Command to Run). This name is also the default root name for the test (see Determining the Full Test Name).

`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 "<arg1> <arg2> ..." "<arg3> <arg4> ..." ...`

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.

`POSTFIX_AND_ARGS_<IDX> <postfix> <arg1> <arg2> ...`

If specified, gives a sequence of sets of test postfix names and arguments lists for different tests. For example, a set of three different tests with argument lists can be specified as:

```
POSTIFX_AND_ARGS_0 postfix1 --arg1 --arg2="dummy"
POSTIFX_AND_ARGS_1 postfix2  --arg2="fly"
POSTIFX_AND_ARGS_3 postfix3  --arg2="bags"
```

This will create three different test cases with the postfix names `postfix1`, `postfix2`, and `postfix3`. 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.

`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 <host1> <host2> ...`

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 <host1> <host2> ...`

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 <category1> <category2> ...`

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 <hosttype1> <hosttype2> ...`

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` etc.

`XHOSTTYPE <hosttype1> <hosttype2> ...`

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 "<regex1>;<regex2>;..."`

If specified, then a test will be assumed to pass only if one of the regular expressions `<regex1>`, `<regex2>` 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 "<regex1>;<regex2>;..."`

If specified, then a test will be assumed to fail if one of the regular expressions `<regex1>`, `<regex2>` 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 <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`.

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 and determine pass/fail. Therefore, this wrapper funtion does not provide any fundamentally new features that is avaiable in the basic usage if CMake/CTes. However, this wrapper function takes care of many of the details and boiler-plate CMake code that it takes to add such as 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:

This funtion is primarily designed to make it easy to run tests for exectaubles built usign 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:

```
${PACKAGE_NAME}_<exeRootName>${${PROJECT_NAME}_CMAKE_EXECUTABLE_SUFFIX}
```

which is (by no coincidence) idential to how it is selected in `TRIBITS_ADD_EXECUTABLE()` (see Executable and Target Name).

If `NONEXEPREFIX` is passed in, the prefix `${PACKAGE_NAME}_` is not prepended to the assumed name. If `NOEXESUFFIX` is passed in, then `${${PROJECT_NAME}_CMAKE_EXECUTABLE_SUFFIX}` is not assumed to be appended to the name.

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:

By default, the base test name is selected to be:

```
${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:

```
${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:

ToDo: Explain how multiple tests can be added with different sets of arguments in one of two ways.

Determining Pass/Fail:

ToDo: Fill in!

Setting additional test properties:

ToDo: Fill in!

Debugging and Examining Test Generation:

ToDo: Describe setting ${PROJECT_NAME}_VERBOSE_CONFIGURE=ON and seeing what info it prints out.

ToDo: Describe how to examine the generated CTest files to see what test(s) actually got added (or not added) and what the pass/fail criteria is.

Disabling Tests Externally:

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.

10.20 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 <category1> <category2> ...]
  [HOST <host1> <host2> ...]
  [XHOST <host1> <host2> ...]
  [HOSTTYPE <hosttype1> <hosttype2> ...]
  [XHOSTTYPE <hosttype1> <hosttype2> ...]
  [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 takes the form:

```
TEST_<i>
   (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:

- Setting Additional Test Properties (TRIBITS_ADD_ADVANCED_TEST())
- Disabling Tests Externally (TRIBITS_ADD_ADVANCED_TEST())
- Debugging and Examining Test Generation (TRIBITS_ADD_ADVANCED_TEST())

Overall Arguments (TRIBITS_ADD_ADVANCED_TEST()):
Some overall arguments are:

`<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.

`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 <category1> <category2> ...`

Gives the test categories this test will be added. See TRIBITS_ADD_TEST() for more details.

`HOST <host1> <host2> ...`

The list of hosts to enable the test for (see TRIBITS_ADD_TEST()).

`XHOST <host1> <host2> ...`

The list of hosts *not* to enable the test for (see TRIBITS_ADD_TEST()).

`HOSTTYPE <hosttype1> <hosttype2> ...`

The list of host types to enable the test for (see TRIBITS_ADD_TEST()).

`XHOSTTYPE <hosttype1> <hosttype2> ...`

The list of host types *not* to enable the test for (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`.

Overall arguments that control overall pass/fail are described in ???.

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>`:

`EXEC <exeRootName> [NOEXEPREFIX] [NOEXESUFFIX] [ADD_DIR_TO_NAME] [DIRECTORY <dir>]`

If specified, then `<exeRootName>` gives the the name of an executable target that will be run as the command. The full executable path is determined in exactly the same way it is in the TRIBITS_ADD_TEST() function (see Determining the Exectuable or Command to Run). 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>` is basically equivalent to `CMND <cmndExec>` when `NOEXEPREFIX` and `NOEXESUFFIX` are specified. 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).

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_<i> 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>.

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.

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 Ordering (TRIBITS_ADD_ADVANCED_TEST()):
For the most part, the listed arguments can appear in any order except for the following restrictions:

- 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.

Implementation Details (TRIBITS_ADD_ADVANCED_TEST()):

ToDo: Describe the generation of the `*.cmake` file and what gets added with ADD_TEST().

Setting Additional Test Properties (TRIBITS_ADD_ADVANCED_TEST()):

ToDo: Fill in!

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()):

ToDo: Describe setting `${PROJECT_NAME}_VERBOSE_CONFIGURE=ON` and seeing what info it prints out.

ToDo: Describe how to examine the generated CTest files to see what test(s) actually got added (or not added) and what the pass/fail criteria is.

10.21 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.

11 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_bultin_command()`, you can always access the original built-in command as `_some_bultin_command()`.

11.1 ADD_SUBDIRECTORIES()

Macro that adds a list of subdirectories all at once (removed boiler-place code).

Usage:

```
ADD_SUBDIRECTORIES(<dir1> <dir2> ...)
```

11.2 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>)
```

11.3 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>)
```

11.4 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.

11.5 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.

11.6 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().

11.7 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> ...)
```

11.8 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().

11.9 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>"`.

11.10 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>`.

11.11 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.

11.12 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.

11.13 CONCAT_STRINGS()

Concatenate a set of string argumnets.

Usage:

```
CONCAT_STRINGS(<outputVar> "<str0>" "<str1>" ...)
```

On output, <outputVar> is set to "<str0><str1>...".

11.14 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).

11.15 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).

11.16 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!

11.17 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 "")
```

11.18 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 "")
```

11.19 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.

11.20 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.

11.21 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.

11.22 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 powerful. What it does is to allow you to create your own keyword-based macros and functions like CMake has.

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 scopt:

- `MyVar_ARG0="a;b"`

- `MyVar_ARG1=""`
- `MyVar_ARG2="c"`
- `MyVar_OPT0="FALSE"`
- `MyVar_OPT1="TRUE"`

Any initial arguments that are not recongnised as `<argNamesList>` 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`.

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.

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.

11.23 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.

11.24 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().

11.25 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> ...)
```

11.26 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.

11.27 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.

11.28 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!

11.29 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>)`.

11.30 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.

11.31 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.

11.32 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.

11.33 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!

11.34 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).

11.35 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>`.

11.36 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`.

11.37 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.

11.38 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.

11.39 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.

11.40 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.

12 References

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 ???). 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 ([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.

[SCALE] http://scale.ornl.gov/