TriBITS Developers Guide and Reference

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

Abstract

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

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.

The remainder of this documented is structured as follows. First, there is a discussion of the various TriBITS Developer and User Roles. Then a brief discussion of CMake Language Overivew and Gotchas is provided.

ToDo: Finish outline of the document.

The final sections are detailed documentation of TriBITS Global Project Settings, TriBITS Macros and Functions, and General Utility Macros and Functions.

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 CMake Language Overivew and Gotchas

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

However, we have observed that most mistakes that people make when using TriBITS, and most of the problems they have when using the 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. Also, documenation for any built-in CMake command is available locally by running:

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

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

The CMake language that is used to write CMake projects with TriBITS (and that core TriBITS 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.

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

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

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

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 TriBITS Project Structure

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

TriBITS Structural Units

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

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

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

- TriBITS Subpackage: A part of a parent package that also typically has source files built into libraries and tests but is documented and tested along with the other subpackages the parent package. The primary purpose for supporting subpackages to provide finer-grained of control software dependencies. In `TribitsExampleProject`, `PackageWithSubpackages` is an example of a package with subpackages `SubpackaeA`, `SubpackaeB`, and `SubpackageC`. The full subpackage name has the parent package name prefixed the the subpackage name (e.g. `PackageWithSubpackagesSubpackageA`). The parent package is always implicitly dependent on its subpackages.
- **TriBITS SE Package**: The combined set of TriBITS packages and subpackages. SE packages are the basis for setting dependencies in the TriBITS system. For example, the SE Packages provided by the example `PackageWithSubpackages` is (in order of increasing dependencies) `PackageWithSubpackagesSubpackaeA`, `PackageWithSubpackagesSubpackaeB`, `PackageWithSubpackagesSubpackageC`, and `PackageWithSubpackages`.
- TriBITS TPL: The specification for a particular external dependency that is required or can be used in one or more TriBITS SE Packages. A TPL (a Third Party Library) typically provides a list of libraries or a list include directories for header files but can also be manifisted in order ways as well. Examples of basic TPLs include `BLAS`, `LAPACK`, and `Boost`.
- TriBITS Repository: A collection of one or more TriBITS packages specified in a `PackagesList.cmake` file.
- TriBITS Project: A collection of TriBITS Repositories and Packages that defines a CMake `PROJECT` and can be configured, built, and tested.

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

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

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

- *Local Fixed-Name Variables* are used temporarily in the processing of a TriBITS unit. These include variables such as `PROJECT_NAME`, `REPOSITORY_NAME`, `PACKAGE_NAME`, and `PARENT_PACKAGE_NAME` These are distinguished by having a fixed/constant name. They are typically part of TriBITS reflection system, allowing subordinate units to determine the encapsulating unit in which they are participating. For, example, a TriBITS subpackage can determine its name, its parent package's name and directories, its parent repository name and directories, and the enclosing project's name and directories.
- *Namespaced Variables* are used to refer to properties of a named TriBITS unit. These include variables such as `${REPOSTORY_NAME}_SOURCE_DIR` (e.g. `TribitsExProj_SOURCE_DIR`) and `${PACKAGE_NAME}_BINARY_DIR` (e.g. `SimpleCXX_BINARY_DIR`). They are available after processing the unit, for use by downstream or subordinate units. They are part of the TriBITS dependency system, allowing downstream units to access properties of their known upstream dependencies.

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

TriBITS Project

- This defines a complete CMake project build by calling `PROJECT(${PROJECT_NAME} ...)`.
- Consists of one or more TriBITS Repositories (see TriBITS Repository).
- Defines `PROJECT_NAME` CMake variable (defined in `ProjectName.cmake`)

- Defines a set of native Repositories (see below) that define packages and TPLs. This set of native Repositories defines the official packages list and dependencies data-structure. The list of native Repositories can be empty!
- Allows for extra Repositories to be added on before and after the set of native Repositories (given in `ExtraRepositoriesList.cmake` or by CMake variables)
- Defines a default CDash server and default project name on the server (the project name on the CDash server must be the same as `${PROJECT_NAME}`).

TriBITS Project Files

The core files making up a TriBITS Project are:

```
${${PROJECT_SOURCE_DIR}}/
   CMakeLists.txt
   CTestConfig.cmake
   ProjectName.cmake     # Defines PACAKGE_NAME
   Version.cmake         # [Optional] Dev mode, Project version, VC branch
   cmake/
     NativeRepositoriesList.cmake    # [Optional] Used for some meta-projects
     ExtraRepositoriesList.cmake     # [Optional] Lists repos and VC URLs
     ProjectDependenicesSetup.cmake  # [Optional] Project deps overrides
     CallbackDefineProjectPackaging.cmake  # [Optional] CPack settings
     tribits/    # [Optional] Or provide ${PROJECT_NAME}_TRIBITS_DIR
     ctest/
       CTestCustom.ctest.in  # [Optional] Custom ctest settings
```

ToDo: Document each file in detail here!

TriBITS Project Core Variables

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

`PROJECT_NAME`

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

`PROJECT_SOURCE_DIR`

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

`PROJECT_BINARY_DIR`

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

`${PROJECT_NAME}_SOURCE_DIR`

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

`${PROJECT_NAME}_BINARY_DIR`

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

`${PACKAGE_NAME}_ENABLE_TESTS`

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

`${PACKAGE_NAME}_ENABLE_EXAMPLES`

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

TriBITS Repository

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

In short, a TriBITS Repository:

- Is a collection of related TriBITS Packages and TPLs
- Defines an association between one or more packages
- Defines a CMake variable specific to the collection referred to the in the variable `REPOSITORY_NAME`.
- Defines the base source and binary directories for the Repository `${REPOSITORY_NAME}_SORUCE_DIR` and `${REPOSITORY_NAME}_BINARY_DIR`.
- [Optional] Defines a list of add-on packages and TPLs (in `PackagesList.cmake` and `TPLsList.cmake`)
- [Optional] Defines a common set of initializations and other hooks for a collection of projects.

TriBITS Repository Core Files

The core files making up a TriBITS Repository are:

```
${${REPOSITORY_NAME}_SOURCE_DIR}/
  PackagesList.cmake
  TPLsList.cmake
  Copyright.txt  # [Optional] Only needed if creating version header file
  Version.cmake  # [Optional] Info inserted into ${REPO_NAME}_version.h
  cmake/
     RepositoryDependenciesSetup.cmake # [Optional]
     CallbackSetupExtraOptions.cmake # [Optional] Called after tribits options
     CallbackDefineRepositoryPackaging.cmake # [Optioinal] CPack packaging
```

ToDo: Document each file in detail here!

TriBITS Repository Core Varaibles

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

`REPOSITORY_NAME`

The name of the current TriBITS repository.

`REPOSITORY_DIR`

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

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

`${REPOSITORY_NAME}_SOURCE_DIR`

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

`${REPOSITORY_NAME}_BINARY_DIR`

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

Before processing the various repositories files, the following internal project-scope local CMake variables are defined by TriBITS:

`${PROJECT_NAME}_NATIVE_REPOSITORIES`

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

`${PROJECT_NAME}_EXTRA_REPOSITORIES`

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

`${PROJECT_NAME}_ALL_REPOSITORIES`

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

TriBITS Package

A TriBITS Package:

- Typically defines a set of libraries and/or headers and/or executables and/or tests with cmake build targets for building these and publishes the list of include directories and libraries that are created (along with CMake dependencies).
- Is declared in its parent repositories `PackagesList.cmake` file.
- Defines dependencies on upstream TPLs and/or other SE packages by just naming the dependencies in the file `cmake/Dependencies.cmake` using the macro TRIBITS_DEFINE_PACKAGE_DEPENDENCIES().
- Can optionally have subpackages listed in the argument `SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS` to TRIBITS_DEFINE_PACKAGE_DEPENDENCIES().

- Publishes a list of header file include paths and/or libraries for other downstream dependnet packages to use.

TriBITS Package Core Files

Teh core files that make up TriBITS Package (or Subpackage) are:

```
${${PACKAGE_NAME}_SOURCE_DIR}/
  CMakeLists.txt  # Only processed if the package is enabled
  cmake/
    Dependencies.cmake  # Always processed if the package is listed in the
                        # enclosing Repository
```

ToDo: Document each file in detail here!

TriBITS Package Core Variables

These are top-level (non-cache) CMake variables that are available once an SE Package (Package or Subpackage) is processed.

`PACKAGE_NAME`

The name of the current TriBITS SE package. This is set automatically by TriBITS before the packages's `CMakeLists.txt` file is processed.

`PACKAGE_SOURCE_DIR`

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

`PACKAGE_BINARY_DIR`

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

`${PACKAGE_NAME}_SOURCE_DIR`

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

`${PACKAGE_NAME}_BINARY_DIR`

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

`PARENT_PACKAGE_SOURCE_DIR`

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

`PARENT_PACKAGE_BINARY_DIR`

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

`${PACKAGE_NAME}_PARENT_REPOSITORY`

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

`${PACKAGE_NAME}_ENABLE_TESTS`

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

`${PACKAGE_NAME}_ENABLE_EXAMPLES`

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

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

TriBITS Subpackage

A TriBITS Subpackage:

- Typically defines a set of libraries and/or headers and/or executables and/or tests with cmake build targets for building these and publishes the list of include directories and libraries that are created (along with CMake dependencies).
- Is declared in its parent packages's `Dependencies.cmake` file in a call to TRIBITS_DEFINE_PACKAGE_DEPENDENCIES() using the argument `SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS`.
- Defines dependencies on upstream TPLs and/or other SE packages by just naming the dependencies in the file `cmake/Dependencies.cmake` using the macro TRIBITS_DEFINE_PACKAGE_DEPENDENCIES().
- Publishes a list of header file include paths and/or libraries for other downstream dependnet packages to use.
- Can **NOT** have its own subpakcages defined (only top-level packages can have subpackages).

A subpakages set of core files is identical to the TriBITS Package Core Files and the variables defined are almost identicial. The only difference is that a subpakcage may need to refer to its parent package where a top-level package has not parent package.

TriBITS TPL

A TriBITS TPL:

- Defines a set of pre-built libraries and/or header files and/or executables and/or some other resoruces used by one or more TriBITS Packages.
- Is listed as an explicit optional and required dependency in one or more package's `Dependencies.cmake` file using the macro TRIBITS_DEFINE_PACKAGE_DEPENDENCIES().
- Publishes the include directories and/or libraries and/or executables provided by the TPL.

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

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

```
INCLUDE(TribitsTplDeclareLibraries)

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

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

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

Processing of TriBITS Files: Ordering and Details

ToDo: Fill in!

Coexisting Projects, Repositories, and Packages

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

TriBITS Repository == TriBITS Project: It is allowed for a TriBITS Project and a TriBITS Repository to be the same directory and have the same names with no `NativeRepositoriesList.cmake` file. In this case, the repository name and the project name and the same name. This is quite common. This is the case, for example, with the the Trilinos and the TribitsExampleProject projects and repositories. In this case, the Project's and the Repository's `Version.cmake` and `Copyright.txt` files are one and the same.

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

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

This is used in the real TriBITS repository DataTransferKit.

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

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

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

Other than those modifications, a TriBITS project, repository, and package can all be rooted in the same source directory. However, to do so is a little messing and is not recommended. It was only done this way with the base TriBITS directory in order to maintain backward compatibility.

However, one possible use case for collapsing a project, repository, and package into a single base source directory would be to support the stand-alone build of a TriBITS package as its own entity that uses and installation of the TriBITS CMake modules. If a given upstream TriBITS package has no required upstream TriBITS package dependencies and minimal TPL dependencies (or only uses "standard" TriBITS TPLs already defined in the `tribits/tpls/` directory), then creating a stand-alone project build of a loan TriBITS package requires fairly little extra overhead.

Standard TriBITS TPLs

ToDo: Fill in!

5 Example TriBITS Projects

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

The first example covered in detail is TribitsExampleProject. This first example covers all the basics for setting up a simple multi-package TriBITS project. The second example outlined is *MockTrilinos* which mostly exists to test the TirBITS system itself. The last example mentioned is the the `TriBITS` package itself which allows the TriBITS system to be tested and installed from any TriBITS project that lists it, including the `TriBITSProj` project itself (see Coexisting Projects, Repositories, and Packages).

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

TribitsExampleProject

Here, the example project TribitsExampleProject that is provided with the TriBITS directory tree is outlined. From this simple example, one can quickly see how the basic structural elements a TriBITS example project are put together.

ToDo: Fill in!

MockTrilinos

ToDo: Fill in!

The TriBITS Package Itself

ToDo: Fill in!

6 Package Enable/Disable Logic

ToDo: Fill in!

The following TriBITS repository-related variables alter what packages in a given TriBITS repository get enabled implicitly or not:

`${REPOSITORY_NAME}_NO_IMPLICIT_PACKAGE_ENABLE`

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

`${REPOSITORY_NAME}_NO_IMPLICIT_PACKAGE_ENABLE_EXCEPT`

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

The above variables typically are defined in the outer TriBITS Project's `PackageName.cmake` file in order to adjust how its listed repositories are handled.

ToDo: Fill in!

7 Automated testing

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

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

Testing categories for Repositories, Packages, and Tests

ToDo: Define repo category Continuous, Nightly, and Experimental which also map to CDash tracks

ToDo: Define SE package test group PT, ST, and EX

ToDo: Define test category BASIC, CONTINUOUS, NIGHTLY, WEEKLY, and PERFORMANCE

ToDo: Discuss the propery usage of these test categories and why NIGHTLY testing should be the default.

ToDo: Fill in!

Pre-push Testing using checkin-test.py

Describe the checkin-test.py script

TriBITS Package-by-Package CTest/Dash Driver

ToDo: Fill in!

TriBITS CDash Customizations

ToDo: Fill in!

CDash regression email addresses

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

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

1) REGRESSION_EMAIL_LIST (defined in Dependneices.cmake): Package-specific email address overrides specified in the Dependencies.cmake file (the local variable REGRESSION_EMAIL_LIST). This package-specific email address will be overridden if ${REPOSITORY_NAME}_REPOSITORY_OVERRIDE_PACKAGE_EMAIL_LIST using the single Project or Repository master regression emails lists described above.

2) ${REPOSITORY_NAME}_REPOSITORY_EMAIL_URL_ADDRESSS_BASE: A base email address specified at the Repository level creating package-specific email addresses (e.g. <lower-case-package-name>-regression@some.repo.gov, where ${PROJECT_NAME}_REPOSITORY_EMAIL_URL_ADDRESSS_BASE=some.repo.gov). This variable is set in the Repositories cmake/DependenciesSetup.cmake file to provide a default for the repository but can be overridden (i.e. by the project).

3) ${REPOSITORY_NAME}_REPOSITORY_MASTER_EMAIL_ADDRESSS: A Single email address for all packages specified at the Repository level (e.g. my-repo-regression@some.repo.gov). This variable is set in the Repositories cmake/DependenciesSetup.cmake file to provide a default for the repository but can be overridden (i.e. by the project).

4) ${PROJECT_NAME}_PROJECT_EMAIL_URL_ADDRESSS_BASE: A base email address specified at the Project level creating package-specific email addresses (e.g. <lower-case-package-name>-regression@some.project.gov, where ${PROJECT_NAME}_PROJECT_EMAIL_URL_ADDRESSS_BASE=some.project.gov). If not already set, this variable will be set to ${REPOSITORY_NAME}_REPOSITORY_EMAIL_URL_ADDRESSS_BASE for the first repostory processed that has this set.

5) ${PROJECT_NAME}_PROJECT_MASTER_EMAIL_ADDRESSS: A Single email address for all packages specified at the Project level (e.g. my-project-regression@some.project.gov). If not already set, this variable will

be set to ${REPOSITORY_NAME}_REPOSITORY_MASTER_EMAIL_ADDRESSS for the first repostory processed that has this set.

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

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

As a general rule, repository-level settings override project-level settings and package-level settings override both. Also, a project can redefine a reposiotry's regression email list settings.

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

```
TRIBITS_READ_PACKAGES_PROCESS_DEPENDENCIES_WRITE_XML()
```

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

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

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

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

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

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

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

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

1) Change the TriBITS CMake files as described above that result in the desired CDashSubprojectDependeinces.xml file. You can debug this by running the checkin-test.py script and seeing what gets written in the ${PROJECT_NAME}Dependencies.xml file in the CHECKIN directory.

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

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

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

8 Multi-Repository Support

ToDo: Discuss 'egdist', ExtraRepositoriesList.cmake, and the rep clone script.

Multi-Repository Almost Continuous Integration

ToDo: Fill in!

9 Development Workflows

Basic Development Workflow

ToDo: Fill in!

Multi-Repository Development Workflow

ToDo: Fill in!

10 Additional Topics

In this section, a number of miscellaneous topics and TriBITS features are discussed. These features and topics are not considered the primary fetures of TriBITS but can be very useful in many situations.

Project-Specific Build Quick Reference

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

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

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

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

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

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

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

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

To see a simple example of this, see:

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

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

Creating Source Distributions

ToDo: Fill in!

Regulated Backward Compatibility and Deprecated Code

ToDo: Fill in!

TriBITS Dashboard Driver

ToDo: Fill in!

11 TriBITS Global Project Settings

TriBITS defines a number of global project-level settings that can be set by the user and can have their default determined by each individual TriBITS project. If a given TriBITS project does not define its own default, a reasonble default is set by the TriBITS system automatically. These options are defined and are set, for the most part, in the internal TriBITS function `TRIBITS_DEFINE_GLOBAL_OPTIONS_AND_DEFINE_EXTRA_REPOS()` in the TriBITS CMake code file `TribitsGlobalMacros.cmake` which gets called inside of the `TRIBITS_PROJECT()` macro. That function and that file are the definitive source the options that a TriBITS project takes and what the default values are but we strive to document them here as well. Many of these global options (i.e. cache variables) such as `${PROJECT_NAME}_<SOME_OPTION>` allow the project to define a default by setting a local varible `${PROJECT_NAME}_<SOME_OPTION>_DEFAULT` as:

```
SET(${PROJECT_NAME}_<SOME_OPTION>_DEFAULT <someDefault>)
```

either in its top-level `CMakeLists.txt` file or in its `ProjectName.cmake` file. If `${PROJECT_NAME}_<SOME_OPTION>_DEFAULT` is not set by the project, then TriBITS provides a reasonable default value. The TriBITS code for this looks like:

```
IF ("${${PROJECT_NAME}_<SOME_OPTION>_DEFAULT}" STREQUAL "")
  SET(${PROJECT_NAME}_<SOME_OPTION>_DEFAULT <someDefault>)
ENDIF()

ADVANCED_SET( ${PROJECT_NAME}_<SOME_OPTION>
  ${PROJECT_NAME}_<SOME_OPTION>_DEFAULT}
  CACHE BOOL "[documentation]."
  )
```

where `<SOME_OPTION>` is the option name like `TEST_CATEGORIES` and `<someDefault>` is the default set by TriBITS if the project does not define a default. In this way, if the project sets the variable `${PROJECT_NAME}_<SOME_OPTION>_DEFAULT` before this code exeutates, then `${${PROJECT_NAME}_<SOME_OPTION>_DEFAULT}` will be used as the default for the cache varible `${PROJECT_NAME}_<SOME_OPTION>` which, of course, can be overridden by the user when calling `cmake` in a number of ways.

Most of these global options that can be overridden externally by setting the cache variable `${PROJECT_NAME}_<SOME_OPTION>` should be documented in the Project-Specific Build Quick Reference document. A generic version of this document is found in TribitsBuildQuickRef.[rsts,html,pdf]. Some of the more unusual options that might only be of interest to developers mentioned below may not be documented in `<Project>BuildQuickRef.[rst,html,pdf]`.

The global project-level TriBITS options for which defaults can be provided by a given TriBITS project are:

- ${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES

- ${PROJECT_NAME}_ENABLE_Fortran
- ${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS
- ${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES
- ${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES
- ${PROJECT_NAME}_GENERATE_EXPORT_FILE_DEPENDENCIES
- ${PROJECT_NAME}_ELEVATE_ST_TO_PT
- ${PROJECT_NAME}_ENABLE_CPACK_PACKAGING
- ${PROJECT_NAME}_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION
- ${PROJECT_NAME}_CPACK_SOURCE_GENERATOR
- ${PROJECT_NAME}_TEST_CATEGORIES
- MPI_EXEC_MAX_NUMPROCS

These options are described below.

${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES

If `${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES` is `ON` (the TriBITS default value), then any explicitly enabled packages that have disabled upstream required packages or TPLs will be disabled. If `OFF`, then an configure error will occur (for more details see TribitsBuildQuickRef.*). A project define a different default value by setting:

```
SET(${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES_DEFAULT FALSE)
```

${PROJECT_NAME}_ENABLE_Fortran

If `${PROJECT_NAME}_ENABLE_Fortran` is `ON`, then Fortran support for the project will be enabled and the Fortran compiler(s) must be found. By default, TriBITS sets this to `ON` for non-Windows systems (i.e. `WIN32` is not set by CMake) but is `OFF` for a Windows system. A project always requires Fortran, for example, it can set the default:

```
SET(${PROJECT_NAME}_ENABLE_Fortran_DEFAULT TRUE)
```

If a project does not have any native Fortran code a good default would be:

```
SET(${PROJECT_NAME}_ENABLE_Fortran_DEFAULT OFF)
```

NOTE: It is usually not a good idea to always force off Fortran, or any compiler, because extra repositories and packages might be added by someone that might require the compiler and we don't want to unnecessarily limit the generality of a given TriBITS build. Setting the default for all platforms should be sufficient.

${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS

If `${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS` is set to `ON`, then any defined libraries or header files that are listed in calls to TRIBITS_ADD_LIBRARY() will be installed (unless options are passed into TRIBITS_ADD_LIBRARY() that disable installs). If set to `OFF`, then headers and librareis will be installed by default and only `INSTALLABLE` executables added with TRIBITS_ADD_EXECUTABLE() will be installed. However, as described in TribitsBuildQuickRef.*, shared libraries will still be always be installed if enabled since they are needed by the installed executables. The TriBITS default is to set this to `ON`.

For a TriBITS project that primarily is delivering libraries (e.g. Trilinos), then it makes sense to leave the TriBITS default or explicitly set:

```
SET(${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS_DEFAULT ON)
```

For a TriBITS project that is primarily delivering executablers (e.g. VERA), then it makes sense to set the default to:

```
SET(${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS_DEFAULT OFF)
```

${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES

If `${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES` is `ON`, then `Makefile.export.<PACKAGE_NAME>` will get created at configure time in the build tree and installed into the install tree. See TribitsBuildQuickRef.* for details. The TriBITS default is `ON` but a project can decide to turn this off by default by setting:

```
SET(${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES_DEFAULT OFF)
```

A project might want to disable the generation of export makefiles by default if its main purpose is to provide executables. There is no reason to provide an export makefile if libraies and headers are not actaully installed (see ${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS)

${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES

If `${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES` is set to `ON`, then `<PACKAGE_NAME>Config.cmake` files are created at configure time in the build tree and installed into the install tree. These files are used by external CMkae projects to pull in the list of compilers, compiler options, include directories and libraries. The TriBITS default is `ON`. A project can change the default by setting, for example:

```
SET(${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES_DEFAULT OFF)
```

A project would want to turn off the creation and installation of `<PACKAGE_NAME>Config.cmake` files if it was only installing and providing executables. See TribitsBuildQuickRef.* for details.

${PROJECT_NAME}_GENERATE_EXPORT_FILE_DEPENDENCIES

If `${PROJECT_NAME}_GENERATE_EXPORT_FILE_DEPENDENCIES` is `ON`, then the data-structures needed to generate `Makefile.export.<PACKAGE_NAME>` and `<PACKAGE_NAEM>Config.cmake` are created. These data structures are also needed in order to generate export makefiles on demand using the function TRIBITS_WRITE_FLEXIBLE_PACKAGE_CLIENT_EXPORT_FILES(). The default in TriBITS is to turn this `ON` automatically by default if `${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES` or `${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES` are `ON`. Else, by default, TriBITS sets this to `OFF`. The only reason for the project to override the default is to set it to `ON` as with:

```
SET(${PROJECT_NAME}_GENERATE_EXPORT_FILE_DEPENDENCIES_DEFAULT
ON)
```

is so that the necessary data-structures are generated in order to use the function TRIBITS_WRITE_FLEXIBLE_PACKAGE_CLIENT_EXPORT_FILES().

${PROJECT_NAME}_ELEVATE_ST_TO_PT

If `${PROJECT_NAME}_ELEVATE_ST_TO_PT` is set to `ON`, then all `ST` SE packages will be elevated to `PT` packages. The TriBITS default is obviously `OFF`. The default can be changed by setting:

```
SET(${PROJECT_NAME}_ELEVATE_ST_TO_PT_DEFAULT ON)
```

There are projects, especially meta-projects, where the distiction between `PT` and `ST` code is not helpful or the assignment of `PT` and `ST` packages in a repository is not appropriate. An example project like this CASL VERA. Changing the default to `ON` allows any packages to be considered in pre-push testing.

${PROJECT_NAME}_ENABLE_CPACK_PACKAGING

If `${PROJECT_NAME}_ENABLE_CPACK_PACKAGING` is `ON`, then CPack support is enabled and some TriBITS code is avoided that is needed to set up data-structures that are used by the built-in CMake target `package_source`. The TriBITS default is `OFF` with the idea that the average developer or user will not be wanting to create source distributions with CPack. However, this default can be changed by setting:

```
SET(${PROJECT_NAME}_ENABLE_CPACK_PACKAGING ON)
```

${PROJECT_NAME}_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION

If `${PROJECT_NAME}_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION` is `TRUE`, then the directories for subpackages that are not enabled are left out of the source tarball. This reduces the size of the tarball as much as possible but does require that the TriBITS packages and subpackages be properly set up to allow disabled subpackages from being excluded. The TriBITS default is `TRUE` but this can be changed by setting:

```
SET(${PROJECT_NAME}_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION_DEFAULT FALSE
```

${PROJECT_NAME}_CPACK_SOURCE_GENERATOR

The variable `${PROJECT_NAME}_CPACK_SOURCE_GENERATOR` determines the CPack source generation types that are created when the `package_source` target is run. The TriBITS default is set to `TGZ`. However, this default can be overridded by setting, for example:

```
SET(${PROJECT_NAME}_CPACK_SOURCE_GENERATOR_DEFAULT "TGZ;TBZ2")
```

This variable should generally be set in the file:

```
<projectDir>/cmake/CallbackDefineProjectPackaging.cmake
```

instead of in the base-level `CMakeLists.txt` file so that it goes along with rest of the project-specific CPack packaging options.

${PROJECT_NAME}_TEST_CATEGORIES

The cache varaible `${PROJECT_NAME}_TEST_CATEGORIES` detemines what tests defined using TRIBITS_ADD_TEST() and TRIBITS_ADD_ADVANCED_TEST() will be added for `ctest` to run (see *Automated testing*) for discussion of test categories). The TriBITS default is `NIGHTLY` for a standard local build. The `checkin-test.py` script sets this to `BASIC`. A TriBITS project can override the default for a basic configure using, for example:

```
SET(${PROJECT_NAME}_TEST_CATEGORIES BASIC)
```

The justification for having the default test category be `NIGHTLY` instead of `BASIC` is that when someone is enabling a package to develop on it or install it, we want them by default to be seeing the full version of the test suite (shy of the `WEEKLY` tests which can be very expensive) for the packages they are explictly enabling. Typically they will not be enabling forward (downstream) dependent packages so the cost of running the test suite should not be too prohibitive. This all depends on how good of a job the development teams do in making their test suites run fast and keeping the cost of running the tests down. See the section Automated testing for a more detailed discussion.

MPI_EXEC_MAX_NUMPROCS

The varaible `MPI_EXEC_MAX_NUMPROCS` gives the maximum number of processes for an MPI test that will be allowed as defined by TRIBITS_ADD_TEST() and TRIBITS_ADD_ADVANCED_TEST(). The TriBITS default is set to be `4` (for no good reason really but it needs to stay that way for backward compatibility). This default can be changed by setting:

```
SET(MPI_EXEC_MAX_NUMPROCS_DEFAULT <newDefaultMax>)
```

While this default can be changed for the project as a whole on all platforms, it is likely better to change this default on a machine-by-machine basis to correspond to the loat that can be accomidated by a given machine (or class of machines). For example if a given machine has 64 cores, a reasonble number for `MPI_EXEC_MAX_NUMPROCS_DEFAULT` is 64.

12 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. All of these functions and macros should be aviable when processing the project's and package's variables files if used properly. Therefore, no explicit `INCLUDE()` statements should be needed other than the initial include of the `TribitsProject.cmake` file in the top-level CMakeLists.txt file so the command TRIBITS_PROJECT() can be executed.

TRIBITS_PROJECT()

Defines and processes a TriBITS project.

Usage:

```
TRIBITS_PROJECT()
```

Requires that the project name variable `PROJECT_NAME` be defined before calling this macro. Also, all default values for project settings should be set before calling this (see TriBITS Global Project Settings). Also, the variable `${PROJECT_NAME}_TRIBITS_DIR` must be set as well.

This macro then adds all of the necssary paths to `CMAKE_MODULE_PATH` and then performs all processing of the TriBITS project files (see ???).

ToDo: Give documentation!

TRIBITS_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** (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".

NOTE: This macro just sets the varaible:

```
${REPOSITORY_NAME}_PACKAGES_AND_DIRS_AND_CLASSIFICATIONS
```

in the current scope. The advantages of using this macro instead of directly setting this varible include:

- Asserts that the varible `REPOSITORY_NAME` is defined and set
- Avoids having to hard-code the assumed repository name `${REPOSITORY_NAME}`. This provides more flexibility for how other TriBITS project name a given TriBITS repo (i.e. the name of repo subdirs).
- Avoid mispelling the name of the varible `${REPOSITORY_NAME}_PACKAGES_AND_DIRS_AND_CLASSIFICATIONS`. If you misspell the name of the macro, it is an immediate error in CMake.

TRIBITS_DEFINE_REPOSITORY_TPLS()

Define the list of TPLs, find modules, and classifications for a given TriBITS repository. This macro is typically called from inside of a TPLsList.cmake fil for a given TriBITS repo.

Usage:

```
TRIBITS_DEFINE_REPOSITORY_TPLS(
  <tpl0_name>   <tpl0_findmod>  <tpl0_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.

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_PACKAGES()).
  - **OPTREQ:** Determines if the outer parent package has an OPTIONAL or REQUIRED dependence on this subpackage.

Other variables that this macro handles:

- **REGRESSION_EMAIL_LIST:** The email list that is used to send CDash error messages. If this is missing, then the email list that CDash errors go to is determined by other means (see ???).

NOTE: All this macro really does is to just define the variables:

- LIB_REQUIRED_DEP_PACKAGES
- LIB_OPTIONAL_DEP_PACKAGES
- TEST_REQUIRED_DEP_PACKAGES
- TEST_OPTIONAL_DEP_PACKAGES
- LIB_REQUIRED_DEP_TPLS
- LIB_OPTIONAL_DEP_TPLS
- TEST_REQUIRED_DEP_TPLS
- TEST_OPTIONAL_DEP_TPLS
- REGRESSION_EMAIL_LIST
- SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS

which are then read by the TriBITS cmake code to build the package dependency graph. The advantage of using this macro instead of just directly setting the varibles is that you only need to list the dependencies you have. Otherwise, you need to set all of these varibles, even those that are empty. This is a error checking property of the TriBITS system to avoid misspelling the names of these variables.

TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES()

Macro used in Dependencies.cmake files to allow some upstream dependent packages to be missing.

Usage:

```
TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES(<pack_1> <pack_2> ...)
```

If the missing upstream SE package <pack_i> is optional, then the effect will be to simply ignore the missing package and remove it from the dependency list. However, if the missing upstream SE package <pack_i> is required, then in addition to ignoring the missing package, the current SE (sub)package will also ee hard disabled, i.e. ${PROJECT_NAME}_ENABLE_{CURRENT_PACKAGE}=OFF.

This function is typically used in packages in external TriBITS repos that are depend on other packages in other exteral TriBITS repos that might be missing.

NOTE: Using this function effectively turns off error checking for misspelled package names so it is important to only use it when it absolutely is needed.

TRIBITS_TPL_DECLARE_LIBRARIES()

Function that sets up cache variables for users to specify where to find a TPL's headers and libraries. This function is typically called inside of a file `FindTPL<tpl_name>.cmake` file.

Usage:

```
TRIBITS_TPL_DECLARE_LIBRARIES(
  <tpl_name>
  [REQUIRED_HEADERS <header1> <header2> ...]
  [MUST_FIND_ALL_HEADERS]
  [REQUIRED_LIBS_NAMES <libname1> <libname2> ...]
  [MUST_FIND_ALL_LIBS]
  [NO_PRINT_ENABLE_SUCCESS_FAIL]
  )
```

This function can set up a with header files and/or libraries.

The input arguments to this function are:

- `<tpl_name>`: Name of the TPL that is listed in a TPLsList.cmake file. Below, this is referted to as the local CMake variable `TPL_NAME`.
- `REQUIRED_HEADERS`: List of header files that are searched for the TPL using `FIND_PATH()`.
- `MUST_FIND_ALL_HEADERS`: If set, then all of the header files listed in REQUIRED_HEADERS must be found in order for TPL_${TPL_NAME}_INCLUDE_DIRS to be defined.
- `REQUIRED_LIBS_NAMES`: List of libraries that are searched for when looked for the TPLs libraries with FIND_LIBRARY(...).
- `MUST_FIND_ALL_LIBS`: If set, then all of the library files listed in REQUIRED_LIBS_NAMES must be found or the TPL is considered not found!
- **`NO_PRINT_ENABLE_SUCCESS_FAIL`: If set, then the final success/fail** will not be printed

The following cache variables, if set, will be used by that this function:

- `${TPL_NAME}_INCLUDE_DIRS:PATH`: List of paths to search first for header files defined in `REQUIRED_HEADERS`.
- `${TPL_NAME}_INCLUDE_NAMES:STIRNG`: List of include names to be looked for instead of what is specified in REQUIRED_HEADERS.
- `${TPL_NAME}_LIBRARY_DIRS:PATH`: The list of directories to search first for libraies defined in REQUIRED_LIBS_NAMES.
- `${TPL_NAME}_LIBRARY_NAMES:STIRNG`: List of library names to be looked for instead of what is specified in REQUIRED_LIBS_NAMES.

This function sets global varibles to return state so it can be called from anywhere in the call stack. The following cache variables defined that are intended for the user to set and/or use:

- `TPL_${TPL_NAME}_INCLUDE_DIRS`: A list of common-separated full directory paths that contain the TPLs headers. If this varible is set before calling this function, then no headers are searched for and this variable will be assumed to have the correct list of header paths.
- `TPL_${TPL_NAME}_LIBRARIES`: A list of commons-seprated full library names (output from FIND_LIBRARY(...)) for all of the libraries found for the TPL. IF this varible is set before calling this function, no libraries are searched for and this varaible will be assumed to have the correct list of libraries to link to.

TRIBITS_PACKAGE()

Macro called at the very beginning of a package's top-level CMakeLists.txt file.

Usage:

```
TRIBITS_PACKAGE(
  <packageName>
  [ENABLE_SHADOWING_WARNINGS]
  [DISABLE_STRONG_WARNINGS]
  [CLEANED]
  [DISABLE_CIRCULAR_REF_DETECTION_FAILURE]
  )
```

See TRIBITS_PACKAGE_DECL() for the documentation for the arguments and TRIBITS_PACKAGE_DECL() and TRIBITS_PACKAGE() for a description the side-effects (and varibles set) after calling this macro.

TRIBITS_PACKAGE_DECL()

Macro called at the very beginning of a package's top-level CMakeLists.txt file when a packages has subpackages.

If the package does not have subpackages, just call TRIBITS_PACKAGE() which calls this macro.

Usage:

```
TRIBITS_PACKAGE_DECL(
  <packageName>
  [ENABLE_SHADOWING_WARNINGS]
  [DISABLE_STRONG_WARNINGS]
  [CLEANED]
  [DISABLE_CIRCULAR_REF_DETECTION_FAILURE]
  )
```

The arguments are:

`<packageName>`

Gives the name of the Package, mostly just for checking and documentation purposes. This much match the name of the package provided in the PackagesLists.cmake or it is an error.

`ENABLE_SHADOWING_WARNINGS`

If specified, then shadowing warnings will be turned on for supported platforms/compilers. The default is for shadowing warnings to be turned off. Note that this can be overridden globally by setting the cache variable ${PROJECT_NAME}_ENABLE_SHADOWING_WARNINGS.

`DISABLE_STRONG_WARNINGS`

If specified, then all strong warnings will be turned off, if they are not already turned off by global cache variables. Strong warnings are turned on by default in development mode.

`CLEANED`

If specified, then warnings will be promoted to errors for all defined warnings.

`DISABLE_CIRCULAR_REF_DETECTION_FAILURE`

If specified, then the standard grep looking for RCPNode circular references that causes tests to fail will be disabled. Note that if these warnings are being produced then it means that the test is leaking memory and user like may also be leaking memory.

There are several side-effects of calling this macro:

- The the varibles listed the packages set of library targets `${PACKAGE_NAME}_LIB_TARGETS` and all targets `${PACKAGE_NAME}_ALL_TARGETS` and are initialized to emtpy.
- The local varibles `PACKAGE_SOURCE_DIR` and `PACKAGE_BINARY_DIR` are set for this package's use in its CMakeLists.txt files.
- Package-specific compiler options are set up in package-scoped (i.e., the package's subdir and its subdirs) in `CMAKE_<LANG>_FLAG`.
- This packages's cmake subdir `${PACKAGE_SOURCE_DIR}/cmake` is added to `CMAKE_MODULE_PATH` locally so that the package's try-compile modules can be read in with just a raw `INCLUDE()` leaving off the full path and the `*.cmake` extension.

TRIBITS_PACKAGE_DEF()

Macro called after subpackages are processed in order to handle the libraries, tests, and examples of the final package.

Usage:

```
TRIBITS_PACKAGE_DEF()
```

If the package does not have subpackages, just call TRIBITS_PACKAGE() which calls this macro.

This macro has several side effects:

- The varible `PACKAGE_NAME` is set in the local scope for usage by the package's CMakeLists.txt files.
- The intra-package dependency varibles (i.e. list of include directoires, list of libraries, etc.) are initialized to emtpy.

TRIBITS_PROCESS_SUBPACKAGES()

Macro that processes subpackages for packages that have them. This is called in the parent packages top-level CMakeLists.txt file.

Usage:

```
TRIBITS_PROCESS_SUBPACKAGES()
```

Must be called after TRIBITS_PACKAGE_DECL() but before TRIBITS_PACKAGE_DEF().

TRIBITS_ADD_TEST_DIRECTORIES()

Macro called to add a set of test directories for an SE package.

Usage:

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

This macro only needs to be called from the top most CMakeList.txt file for which all subdirectories are all "tests".

This macro can be called several times within a package and it will have the right effect.

Currently, really all it does macro does is to call `ADD_SUBDIRECTORY(<diri>)` if `${PACKAGE_NAME}_ENABLE_TESTS` or `${PARENT_PACKAGE_NAME}_ENABLE_TESTS` are true. However, this macro may be extended in the futgure in order to modify behavior related to adding tests and examples in a uniform way..

TRIBITS_ADD_EXAMPLE_DIRECTORIES()

Macro called to conditionally add a set of example directories for an SE package.

Usage:

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

This macro only needs to be called from the top most CMakeList.txt file for which all subdirectories are all "examples".

This macro can be called several times within a package and it will have the right effect.

Currently, really all it does macro does is to call `ADD_SUBDIRECTORY(<diri>)` if `${PACKAGE_NAME}_ENABLE_EXAMPLES` or `${PARENT_PACKAGE_NAME}_ENABLE_EXAMPLES` are true. However, this macro may be extended in the futgure in order to modify behavior related to adding tests and examples in a uniform way..

TRIBITS_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.

TRIBITS_WRITE_FLEXIBLE_PACKAGE_CLIENT_EXPORT_FILES()

Utility function for writing ${PACKAGE_NAME}Config.cmake and/or the Makefile.export.${PACKAGE_NAME} for package PACKAGE_NAME with some greater flexibility than TRIBITS_WRITE_PACKAGE_CLIENT_EXPORT_FILES()

Usage:

```
TRIBITS_WRITE_FLEXIBLE_PACKAGE_CLIENT_EXPORT_FILES(
  PACKAGE_NAME <pakageName>
  [EXPORT_FILE_VAR_PREFIX <exportFileVarPrefix>]
  [WRITE_CMAKE_CONFIG_FILE <cmakeConfigFileFullPath>]
  [WRITE_EXPORT_MAKLEFILE <exportMakefileFileFullPath>]
  [WRITE_INSTALL_CMAKE_CONFIG_FILE]
  [WRITE_INSTALL_EXPORT_MAKLEFILE]
  )
```

The arguments are:

PACKAGE_NAME <pakageName>

Gives the name of the TriBITS package for which the export files should be created for.

EXPORT_FILE_VAR_PREFIX <exportFileVarPrefix>

If specified, then all of the varibles in the generated export files will be prefixed with "<exportFileVarPrefix>_" instead of "${PACKAGE_NAME}_". This is to provide flexibility.

WRITE_CMAKE_CONFIG_FILE <cmakeConfigFileFullPath>

If specified, then the package <packageName>'s cmake configure export file for extenral CMake client projects will be created in the file <cmakeConfigFileFullPath>. NOTE: the argument should be the full path!

WRITE_EXPORT_MAKLEFILE <exportMakefileFileFullPath>

If specified, then the package <packageName>'s cmake configure export file for external Makefile client projects will be created in the file <exportMakefileFileFullPath>. NOTE: the argument should be the full path!

WRITE_INSTALL_CMAKE_CONFIG_FILE

If specified, then the package <packageName>'s install cmake configure export to be installed will be written. The name and location of this file is hard-coded.

`WRITE_INSTALL_EXPORT_MAKLEFILE`

> If specified, then the package <packageName>'s install export makefile to be installed will be written. The name and location of this file is hard-coded.

NOTE: The arguments to this function may look strange but the motivation is to support versy speicalized use cases such as when a TriBITS package needs to generate an export makefile for a given package but name the export makefile differently and use different variable name prefixes. The particular driver use case is when wrapping an external autotools project that depends on Trilinos and needs to read in the Makefile.export.Trilinos file but this file needs to be generated for a subset of enabled packages on the fly during a one-pass configure.

NOTE: This function does *not* contain the the INSTALL() commands because CMake will not allow those to even be present in scripting mode that is used for unit testing this function.

TRIBITS_ADD_OPTION_AND_DEFINE()

Add an option and a define variable in one shot.

Usage:

```
TRIBITS_ADD_OPTION_AND_DEFINE( <userOptionName>  <macroDefineName>
  "<docStr>"  <defaultValue> )
```

This macro sets the user cache `BOOL` variable `<userOptionName>` and if it is true, then sets the global (internal cache) macro define variable `<macroDefineName>` to `ON`, and otherwise sets it to `OFF`. This is designed to make it easy to add a user-enabled option to a configured header file and have the define set in one shot. This would require that the package's configure file (see TRIBITS_CONFIGURE_FILE()) have the line:

```
#cmakedefine <macroDefineName>
```

TRIBITS_CONFIGURE_FILE()

Macro that configures the package's main configured header file (typically called `${PACKAGE_NAME}_config.h` but any name can be used).

Usage:

```
TRIBITS_CONFIGURE_FILE(<packageConfigFile>)
```

This function requires the file:

```
${PACKAGE_SOURCE_DIR}/cmake/<packageConfigFile>.in
```

exists and it creates the file:

```
${CMAKE_CURRENT_BINARY_DIR}/<packageConfigFile>
```

by calling the built-in `CONFIGURE_FILE()` command:

```
CONFIGURE_FILE(
  ${PACKAGE_SOURCE_DIR}/cmake/<packageConfigFile>.in
  ${CMAKE_CURRENT_BINARY_DIR}/<packageConfigFile>
  )
```

which does basic sustitution of CMake variables (see documentation for built-in `CONFIGURE_FILE()` command for rules on how it performs substitutions).

In addition to just calling `CONFIGURE_FILE()`, this function also aids in creating configured header files adding macros for deprecating code.

Deprecated Code Macros

If `${PARENT_PACKAGE_NAME}_SHOW_DEPRECATED_WARNINGS` is `TRUE`, then the local CMake varible `${PARENT_PACKAGE_NAME_UC}_DEPRECATED_DECLARATIONS` adds a define `<PARENT_PACKAGE_NAME_UC>_DEPRECATED` (where `<PARENT_PACKAGE_NAME_UC>` is the package name in all upper-case letters) add the compiler-specific deprecated warning for an entity. To use this, just add the line:

```
@<PARENT_PACKAGE_NAME_UC>_DEPRECATED_DECLARATIONS@
```

to the <packageConfigFile>.in file and it will be expended.

Then C/C++ code can use this macro to deprecate functions, variables, classes, etc., for example, using:

```
<PARENT_PACKAGE_NAME_UC>_DEPRECATED class SomeDepreatedClass { ... }.
```

If the particular compiler does not support deprecated warnings, then this macro is defined to be empty. See Regulated Backward Compatibility and Deprecated Code for more details.

TRIBITS_COPY_FILES_TO_BINARY_DIR()

Function that copies a list of files from a soruce directory to a destination directory at configure time, typically so that it can be used in one or more tests. This sets up all of the custom CMake commands and targets to ensure that the files in the destiation directory are always up to date just by building the `ALL` target.

Usage:

```
TRIBITS_COPY_FILES_TO_BINARY_DIR(
  <targetName>
  [SOURCE_FILES <file1> <file2> ...]
  [SOURCE_DIR <sourceDir>]
  [DEST_FILES <dfile1> <dfile2> ...]
  [DEST_DIR <destDir>]
  [TARGETDEPS <targDep1> <targDep2> ...]
  [EXEDEPS <exeDep1> <exeDep2> ...]
  [NOEXEPREFIX]
  [CATEGORIES <category1>  <category2> ...]
  )
```

This function has a few valid calling modes:

1) Source files and destination files have the same name:

```
TRIBITS_COPY_FILES_TO_BINARY_DIR(
  <targetName>
  SOURCE_FILES <file1> <file2> ...
  [SOURCE_DIR <sourceDir>]
  [DEST_DIR <destDir>]
  [TARGETDEPS <targDep1> <targDep2> ...]
  [EXEDEPS <exeDep1> <exeDep2> ...]
  [NOEXEPREFIX]
  [CATEGORIES <category1>  <category2> ...]
  )
```

In this case, the names of the source files and the destination files are the same but just live in different directories.

2) Source files have a prefix different from the destination files:

```
TRIBITS_COPY_FILES_TO_BINARY_DIR(
  <targetName>
  DEST_FILES <file1> <file2> ...
  SOURCE_PREFIX <srcPrefix>
  [SOURCE_DIR <sourceDir>]
  [DEST_DIR <destDir>]
  [EXEDEPS <exeDep1> <exeDep2> ...]
  [NOEXEPREFIX]
  [CATEGORIES <category1>  <category2> ...]
  )
```

In this case, the source files have the same basic name as the destination files except they have the prefix 'srcPrefix' appended to the name.

3) Source files and destination files have completely different names:

```
TRIBITS_COPY_FILES_TO_BINARY_DIR(
  <targetName>
  SOURCE_FILES <sfile1> <sfile2> ...
  [SOURCE_DIR <sourceDir>]
  DEST_FILES <dfile1> <dfile2> ...
  [DEST_DIR <destDir>]
  [EXEDEPS <exeDep1> <exeDep2> ...]
  [NOEXEPREFIX]
  [CATEGORIES <category1>  <category2> ...]
  )
```

In this case, the source files and destination files have completely different prefixes.

The individual arguments are:

`SOURCE_FILES <file1> <file2> ...`

> Listing of the source files relative to the source directory given by the argument `SOURCE_DIR <sourceDir>`. If omited, this list will be the same as `DEST_FILES` with the argument `SOURCE_PREFIX <srcPrefix>` appended.

`SOURCE_DIR <sourceDir>`

> Optional argument that gives (absolute) the base directory for all of the source files. If omited, this takes the default value of `${CMAKE_CURRENT_SOURCE_DIR}`.

`DEST_FILES <file1> <file2> ...`

> Listing of the destination files relative to the destination directory given by the argument `DEST_DIR <destDir>` If omited, this list will be the same as given by the `SOURCE_FILES` list.

`DEST_DIR <destDir>`

> Optional argument that gives the (absolute) base directory for all of the destination files. If omited, this takes the default value of `${CMAKE_CURRENT_BINARY_DIR}`

`TARGETDEPS <targDep1> <targDep2> ...`

> Listing of general CMake targets that these files will be added as dependencies to.

`EXEDEPS <exeDep1> <exeDep2> ...`

Listing of executable targets that these files will be added as dependencies to. By default the prefix `${PACKAGE_NAME}_` will is appended to the names of the targets. This ensures that if the executable target is built that these files will also be copied as well.

`NOEXEPREFIX`

Option that determines if the prefix `${PACKAGE_NAME}_` will be appended to the arguments in the `EXEDEPS` list.

TRIBITS_INCLUDE_DIRECTORIES()

This function is to override the standard behavior of include_directories for a TriBITS package.

Usage:

```
TRIBITS_INCLUDE_DIRECTORIES(
  [REQUIRED_DURING_INSTALLATION_TESTING] <dir0> <dir1> ...
  )
```

If specified, `REQUIRED_DURING_INSTALLATION_TESTING` can appear anywhere in the argument list.

This function allows overriding the default behavior for installation testing, to ensure that include directories will not be inadvertently added to the build lines for tests during installation testing. Normally we want the include directories to be handled as cmake usually does. However during TriBITS installation testing we do not want most of the include directories to be used as the majority of the files should come from the installation we are building against. There is an exception to this and that is when there are test only headers that are needed. For that case we allow people to set `REQUIRED_DURING_INSTALLATION_TESTING` to tell us that this include directory does need to be set for instaltion testing.

TRIBITS_ADD_LIBRARY()

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

Usage:

```
TRIBITS_ADD_LIBRARY(
  <libName>
  [HEADERS <h0> <h1> ...]
  [NOINSTALLHEADERS <nih0> <hih1> ...]
  [SOURCES <src0> <src1> ...]
  [DEPLIBS <deplib0> <deplib1> ...]
  [IMPORTEDLIBS <ideplib0> <ideplib1> ...]
  [TESTONLY]
  [NO_INSTALL_LIB_OR_HEADERS]
  [CUDALIBRARY]
  )
```

Sections:

Formal Arguments (TRIBITS_ADD_LIBRARY())

`<libName>`

Required name of the library. This is the name passed to `ADD_LIBRARY(<libName> ...)`. The name is *not* prefixed by the packae name. CMake will of course add any standard prefix or post-fix to the library file name appropriate for the platform and if this is a static or shared library build.

`HEADERS <h0> <h1> ...`

List of public header files for using this library. By default, these header files are assumed to be in the current source directory. They can also contain the relative path or absolute path to the files if they are not in the current source directory. List list of headers is passed into `ADD_LIBRARY(...)` as well (which is not strictly needed but is helpful for some build tools, like MS Visual Stuido). By default, these headers will be installed as well (see Include Directories (TRIBITS_ADD_LIBRARY())).

`NOINSTALLHEADERS <nih0> <hih1> ...`

List of private header files which are used by this library. These headers are not installed and do not needed to be passed in for any purpose other than to pass them into `ADD_LIBRARY()` as some build tools like to have these listed (e.g. MS Visual Studio).

`SOURCES <src0> <src1> ...`

List of source files passed into `ADD_LIBRARY()` that are compiled into header files and included in the library. The compiler used to compile the files is determined automatically based on the file extension (see CMake documentation).

`DEPLIBS <deplib0> <deplib1> ...`

List of dependent libraries that are built in the current SE package that this library is dependent on. These libraries are passed into `TARGET_LINK_LIBRARIES(<libName> ...)` so that CMake knows about the dependency. You should **not** list libraries in other upstream SE packages or libraries built externally from this TriBITS CMake project. The TriBITS system automatically handles linking to libraries in uptream TriBITS packages and external libraries need to be listed in `IMPORTEDLIBS` instead.

`IMPORTEDLIBS <ideplib0> <ideplib1> ...`

List of dependent libraries built exteranlly from this TriBITS CMake project. These libraries are passed into `TARGET_LINK_LIBRARIES(<libName> ...)` so that CMake knows about the dependency. These libraries are added the `${PACKAGE_NAME}_LIBRARIES` so that downstream SE packages will also have these libraries and the link line also and these libraries will show up in the generated `Makefile.export.${PACKAGE_NAME}` and `${PACKAGE_NAME}Config.cmake` files if they are generated.

`TESTONLY`

If passed in, then `<libName>` will **not** be added to `${PACKAGE_NAME}_LIBRARIES` and an install target for the library will not be added. In this case, the current include directories will be set in the global variable `<libName>_INCLUDE_DIR` which will be used in TRIBITS_ADD_EXECUTABLE() when a test-only library is linked in.

`NO_INSTALL_LIB_OR_HEADERS`

If specified, then no install targets will be added for the library `<libName>` or the header files listed in `HEADERS`.

`CUDALIBRARY`

If specified then CUDA_ADD_LIBRARY() is used instead of ADD_LIBRARY() where CUDA_ADD_LIBRARY() is assumed to be defined by the standard FindCUDA.cmake module as processed using the standard TriBITS FindTPLCUDA.cmake file. For this option to work, this SE package must have an enabled direct or indirect dependency on the TriBITS CUDA TPL or a configure-time error will occur about not finding CUDA_ALL_LIBRARY().

Include Directories (TRIBITS_ADD_LIBRARY())

Any base directories for these header files listed in HEADERS or NOINSTALLHEADERS should be passed into INCLUDE_DIRECTORIES() *before* calling this function. These include directories will then be added to current packages list of include directories ${PACKAGE_NAME}_INCLUDE_DIRS.

Install Targets (TRIBITS_ADD_LIBRARY())

By default, an install target for the library is created using INSTALL(TARGETS <libName> ...) to install into the directory ${CMAKE_INSTALL_PREFIX}/lib/ (actual install directory is given by ${PROJECT}_INSTALL_LIB_DIR). However, this install target will not get created if ${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS=FALSE and BUILD_SHARD_LIBS=OFF. But when BUILD_SHARD_LIBS=ON, the install target will get created. Also, this install target will *not* get created if TESTONLY or NO_INSTALL_LIB_OR_HEADERS are passed in.

By default, an install target for the headers listed in HEADERS will get created using INSTALL(FILES <h1> <h2> ...), but only if TESTONLY and NO_INSTALL_LIB_OR_HEADERS are not passed in as well. These headers get installed into the flat directory ${CMAKE_INSTALL_PREFIX}/include/ (the actual install directory is given by ${PROJECT_NAME}_INSTALL_INCLUDE_DIR). Note that an install target will *not* get created for the headers listed in NOINSTALLHEADERS.

Additional Library and Source File Properties (TRIBITS_ADD_LIBRARY())

Once ADD_LIBRARY(<libName> ... <src0> <src1> ...) is called, one can set and change properties on the <libName> library target using SET_TARGET_PROPERTIES() as well as properties on any of the source files listed in SOURCES using SET_SOURCE_FILE_PROPERTIES() just like in any CMake project.

Miscellaneous Notes (TRIBITS_ADD_LIBRARY())

WARNING: Do **NOT** use ADD_DEFINITIONS() to add defines -D<someDefine> to the compile command line that will affect a header file! These defines are only set locally in this directory and child directories. These defines will **NOT** be set when code in peer directories (e.g. a downstream TriBIS pacakge) compiles code that may include these header files. To add defines, please use a configured header file (see TRIBITS_CONFIGURE_FILE()).

TRIBITS_ADD_EXECUTABLE()

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

Usage:

```
TRIBITS_ADD_EXECUTABLE(
  <exeRootName>  [NOEXEPREFIX]  [NOEXESUFFIX]  [ADD_DIR_TO_NAME]
  SOURCES <src0> <src1> ...
  [CATEGORIES <category0>  <category1> ...]
  [HOST <host0> <host1> ...]
  [XHOST <host0> <host1> ...]
  [HOSTTYPE <hosttype0> <hosttype1> ...]
  [XHOSTTYPE <hosttype0> <hosttype1> ...]
  [DIRECTORY <dir>]
  [DEPLIBS <lib0> <lib1> ...]
  [COMM [serial] [mpi]]
  [LINKER_LANGUAGE (C|CXX|Fortran)]
  [DEFINES -D<define0> -D<define1> ...]
```

```
  [INSTALLABLE]
  )
```

Sections:

Formal Arguments (TRIBITS_ADD_EXECUTABLE())

`<exeRootName>`

The root name of the exectuable (and CMake target) (see Executable and Target Name (TRIBITS_ADD_EXECUTABLE())).

`NOEXEPREFIX`

If passed in, then `${PACKAGE_NAME}_` is not added the beginning of the executable name (see Executable and Target Name (TRIBITS_ADD_EXECUTABLE())).

`NOEXESUFFIX`

If passed in, then `${${PROJECT_NAME}_CMAKE_EXECUTABLE_SUFFIX}` and not added to the end of the executable name (see Executable and Target Name (TRIBITS_ADD_EXECUTABLE())).

`ADD_DIR_TO_NAME`

If passed in, the directory path relative to the package base directory (with "/" replaced by "_") is added to the executable name (see Executable and Target Name (TRIBITS_ADD_EXECUTABLE())). This provides a simple way to create unique test exectuable names inside of a given TriBITS package. Only test executables in the same directory would need to have unique `<execRootName>` passed in.

`SOURCES <src0> <src1> ...`

Gives the source files that will be compiled into the built executable. By default, these sources are assumed to be in the current working directory or gives the relative path to the current working directory. If `<srci>` is an absolute path, then that full file path is used. This list of sources (with adjusted directory path) are passed into `ADD_EXECUTABLE(<fullExeName> ... )`. After calling this function, the properties of the source files can be altered using `SET_SOURCE_FILE_PROPERTIES()`.

`DIRECTORY <dir>`

If specified, then the soruces for the exectuable listed in `SOURCES <src0> <src1> ...` are assumed to be in the relative or absolute directory `<dir>` instead of the current source directory. This directrory path is prepended to each source file name `<srci>` unless `<srci>` is an absolute path.

`CATEGORIES <category0> <category1> ...`

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

`HOST <host0> <host1> ...`

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

`XHOST <host0> <host1> ...`

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

`HOSTTYPE <hosttype0> <hosttype1> ...`

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

`XHOSTTYPE <hosttype0> <hosttype1> ...`

The list of host types for which **not** to enable the test (see TRIBITS_ADD_TEST()).

`DEPLIBS <lib0> <lib1> ...`

Specifies extra libraries that will be linked to the executable using `TARGET_LINK_LIBRARY()`. Note that regular libraries (i.e. not "TESTONLY") defined in the current SE package or any upstream SE packages do **NOT** need to be listed! TriBITS automatically links these libraries to the executable! The only libraries that should be listed in this argument are either `TESTONLY` libraries, or other libraries that are built external from this CMake project and are not provided through a proper TriBITS TPL. The latter usage is not recommended. External TPLs should be handled as a declared TriBITS TPL. For a `TESTONLY` library, the include directories will automatically be added using:

```
INCLUDE_DIRECTORIES(${<libi>_INCLUDE_DIRS})
```

where `<libi>_INCLUDE_DIRS` was set by:

```
TRIBITS_ADD_LIBRARY(<libi> ... TESTONLY ...)
```

Therefore, to link to a defined `TESTONLY` library in any upstream enabled package, one just needs to pass in the library name through `DEPLIBS ... <libi> ...` and that is it!

`COMM [serial] [mpi]`

If specified, selects if the test will be added in serial and/or MPI mode. See the `COMM` argument in the script TRIBITS_ADD_TEST() for more details.

`LINKER_LANGUAGE (C|CXX|Fortran)`

If specified, overrides the linker language used by setting the target property `LINKER_LANGUAGE`. By default, CMake choses the compiler to be used as the linker based on file extensions. The most typical use case is when Fortran-only or C-only sources are passed in through `SOURCES` but a C++ linker is needed because there are upstream C++ libraries.

`DEFINES -D<define0> -D<define1> ...`

Add the listed defines using `ADD_DEFINITIONS()`. These should only affect the listed sources for the built executable and not other compiles in this directory due to the FUNCTION scoping.

`INSTALLABLE`

If passed in, then an install target will be added to install the built exectuable into the `${CMAKE_INSTALL_PREFIX}/bin/` directory (see Install Target (TRIBITS_ADD_EXECUTABLE())).

Executable and Target Name (TRIBITS_ADD_EXECUTABLE())

By default, the full name of the executable and target name `<fullExecName>` =

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

If `ADD_DIR_TO_NAME` is set, then the directory path relative to the package base directory (with "/" replaced with "_"), or `<relDirName>`, is added to the executable name to form `<fullExecName>` =

```
${PACKAGE_NAME}_<relDirName>_<exeRootName>
```

If the option NOEXEPREFIX is pased in, the prefix ${PACKAGE_NAME}_ is removed.

CMake will add the executable suffix ${${PROJECT_NAME}_CMAKE_EXECUTABLE_SUFFIX} the actual executable file if the option NOEXESUFFIX is not passed in but this suffix is never added to the target name.

The reason that a default prefix is prepended to the executable and target name is because the primary reason to create an executable is typically to create a test or an example that is private to the package. This prefix helps to namespace the exexutable and its target so as to avoid name clashes with targets in other packages. It also helps to avoid clashes if the executable gets installed into the install directory (if INSTALLABLE is specified).

Additional Executable and Source File Properties (TRIBITS_ADD_EXECUTABLE())

Once ADD_EXECUTABLE(<fullExeName> ...) is called, one can set and change properties on the <fullExeName> executable target using SET_TARGET_PROPERTIES() as well as properties on any of the source files listed in SOURCES using SET_SOURCE_FILE_PROPERTIES() just like in any CMake project.

Install Target (TRIBITS_ADD_EXECUTABLE())

If INSTALLABLE is passed in, then an install target INSTALL(TARGETS <fullExeName> ...) is added to install the built executable into the ${CMAKE_INSTALL_PREFIX}/bin/ directory (actual install directory path is determined by ${PROJECT_NAME}_INSTALL_RUNTIME_DIR).

TRIBITS_ADD_EXECUTABLE_AND_TEST()

Add an executable and a test (or several tests) all in one shot.

Usage:

```
TRIBITS_ADD_EXECUTABLE_AND_TEST(
  <exeRootName>  [NOEXEPREFIX]  [NOEXESUFFIX]  [ADD_DIR_TO_NAME]
  SOURCES <src0> <src1> ...
  [NAME <testName> | NAME_POSTFIX <testNamePostfix>]
  [CATEGORIES <category0>  <category1> ...]
  [HOST <host0> <host1> ...]
  [XHOST <xhost0> <xhost1> ...]
  [XHOST_TEST <xhost0> <xhost1> ...]
  [HOSTTYPE <hosttype0> <hosttype1> ...]
  [XHOSTTYPE <xhosttype0> <xhosttype1> ...]
  [XHOSTTYPE_TEST <xhosttype0> <xhosttype1> ...]
  [DIRECTORY <dir>]
  [DEFINES -DS<someDefine>]
  [DEPLIBS <lib0> <lib1> ... ]
  [COMM [serial] [mpi]]
  [ARGS "<arg0> <arg1> ..." "<arg2> <arg3> ..." ...]
  [NUM_MPI_PROCS <numProcs>]
  [LINKER_LANGUAGE (C|CXX|Fortran)]
  [STANDARD_PASS_OUTPUT
    | PASS_REGULAR_EXPRESSION "<regex0>;<regex1>;..."]
  [FAIL_REGULAR_EXPRESSION "<regex0>;<regex1>;..."]
  [WILL_FAIL]
  [ENVIRONMENT <var0>=<value0> <var1>=<value1> ...]
  [INSTALLABLE]
  [TIMEOUT <maxSeconds>]
  )
```

This function takes a fairly common set of arguments to TRIBITS_ADD_EXECUTABLE() and TRIBITS_ADD_TEST() but not the full set passed to TRIBITS_ADD_TEST(). See the documentation for TRIBITS_ADD_EXECUTABLE() and TRIBITS_ADD_TEST() to see which arguments are accpeted by which functions.

Arguments that are specific to this function and not contained in `TRIBITS_ADD_EXECUTABLE()` or `TRIBITS_ADD_TEST()` include:

`XHOST_TEST <xhost0> <xhost1> ...`

When specified, this disables just running the tests for the named hosts `<xhost0>`, `<xhost0>` etc. but still builds the executable for the test.

`XHOSTTYPE_TEST <xhosttype0> <hosttype1> ...`

When specified, this disables just running the tests for the named host types `<hosttype0>`, `<hosttype0>`, ..., but still builds the executable for the test.

This is the function to use for simple test executbles that you want to run that either takes no arguments or just a simple set of arguments passed in through `ARGS`.

TRIBITS_ADD_TEST()

Add a test or a set of tests for a single executable or command.

Usage:

```
TRIBITS_ADD_TEST(
  <exeRootName>  [NOEXEPREFIX]  [NOEXESUFFIX]
  [NAME <testName> | NAME_POSTFIX <testNamePostfix>]
  [DIRECTORY <directory>]
  [ADD_DIR_TO_NAME]
  [ARGS "<arg0> <arg1> ..." "<arg2> <arg3> ..." ...
    | POSTFIX_AND_ARGS_0 <postfix0> <arg0> <arg1> ...
      POSTFIX_AND_ARGS_1 ... ]
  [COMM [serial] [mpi]]
  [NUM_MPI_PROCS <numProcs>]
  [CATEGORIES <category0>  <category1> ...]
  [HOST <host0> <host1> ...]
  [XHOST <host0> <host1> ...]
  [HOSTTYPE <hosttype0> <hosttype1> ...]
  [XHOSTTYPE <hosttype0> <hosttype1> ...]
  [STANDARD_PASS_OUTPUT
    | PASS_REGULAR_EXPRESSION "<regex0>;<regex1>;..."]
  [FAIL_REGULAR_EXPRESSION "<regex0>;<regex1>;..."]
  [WILL_FAIL]
  [ENVIRONMENT <var0>=<value0> <var1>=<value1> ...]
  [TIMEOUT <maxSeconds>]
  )
```

Sections:

- Disabling Tests Externally (TRIBITS_ADD_TEST())

Formal Arguments (TRIBITS_ADD_TEST())

`<exeRootName>`

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

`NOEXEPREFIX`

If specified, then the prefix `${PACKAGE_NAME}_` is not assumed to be prepended to `<exeRootName>`.

`NOEXESUFFIX`

If specified, then the postfix `${${PROJECT_NAME}_CMAKE_EXECUTABLE_SUFFIX}` is not assumed to be post-pended to `<exeRootName>`.

`NAME <testRootName>`

If specified, gives the root name of the test. If not specified, then `<testRootName>` is taken to be `<exeRootName>`. The actual test name will always prefixed as `${PACKAGE_NAME}_<testRootName>` passed into the call to the built-in CMake command `ADD_TEST(...)`. The main purpose of this argument is to allow multiple tests to be defined for the same executable. CTest requires all test names to be globally unique in a single project.

`NAME_POSTFIX <testNamePostfix>`

If specified, gives a postfix that will be added to the standard test name based on `<exeRootName>` (appended as `_<NAME_POSTFIX>`). If the `NAME <testRootName>` argument is given, this argument is ignored.

`DIRECTORY <dir>`

If specified, then the executable is assumed to be in the directory given by by `<dir>`. The directory `<dir>` can either be a relative or absolute path. If not specified, the executable is assumed to be in the current bindary directory.

`ADD_DIR_TO_NAME`

If specified, then the directory name that this test resides in will be added into the name of the test after the package name is added and before the root test name (see below). The directory will have the package's base directory stripped off so only the unique part of the test directory will be used. All directory seperators will be changed into underscores.

`RUN_SERIAL`

If specified then no other tests will be allowed to run while this test is running. This is useful for devices(like cuda cards) that require exclusive access for processes/threads. This just sets the CTest test property `RUN_SERIAL` using the built-in CMake function `SET_TESTS_PROPERTIES()`.

`ARGS "<arg0> <arg1> ..." "<arg2> <arg3> ..." ...`

If specified, then a set of arguments can be passed in quotes. If multiple groups of arguments are passed in different quoted clusters of arguments then a different test will be added for each set of arguments. In this way, many different tests can be added for a single executable in a single call to this function. Each of these separate tests will be named `${TEST_NAME}_xy` where `xy` = `00`, `01`, `02`, and so on. **WARNING:** When

defining multiple tests it is prefered to use the POSTFIX_AND_ARGS_<IDX> form instead. **WARNING:** Multiple arguments passed to a single test invocation must be quoted or multiple tests taking single arguments will be created instead! See Adding Multiple Tests (TRIBITS_ADD_TEST()) for more details and exmaples.

POSTFIX_AND_ARGS_<IDX> <postfix> <arg0> <arg1> ...

If specified, gives a sequence of sets of test postfix names and arguments lists for different tests (up to POSTFIX_AND_ARGS_19). For example, a set of three different tests with argument lists can be specified as:

```
POSTIFX_AND_ARGS_0 postfix0 --arg1 --arg2="dummy"
POSTIFX_AND_ARGS_1 postfix1  --arg2="fly"
POSTIFX_AND_ARGS_2 postfix2  --arg2="bags"
```

This will create three different test cases with the postfix names postfix0, postfix1, and postfix2. The indexes must be consecutive starting a 0 and going up to (currently) 19. The main advantages of using these arguments instead of just 'ARGS' are that you can give meaningful name to each test case and you can specify multiple arguments without having to quote them and you can allow long argument lists to span multiple lines. See Adding Multiple Tests (TRIBITS_ADD_TEST()) for more details and exmaples.

COMM [serial] [mpi]

If specified, selects if the test will be added in serial and/or MPI mode. If the COMM argument is missing, the test will be added in both serial and MPI builds of the code.

NUM_MPI_PROCS <numProcs>

If specified, gives the number of processes that the test will be defined to run. If <numProcs> is greater than ${MPI_EXEC_MAX_NUMPROCS} then the test will be excluded. If not specified, then the default number of processes for an MPI build will be ${MPI_EXEC_DEFAULT_NUMPROCS}. For serial builds, this argument is ignored.

HOST <host0> <host1> ...

If specified, gives a list of hostnames where the test will be included. The current hostname is determined by the built-in CMake command SITE_NAME(${PROJECT_NAME}_HOSTNAME). On Linux/Unix systems, this is typically the value returned by 'uname -n'. If this list is given, the value of ${${PROJECT_NAME}_HOSTNAME} must equal one of the listed host names <hosti> or test will not be added. The value of ${PROJECT_NAME}_HOSTNAME gets printed out in the TriBITS cmake output under the section Probing the environment.

XHOST <host0> <host1> ...

If specified, gives a list of hostnames (see HOST argument) where the test will *not* be added. This check is performed after the check for the hostnames in the HOST list if it should exist. Therefore, this list exclusion list overrides the 'HOST' inclusion list.

CATEGORIES <category0> <category1> ...

If specified, gives the specific categories of the test. Valid test categories include BASIC, CONTINUOUS, NIGHTLY, WEEKLY and PERFORMANCE. By default, the category is BASIC. When the test category does not match ${PROJECT_NAME}_TEST_CATEGORIES, then the test is not added. When the CATEGORIES is BASIC it will match ${PROJECT_NAME}_TEST_CATEGORIES eqaual to CONTINUOUS, NIGHTLY, and WEEKLY. When the CATEGORIES contains CONTINUOUS it will match ${PROJECT_NAME}_TEST_CATEGORIES equal to CONTINUOUS, NIGHTLY, and WEEKLY. When the CATEGORIES is NIGHTLY it will match ${PROJECT_NAME}_TEST_CATEGORIES equal to NIGHTLY and WEEKLY. When the CATEGORIES is PERFORMANCE it will match ${PROJECT_NAME}_TEST_CATEGORIES=PERFORMANCE only.

`HOSTTYPE <hosttype0> <hosttype1> ...`

If specified, gives the names of the host system type (given by `CMAKE_HOST_SYSTEM_NAME` which is printed in the TriBITS cmake confgiure output in the section `Probing the environment`) to include the test. Typical host system type names include `Linux`, `Darwain` etc.

`XHOSTTYPE <hosttype0> <hosttype1> ...`

If specified, gives the names of the host system type to *not* include the test. This check is performed after the check for the host system names in the `HOSTTYPE` list if it should exist. Therefore, this list exclusion list overrides the `HOSTTYPE` inclusion list.

`STANDARD_PASS_OUTPUT`

If specified, then the standard test output `End Result:  TEST PASSED` is greped for to determine success. This is needed for MPI tests on some platforms since the return value is unreliable. This is set using the built-in ctest property `PASS_REGULAR_EXPRESSION`.

`PASS_REGULAR_EXPRESSION "<regex0>;<regex1>;..."`

If specified, then a test will be assumed to pass only if one of the regular expressions `<regex0>`, `<regex1>` etc. match the output. Otherwise, the test will fail. This is set using the built-in test property `PASS_REGULAR_EXPRESSION`. Consult standard CMake documentation.

`FAIL_REGULAR_EXPRESSION "<regex0>;<regex1>;..."`

If specified, then a test will be assumed to fail if one of the regular expressions `<regex0>`, `<regex1>` etc. match the output. Otherwise, the test will pass. This is set using the built-in test property `FAIL_REGULAR_EXPRESSION`.

`WILL_FAIL`

If passed in, then the pass/fail criteria will be inverted. This is set using the built-in test property `WILL_FAIL`.

`ENVIRONMENT <var0>=<value0> <var1>=<value1> ...`

If passed in, the listed environment varaibles will be set before calling the test. This is set using the built-in test property `ENVIRONMENT`.

`TIMEOUT <maxSeconds>`

If passed in, gives maximum number of seconds the test will be allowed to run beforebeing timed-out. This sets the test property `TIMEOUT`. **WARNING:** Rather than just increasing the timeout for an expensive test, please try to either make the test run faster or relegate the test to being run less often (i.e. set `CATEGORIES NIGHTLY` or even `WEEKLY` for extremently expensive tests). Expensive tests are one of the worse forms of technical debt that a project can have!

In the end, this function just calls the built-in CMake commands `ADD_TEST(${TEST_NAME} ...)` and `SET_TESTS_PROPERTIES(${TEST_NAME} ...)` to set up a executable process for `ctest` to run, determine pass/fail criteria, and set some other test properties. Therefore, this wrapper funtion does not provide any fundamentally new features that are already avaiable in the basic usage if CMake/CTest. However, this wrapper function takes care of many of the details and boiler-plate CMake code that it takes to add such a test (or tests) and enforces consistency across a large project for how tests are defined, run, and named (to avoid test name clashes).

If more flexibility or control is needed when defining tests, then the function `TRIBITS_ADD_ADVANCED_TEST()` should be used instead.

In the following subsections, more details on how tests are defined and run is given.

Determining the Exectuable or Command to Run (TRIBITS_ADD_TEST())

This funtion is primarily designed to make it easy to run tests for exectaubles built using the function TRIBITS_ADD_EXECUTABLE(). To set up tests to run arbitrary executables, see below.

By default, the command to run for the executable is determined by first getting the exectuable name which by default is assumed to be `<fullExeName>` =:

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

which is (by no coincidence) idential to how it is selected in TRIBITS_ADD_EXECUTABLE(). This name can be alterned by passing in `NOEXEPREFIX`, `NOEXESUFFIX`, and `ADD_DIR_TO_NAME` as described in Executable and Target Name (TRIBITS_ADD_EXECUTABLE()).

By default, this executable is assumed to be in the current CMake binary directory `${CMAKE_CURRENT_BINARY_DIR}` but the directory location can be changed using the `DIRECTORY <dir>` argument.

If an arbitrary exectuable is to be run for the test, then pass in `NOEXEPREFIX` and `NOEXESUFFIX` and set `<exeRootName>` to the relative or absolute path of the exeutable to be run. If `<exeRootName>` is not an absolute path, then `${CMAKE_CURRENT_BINARY_DIR}/<exeRootName>` is set as the executable to run.

Whatever executable path is specified using this logic, if the executable is not found, then when `ctest` goes to run the test, it will mark it as `NOT RUN`.

Determining the Full Test Name (TRIBITS_ADD_TEST())

By default, the base test name is selected to be `<fullTestName>` =

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

If `NAME <testRootName>` is passed in, then `<testRootName>` is used instead of `<exeRootName>`.

If `NAME_POSTFIX <testNamePostfix>` is passed in, then the base test name is selected to be `<fullTestName` =

```
${PACKAGE_NAME}_<exeRootName>_<testNamePostfix>
```

If `ADD_DIR_TO_NAME` is passed in, then the directory name realtive to the package directory name is added to the name as well to help disambiguate the test name (see the above).

Let the test name determined by this process be `TEST_NAME`. If no arguments or one set of arguments are passed in through `ARGS`, then this is the test name actaully passed in to `ADD_TEST()`. If multiple tests are defined, then this name becomes the base test name for each of the tests. See below.

Finally, for any test that gets defined, if MPI is enabled (i.e. `TPL_ENABLE_MPI=ON`), then the terminal suffix *_MPI_${NUM_MPI_PROCS}* will be added to the end of the test name (even for multiple tests). No such prefix is added for the serial case (i.e. `TPL_ENABLE_MPI=OFF`).

Adding Multiple Tests (TRIBITS_ADD_TEST())

Using this function, one can add exectuable arguments and can even add multiple tests in one of two ways. One can either pass in 1 or more **quoted** clusters of arguments using:

```
ARGS "<arg0> <arg1> ..." "<arg2> <arg3> ..." ...
```

or can pass in an explicit test name postfix and arguments with:

```
POSTFIX_AND_ARGS_0 <postfix0> <arg0> <arg1> ...
POSTFIX_AND_ARGS_1 <postfix1> <arg2> ...
...
```

If only one short set of arguments needs to be passed in, then passing:

```
ARGS "<arg0> <arg1>"
```

may be preferable since it will not add any postfix name to the test. To add more than one test case using `ARGS`, you use more than one quoted set of arugments such as with:

```
ARGS "<arg0> <arg1>" "<arg2> <arg2>"
```

which creates 2 tests with the names `<fullTestName>_00` passing arguments `"<arg0> <arg1>"` and `<fullTestName>_01` passing arguments `"<arg2> <arg3>"`. However, when passing multiple sets of arguments it is preferable to **not** use `ARGS` but instead use:

```
POSTFIX_AND_ARGS_0 test_a <arg0> <arg1>
POSTFIX_AND_ARGS_1 test_b <arg2> <arg2>
```

which also creates the same 2 tests but now with the improved names `<fullTestName>_test_a` passing arguments `"<arg0> <arg1>"` and `<fullTestName>_test_b` passing arguments `"<arg2> <arg3>"`. In this way, the individual tests can be given more understandable names.

The other advantage of the `POSTFIX_AND_ARGS_<IDX>` form is that the arugments `<arg0>`, `<arg1>`, ... do not need to be quoted and can therefore be extended over multiple lines like:

```
POSTFOX_AND_ARGS_0 long_args --this-is-the-first-long-arg=very
  --this-is-the-second-long-arg=verylong
```

If you don't use quotes when using `ARGS` you actually get more than one test. For example, if you pass in:

```
ARGS --this-is-the-first-long-arg=very
  --this-is-the-second-long-arg=verylong
```

you actually get two tests, not one test. This is a common mistake that people make when using the `ARGS` form of passing arguments. This can't be fixed or it will break backward compatibility. If this could be designed fresh, the `ARGS` argument would only create a single test and the arguments would not be quoted.

Determining Pass/Fail (TRIBITS_ADD_TEST())

The only means to determine pass/fail is to use the built-in test properties `PASS_REGULAR_EXPRESSION` and `FAIL_REGULAR_EXPRESSION` which can only grep STDOUT/STDERR or to check for a 0 return value (or invert these using `WILL_FAIL`). For simple tests, that is enough. However, for more complex executables, one may need to examine the output files to determine pass fail. Raw CMake/CTest cant' do this. In this case, one should use TRIBITS_ADD_ADVANCED_TEST().

Setting additional test properties (TRIBITS_ADD_TEST())

After this function returns, any tests that get added using `ADD_TEST()` can have additional properties set and changed using `SET_TEST_PROPERTIES()`. Therefore, any tests properties that are not directly supported by this function and passed through this wrapper function can be set in the outer `CMakeLists.txt` file after the call to `TRIBITS_ADD_TEST()`.

ToDo: Describe how to use new variable ADDED_TESTS_OUT to get the list of tests actually added (if they are added) in order to make it easy to set additional test properties.

Debugging and Examining Test Generation (TRIBITS_ADD_TEST())

In order to see what tests are getting added and to debug some issues in test creation, one can set the cache variable `${PROJECT_NAME}_VERBOSE_CONFIGURE=ON`. This will result in the printout of some information about the test getting added or not.

Also, CMake writes a file `CTestTestfile.cmake` in the current binary directory which contains all of the added tests and test properties that are set. This is the file that is read by `ctest` when it runs to determine what tests to run. In that file, one can see the exact `ADD_TEST()` and `SET_TEST_PROPERTIES()` commands. The is the ultimate way to debug exactly what tests are getting added by this function.

Disabling Tests Externally (TRIBITS_ADD_TEST())

The test can be disabled externally by setting the CMake cache variable `${FULL_TEST_NAME}_DISABLE=TRUE`. This allows tests to be disable on a case-by-case basis. This is the *exact* name that shows up in 'ctest -N' when running the test. If multiple tests are added in this funtion through multiple argument sets to `ARGS` or through multiple `POSTFIX_AND_ARGS_<IDX>` arguments, then `${FULL_TEST_NAME}_DISABLE=TRUE` must be set for each test individually.

TRIBITS_ADD_ADVANCED_TEST()

Function that creates an advanced test defined by stringing together one or more executables and/or commands that is run as a separate CMake -P script with very flixible pass/fail criteria.

This function allows you to add a single CTest test as a single unit that is actually a sequence of one or more separate commands strung together in some way to define the final pass/fail. You will want to use this function to add a test instead of `TRIBITS_ADD_TEST()` when you need to run more than one command, or you need more sophisticated checking of the test result other than just greping STDOUT (i.e. by running programs to examine output files).

Usage:

```
TRIBITS_ADD_ADVANCED_TEST(
  <testName>
  TEST_0 (EXEC <execTarget0> | CMND <cmndExec0>) ...
  [TEST_1 (EXEC <execTarget1> | CMND <cmndExec1>) ...]
  ...
  [TEST_N (EXEC <execTargetN> | CMND <cmndExecN>) ...]
  [OVERALL_WORKING_DIRECTORY (<overallWorkingDir> | TEST_NAME)]
  [FAIL_FAST]
  [KEYWORDS <keyword1> <keyword2> ...]
  [COMM [serial] [mpi]]
  [OVERALL_NUM_MPI_PROCS <overallNumProcs>]
  [CATEGORIES <category0> <category1> ...]
  [HOST <host0> <host1> ...]
  [XHOST <host0> <host1> ...]
  [HOSTTYPE <hosttype0> <hosttype1> ...]
  [XHOSTTYPE <hosttype0> <hosttype1> ...]
  [FINAL_PASS_REGULAR_EXPRESSION <regex> | FINAL_FAIL_REGULAR_EXPRESSION <regex>]
  [ENVIRONMENT <var1>=<value1> <var2>=<value2> ...]
  )
```

Each atomic test case is either a package-built executable or just a basic command. An atomic test command 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:

Overall Arguments (TRIBITS_ADD_ADVANCED_TEST())

Below are given some ome overall arguments are. Remaining overall arguments that control overall pass/fail are described in Overall Pass/Fail (TRIBITS_ADD_ADVANCED_TEST()).

`<testName>`

The name of the test (which will have `${PACKAGE_NAME}_` prepended to the name) that will be used to name the output CMake script file as well as the CTest test name passed into `ADD_TEST()`. This must be the first argument.

`OVERALL_WORKING_DIRECTORY <overallWorkingDir>`

If specified, then the working directory `<overallWorkingDir>` will be created and all of the test commands by default will be run from within this directory. If the value `<overallWorkingDir>=TEST_NAME` is given, then the working directory will be given the name `${PACKAGE_NAME}_<testName>`. If the directory `<overallWorkingDir>` exists before the test runs, it will be deleted and created again. Therefore, if you want to preserve the contents of this directory between test runs you need to copy the files it somewhere else. This is a good option to use if the commands create intermediate files and you want to make sure they get deleted before a set of test cases runs again.

`FAIL_FAST`

If specified, then the remaining test commands will be aborted when any test command fails. Otherwise, all of the test cases will be run.

`RUN_SERIAL`

If specified then no other tests will be allowed to run while this test is running. This is useful for devices(like cuda cards) that require exclusive access for processes/threads. This just sets the CTest test property `RUN_SERIAL` using the built-in CMake function `SET_TESTS_PROPERTIES()`.

`COMM [serial] [mpi]`

If specified, selects if the test will be added in serial and/or MPI mode. See the `COMM` argument in the script TRIBITS_ADD_TEST() for more details.

`OVERALL_NUM_MPI_PROCS <overallNumProcs>`

If specified, gives the default number of processes that each executable command runs on. If `<numProcs>` is greater than `${MPI_EXEC_MAX_NUMPROCS}` then the test will be excluded. If not specified, then the default number of processes for an MPI build will be `${MPI_EXEC_DEFAULT_NUMPROCS}`. For serial builds, this argument is ignored.

`CATEGORIES <category0> <category1> ...`

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

`HOST <host0> <host1> ...`

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

`XHOST <host0> <host1> ...`

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

`HOSTTYPE <hosttype0> <hosttype1> ...`

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

`XHOSTTYPE <hosttype0> <hosttype1> ...`

The list of host types for which **not** to enable the test (see TRIBITS_ADD_TEST()).

`ENVIRONMENT <var1>=<value1> <var2>=<value2> ...`

If passed in, the listed environment varaibles will be set before calling the test. This is set using the built-in test property `ENVIRONMENT`.

TEST_<IDX> Test Blocks and Arguments (TRIBITS_ADD_ADVANCED_TEST())

Each test command block `TEST_<IDX>` runs either a package-built test executable or some general command executable and is defined as either `EXEC <exeRootName>` or `CMND <cmndExec>`:

`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 (TRIBITS_ADD_TEST())). If this is an MPI build, then the executable will be run with MPI using `NUM_MPI_PROCS <numProcs>` or `OVERALL_NUM_MPI_PROCS <overallNumProcs>` (if `NUM_MPI_PROCS` is not set for this test case). If the number of maximum MPI processes allowed is less than this number of MPI processes, then the test will *not* be run. Note that `EXEC <exeRootName>` 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())

Since raw CTest does not support the features provided by this function, the way an advanced test is implemented is that a CMake script with the name `${PACKAGE_NAME}_<testName>.cmake` gets created in the current bindary directory that then gets added to CTest using:

```
ADD_TEST(cmake -P ${PACKAGE_NAME}_<testName>.cmake)
```

This CMake script then runs the various test cases and checks the pass/fail for each case to determine overall pass/fail and implement other functionality.

Setting Additional Test Properties (TRIBITS_ADD_ADVANCED_TEST())

After this function returns, if the test gets added using `ADD_TEST()` then additional properties can be set and changed using `SET_TEST_PROPERTIES(${PACKAGE_NAME}_<testName> ...)`. Therefore, any tests properties that are not directly supported by this function and passed through the argument list to this wrapper function can be set in the outer `CMakeLists.txt` file after the call to `TRIBITS_ADD_ADVANCED_TEST()`.

Disabling Tests Externally (TRIBITS_ADD_ADVANCED_TEST())

The test can be disabled externally by setting the CMake cache variable `${FULL_TEST_NAME}_DISABLE=TRUE`. This allows tests to be disable on a case-by-case basis. This is the *exact* name that shows up in 'ctest -N' when running the test.

Debugging and Examining Test Generation (TRIBITS_ADD_ADVANCED_TEST())

In order to see if the test gets added and to debug some issues in test creation, one can set the cache variable `${PROJECT_NAME}_VERBOSE_CONFIGURE=ON`. This will result in the printout of some information about the test getting added or not.

Likely the best way to debugging test generation using this function is to examine the generated file `${PACKAGE_NAME}_<testName>.cmake` in the current binary directory (see Implementation Details (TRIBITS_ADD_ADVANCED_TEST())).

TRIBITS_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.

13 General Utility Macros and Functions

The following subsections give detailed documentation for some CMake macros and functions which are *not* a core part of the TriBITS system but are included in the TriBITS system that are used inside of the TriBITS system and are provided as a convenience to TriBITS project developers. One will see many of these functions and macros used throughout the implementation of TriBITS and even in the `CMakeLists.txt` files for projects that use TriBITS.

These macros and functions are *not* prefixed with `TRIBITS_`. There is really not a large risk to defining and using these non-namespaces utility functions and macros. It turns out that CMake allows you to redefine any macro or function, even built-in ones, inside of your project so even if CMake did add new comamnds that clashed with these names, there would be no conflicit. When overridding a built-in command `some_builtin_command()`, you can always access the original built-in command as `_some_builtin_command()`.

ADD_SUBDIRECTORIES()

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

Usage:

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

ADVANCED_OPTION()

Macro that sets an option and marks it as advanced (removes boiler-plate and duplication).

Usage:

```
ADVANCED_OPTION(<varName> [other arguments])
```

This is identical to:

```
ADVANCED_OPTION(<varName> [other arguments])
MARK_AS_ADVANCED(<varName>)
```

ADVANCED_SET()

Macro that sets a variable and marks it as advanced (removes boiler-plate and duplication).

Usage:

```
ADVANCED_SET(<varName> [other arguments])
```

This is identical to:

```
ADVANCED_SET(<varName> [other arguments])
MARK_AS_ADVANCED(<varName>)
```

APPEND_CMNDLINE_ARGS()

Utility function that appends command-line arguments to a variable of command-line options.

Usage:

```
APPEND_CMNDLINE_ARGS(<var> "<extraArgs>")
```

This function just appends the command-line arguments in the string `"<extraArgs>"` but does not add an extra space if `<var>` is empty on input.

APPEND_GLOB()

Utility macro that does a `FILE(GLOB ...)` and appends to an existing list (removes boiler-plate code).

Usage:

```
APPEND_GLOB(<fileListVar> <glob0> <glob1> ...)
```

On output, `<fileListVar>` will have the list of glob files appended.

APPEND_GLOBAL_SET()

Utility macro that appends arguments to a global variable (reduces boiler-plate code and mistakes).

Usage:

```
APPEND_GLOBAL_SET(<varName> <arg0> <arg1> ...)
```

NOTE: The variable `<varName>` must exist before calling this function. To set it empty initially use GLOBAL_NULL_SET().

APPEND_SET()

Utility function to append elements to a varible (reduces boiler-plate code).

Usage:

```
APPEND_SET(<varName> <arg0> <arg1> ...)
```

Just calls:

```
LIST(APPEND <varName> <arg0> <arg1> ...)
```

APPEND_STRING_VAR()

Append strings to an existing string variable (reduces boiler-place code and reduces mistakes).

Usage:

```
APPEND_STIRNG_VAR(<stringVar> "<string1>" "<string2>" ...)
```

Note that the usage of the characters `'['`, `']'`, `'{'`, `'}'` are taken by CMake to bypass the meaning of ';' to separate string characters.

If you want to ignore the meaning of these special characters and are okay with just adding one string at a time use APPEND_STRING_VAR_EXT().

APPEND_STRING_VAR_EXT()

Append a single string to an existing string variable, ignoring ';' (reduces boiler-place code and reduces mistakes).

Usage:

```
APPEND_STIRNG_VAR_EXT(<stringVar> "<string>")
```

Simply sets `<stringVar> = "${<stringVar>}<string>"`.

APPEND_STRING_VAR_WITH_SEP()

Append strings to a given string varible, joining them using a seperator.

Usage:

```
APPEND_STRING_VAR_WITH_SEP(<stringVar> "<sepStr>" "<str0>" "<str>" ...)
```

Each of the strings `<stri>` are appended to `<stringVar>` using the separation string `<sepStr>`.

ASSERT_DEFINED()

Assert that a varaible is defined and if not call `MESSAGE(SEND_ERROR ...)`.

Usage:

```
ASSERT_DEFINED(<varName>)
```

This is used to get around the problem of CMake not asserting the defreferencing of undefined varibles. For example, how do you know if you did not mispell the name of a varible in an if statement like:

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

? If you mispelled the varible `SOME_VARBLE` (which you likely did in this case), the the if statement will always be false. To avoid this problem when you always expect the explicitly set, instead do:

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

Now if you misspell the varible, it will asset and stop processing. This is not a perfect solution since you can mispell the varible name in the following if statemnt but typically you would always just copy and paste between the two statements so they are always the same. This is the best we can do in CMake unfortunately.

COMBINED_OPTION()

Set up a bool cache variable (i.e. an option) based on a set of dependent options.

Usage:

```
COMBINED_OPTION( <combinedOptionName>
  DEP_OPTIONS_NAMES <depOpName0> <depOptName1> ...
  DOCSTR "<docstr0>" "<docstr1" ...
  )
```

This sets up a bool cache variable `<combinedOptionName>` which is defaulted to `ON` if all of the listed dependent option varaibles `<depOpName0>`, `<depOptName1>`, ... are all `ON`. However, if `<combinedOptionName>` is set to `ON` by the user and not all of the dependent option varibles are also true, this results in a fatal eror and alll processing stops.

This is used by a CMake project to by default automatically turn on a feature that requires a set of other features to also be turned on but allows a user to disable the feature if desired.

CONCAT_STRINGS()

Concatenate a set of string argumnets.

Usage:

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

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

DUAL_SCOPE_APPEND_CMNDLINE_ARGS()

Utility function that appends command-line arguments to a variable of command-line options and sets the result in current scope and parent scope.

Usage:

```
DUAL_SCOPE_APPEND_CMNDLINE_ARGS(<var> "<extraArgs>")
```

Just calls APPEND_CMNDLINE_ARGS() and then `SET(<var> ${<var>} PARENT_SCOPE)`.

DUAL_SCOPE_PREPEND_CMNDLINE_ARGS()

Utility function that prepends command-line arguments to a variable of command-line options and sets the result in current scope and parent scope.

Usage:

```
DUAL_SCOPE_PREPEND_CMNDLINE_ARGS(<var> "<extraArgs>")
```

Just calls PREPEND_CMNDLINE_ARGS() and then `SET(<var> ${<var>} PARENT_SCOPE)`.

DUAL_SCOPE_SET()

Macro that sets a variable name both in the current scope and the parent scope.

Usage:

```
DUAL_SCOPE_SET(<varName> [other args])
```

It turns out that when you call `ADD_SUBDIRECTORY(<someDir>)` or enter a FUNCTION that CMake actaully creates a copy of all of the regular non-cache varaibles in the current scope in order to create a new set of variables for the `CMakeLists.txt` file in `<someDir>`. This means that if you call `SET(SOMEVAR Blah PARENT_SCOPE)` that it will not affect the value of `SOMEVAR` in the current scope. This macro therefore is designed to set the value of the variable in the current scope and the parent scope in one shot to avoid confusion.

Global variables are different. When you move to a subordinate `CMakeLists.txt` file or enter a function, a local copy of the variable is *not* created. If you set the value name locally, it will shadow the global variable. However, if you set the globlal value with SET(SOMEVAR someValue CACHE INTERNAL “”), then the value will get changed in the current subordinate scope and in all parent scopes all in one shot!

GLOBAL_NULL_SET()

Set a variable as a null internal global (cache) variable (removes boiler plate).

Usage:

```
GLOBAL_NULL_SET(<varName>)
```

This just calls:

```
SET(<varName> "" CACHE INTERNAL "")
```

GLOBAL_SET()

Set a variable as an internal global (cache) variable (removes boiler plate).

Usage:

```
GLOBAL_SET(<varName> [other args])
```

This just calls:

```
SET(<varName> [other args] CACHE INTERNAL "")
```

JOIN()

Join a set of strings into a single string using a join string.

Usage:

```
JOIN(<outputStrVar> <sepStr> <quoteElements> "<string0>" "<string1"> ...)
```

Arguments:

`<outputStrVar>`

The name of a variable that will hold the output string.

`<sepStr>`

A string to use to join the list of strings.

`<quoteElements>`

If TRUE, then each `<stingi>` is quoted using an escaped quote char `\"`. If `FALSE` then no escaped quote is used.

On output the variable `<outputStrVar>` is set to:

```
"<string0><sepStr><string1><sepStr>..."
```

If `<quoteElements>=TRUE`, then it is set to:

```
"\"<string0>\"<sepStr>\"<string1>\"<sepStr>..."
```

For example, the latter can be used to set up a set of command-line arguments given a CMake array like:

```
JOIN(CMND_LINE_ARGS " " TRUE ${CMND_LINE_ARRAY})
```

WARNING: Be careful to quote string arguments that have spaces because CMake interpet those ase array boundaries.

MESSAGE_WRAPPER()

Function that wraps the standard CMake/CTest `MESSAGE()` function call in order to allow unit testing to intercept the call.

Usage:

```
MESSAGE_WRAPPER(<arg0> <arg1> ...)
```

This function takes exactly the same argumnets as built-in `MESSAGE()`. When the varible `MESSAGE_WRAPPER_UNIT_TEST_MODE` is set to `TRUE`, then this function will not call `MESSAGE(<arg0> <arg1> ...)` but instead will prepend set to global varible `MESSAGE_WRAPPER_INPUT` that input argumnets. To capture just this call's input, first call `GLOBAL_NULL_SET(MESSAGE_WRAPPER_INPUT(MESSAGE_WRAPPER_INPUT)` before calling this function.

This function allows one to unit test other user-defined CMake macros and functions that call this to catch error conditions wihtout stopping the CMake program. Otherwise, this is used to capture print messages to verify that they say the right thing.

MULTILINE_SET()

Function to set a single string by concatenating a list of separate strings

Usage:

```
MULTILINE_SET(<outputStrVar>
  "<string0>"
  "<string1>"
  ...
  )
```

On output, the local variables `<outputStrVar>` is set to:

```
"<string0><string1>..."
```

The purpose of this is to make it easier to set longer strings without going to far to the right.

PARSE_ARGUMENTS()

Parse a set of macro/functon input arguments into different lists. This allows the easy implementation of keyword-based user-defined macros and functions.

Usage:

```
PARSE_ARGUMENTS(
  <prefix>  <argNamesList>  <optionNamesList>
  <inputArgsList>
  )
```

Arguments to this macro:

`<prefix>`

> Prefix `<prefix>_` added the list and option variables created listed in `<argNamesList>` and `<optionNamesList>`.

`<argNamesList>`

Quoted array of list arguments (e.g. "<argName0>;<argName1>;..."). For each varaible name <argNamei>, a local varible will be created in the current scope with the name <prefix>_<argNamei> which gives the list of varibles parsed out of <inputArgsList>.

<optionNamesList>

Quoted array of list arguments (e.g. "<optName0>;<optName1>;..."). For each varaible name <optNamei>, a local varible will be created in the current scope with the name <prefix>_<optNamei> that is either set to TRUE or FALSE depending if <optNamei> apears in <inputArgsList> or not.

<inputArgsList>

List of arguments keyword-based arguments passed in for the outer macro or function to be parsed out into the different argument and option lists.

What this macro does is very simple yet very 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.

PREPEND_CMNDLINE_ARGS()

Utility function that prepends command-line arguments to a variable of command-line options.

Usage:

```
PREPEND_CMNDLINE_ARGS(<var> "<extraArgs>")
```

This function just prepends the command-line arguments in the string `"<extraArgs>"` but does not add an extra space if `<var>` is empty on input.

PREPEND_GLOBAL_SET()

Utility macro that prepends arguments to a global variable (reduces boiler-plate code and mistakes).

Usage:

```
PREPEND_GLOBAL_SET(<varName> <arg0> <arg1> ...)
```

NOTE: The variable `<varName>` must exist before calling this function. To set it empty initially use GLOBAL_NULL_SET().

APPEND_SET()

Utility function to append elements to a varible (reduces boiler-plate code).

Usage:

```
APPEND_SET(<varName> <arg0> <arg1> ...)
```

Just calls:

```
LIST(APPEND <varName> <arg0> <arg1> ...)
```

PRINT_NONEMPTY_VAR()

Print a defined variable giving its name then value only if it is not empty.

Usage:

```
PRINT_NONEMPTY_VAR(<varName>)
```

Calls `PRINT_VAR(<varName>)` if `${<varName>}` is not empty.

PRINT_VAR()

Unconditionally print a variable giving its name then value.

Usage:

```
PRINT_VAR(<varName>)
```

This prints:

```
MESSAGE("-- " "${VARIBLE_NAME}='${${VARIBLE_NAME}}'")
```

The variable `<varName>` can be defined or undefined or empty. This uses an explicit "-- " line prefix so that it prints nice even on Windows CMake.

REMOVE_GLOBAL_DUPLICATES()

Remove duplicate elements from a global list variable.

Usage:

```
REMOVE_GLOBAL_DUPLICATES(<globalVarName>)
```

This function is necessary in order to preserve the "global" nature of the variable. If you just call LIST(REMOVE_DUPLICATES ...) it will actually create a local variable of the same name and shadow the global variable! That is a fun bug to track down!

SET_AND_INC_DIRS()

Set a variable to an include dir and call `INCLUDE_DIRECTORIES()` (removes boiler plate).

Usage:

SET_AND_INC_DIRS(<dirVarName> <includeDir>)

On output, this justs `<dirVarName>` to `<includeDir>` in the local scope and calls `INCLUDE_DIRECTORIES(<includeDir>)`.

SET_CACHE_ON_OFF_EMPTY()

Usage:

```
SET_CACHE_ON_OFF_EMPTY(<varName> <initialVal> "<docString>" [FORCE])
```

Sets a special string cache variable with possible values "", "ON", or "OFF". This results in a nice dropdown box in the CMake cache manipulation GUIs.

SET_DEFAULT()

Give a local variable a default if a non-empty value is not already set.

Usage:

```
SET_DEFAULT(<varName> <arg0> <arg1> ...)
```

If on input `"${<varName>}"==""`, then `<varName>` is set to the given default. Otherwise, the existing non-empty value is preserved.

SET_DEFAULT_AND_FROM_ENV()

Set a default value for a local variable and override from an env var of the same name if it is set.

Usage:

```
SET_DEFAULT_AND_FROM_ENV(<varName> <defaultVal>)
```

First calls `SET_DEFAULT(<varName> <defaultVal>)` and then looks for an environment variable named `<varName>` and if non-empty, then overrides the value of `<varName>`.

This macro is primarily used in CTest code to provide a way to pass in the value of CMake variables. Older versions of `ctest` did not support the option `-D <var>:<type>=<value>` to allow varaibles to be set through the commandline like `cmake` always allowed.

SPLIT()

Split a string varible into a string array/list variable.

Usage:

```
SPLIT("<inputStr>" "<sepStr>" <outputStrListVar>)
```

The `<sepStr>` string is used with `STRING(REGEX ...)` to replace all occurrences of `<sepStr>` in `<inputStr>` with ";" and writing into `<outputStrListVar>`.

WARNING: `<sepStr>` is interpreted as a regular expression so keep that in mind when considering special regex chars like `'*'`, `'.'`, etc!

TIMER_GET_RAW_SECONDS()

Return the raw time in seconds since epoch, i.e., since 1970-01-01 00:00:00 UTC.

Usage:

```
TIMER_GET_RAW_SECONDS(<rawSecondsVar>)
```

This function is used along with TIMER_GET_REL_SECONDS(), and TIMER_PRINT_REL_TIME() to time big chunks of CMake code for timing and profiling purposes. See TIMER_PRINT_REL_TIME() for more details and an example.

NOTE: This function runs an external process to run the `date` command. Therefsore, it only works on Unix/Linux type systems that have a standard `date` command. Since this runs an external process, this function should only be used to time very course grained operations (i.e. that take longer than a second).

TIMER_GET_REL_SECONDS()

Return the relative time between start and stop seconds.

Usage:

```
TIMER_GET_REL_SECONDS(<startSeconds> <endSeconds> <relSecondsOutVar>)
```

This simple function computes the relative number of seconds between `<startSeconds>` and `<endSeconds>` (i.e. from TIMER_GET_RAW_SECONDS()) and sets the result in the local variable `<relSecondsOutVar>`.

TIMER_PRINT_REL_TIME()

Print the relative time between start and stop timers in `<min>m<sec>s` format.

Usage:

```
TIMER_PRINT_REL_TIME(<startSeconds> <endSeconds> "<messageStr>")
```

Differences the raw times `<startSeconds>` and `<endSeconds>` (i.e. gotten from TIMER_GET_RAW_SECONDS()) and prints the time in `<min>m<sec>s` format. This can only resolve times a second or greater apart. If the start and end times are less than a second then `0m0s` will be printed.

This is meant to be used with TIMER_GET_RAW_SECONDS() to time expensive blocks of CMake code like:

```
TIMER_GET_RAW_SECONDS(REAL_EXPENSIVE_START)

REAL_EXPENSIVE(...)

TIMER_GET_RAW_SECONDS(REAL_EXPENSIVE_END)

TIMER_PRINT_REL_TIME(${REAL_EXPENSIVE_START} ${REAL_EXPENSIVE_END}
   "REAL_EXPENSIVE() time")
```

This will print something like:

```
REAL_EXPENSIVE() time: 0m5s
```

Again, don't try to time something that takes less than 1 second as it will be recored as `0m0s`.

UNITTEST_COMPARE_CONST()

Perform a single unit test equality check and update overall test statistics

Usage:

```
UNITTEST_COMPARE_CONST(<varName> <expectedValue>)
```

If `${<varName>} == <expectedValue>`, then the check passes, otherwise it fails. This prints the variable name and values and shows the test result.

This updates the global variables `UNITTEST_OVERALL_NUMRUN`, `UNITTEST_OVERALL_NUMPASSED`, and `UNITTEST_OVERALL_PASS` which are used by the unit test harness system to assess overall pass/fail.

UNITTEST_STRING_REGEX()

Perform a series regexes of given strings and update overall test statistics.

Usage:

```
UNITTEST_STRING_REGEX(
  <inputString>
  REGEX_STRINGS <str0> <str1> ...
  )
```

If the `<inputString>` matches all of the of the regexs `<str0>`, "<str1>", ..., then the test passes. Otherwise it fails.

This updates the global variables `UNITTEST_OVERALL_NUMRUN`, `UNITTEST_OVERALL_NUMPASSED`, and `UNITTEST_OVERALL_PASS` which are used by the unit test harness system to assess overall pass/fail.

UNITTEST_FILE_REGEX()

Perform a series regexes of given strings and update overall test statistics.

Usage:

```
UNITTEST_FILE_REGEX(
  <inputFileName>
  REGEX_STRINGS <str1> <str2> ...
  )
```

The contents of `<inputFileName>` are read into a string and then passed to UNITTEST_STRING_REGEX() to assess pass/fail.

UNITTEST_FINAL_RESULT()

Print final statstics from all tests and assert final pass/fail

Usage:

```
UNITTEST_FINAL_RESULT(<expectedNumPassed>)
```

If `${UNITTEST_OVERALL_PASS}==TRUE` and `${UNITTEST_OVERALL_NUMPASSED} == <expectedNumPassed>`, then the overall test program is determined to have passed and string:

```
"Final UnitTests Result: PASSED"
```

is printed. Otherwise, the overall tets program is determined to have failed, the string:

```
"Final UnitTests Result: FAILED"
```

is printed and `MESSAGE(SEND_ERROR "FAIL")` is called.

The reason that we require passing in the expected number of passed tests is an an extra precaution to make sure that important unit tests are not left out. CMake is a loosely typed language and it pays to be a little paranoid.

14 References

15 Appendix

History of TriBITS

TriBITS started development in November 2007 as a set of helper macros to provide a CMake build system for a small subset of packages in Trilinos. The initial goal was to just to support a native Windows build (using Visual C++) to compile and install these few Trilinos packages on Windows for usage by another project (the Sandia Titan project which included VTK). At that time, Trilinos was using a highly customized autotools build system. Initially, this CMake system was just a set of macros to streamline creating executables and tests. Some of the conventions started in that early effort (e.g. naming conventions of variables and macros where functions use upper case like old FORTRAN and variables are mixed case) were continued in later efforts and are reflected in the current. Then, stating in early 2008, a more detailed evaluation was performed to see if Trilinos should stitch over to CMake as the default (and soon only) supported build and test system (see "Why CMake?" in TriBITS Overview). This lead to the initial implementation of a scale-able package-based architecture (PackageArch) for the Trilinos CMake project in late 2008. This Trilinos CMake PackageArch system evolved over the next few years with development in the system slowing into 2010. This Trilinos CMake build system was then adopted as the build infrastructure for the CASL VERA effort in 2011 where CASL VERA packages were treated as add-on Trilinos packages (see Section Multi-Repository Support). Over the next year, there was significant development of the system to support larger multi-repo projects in support of CASL VERA. That lead to the decision to formally generalize the Trilinos CMake PackageArch build system outside of Trilinos and the name TriBITS was formally adopted in November 2011. Work to refactor the Trilinos CMake system into a general reusable stand-alone CMake-based build system started in October 2011 and an initial implementation was complete in December 2011 when it was used for the CASL VERA build system. In early 2012, the ORNL CASL-related projects Denovo and SCALE ([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/