TriBITS Developers Guide and Reference

Author: Roscoe A. Bartlett (bartlettra@ornl.gov)
Date: 2015-09-22
Version: tribits_start-1327-gca8bb07

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

Contents

1 Introduction **1**

2 Background **2**

2.1 TriBITS Developer and User Roles . . . 2

2.2 CMake Language Overview and Gotchas . . . 3

2.3 Software Engineering Packaging Principles . . . 6

3 TriBITS Project Structure **6**

3.1 TriBITS Structural Units . . . 6

TriBITS Project . . . 8

TriBITS Project Core Files . . . 8

TriBITS Project Core Variables . . . 14

TriBITS Repository . . . 16

TriBITS Repository Core Files . . . 16

TriBITS Repository Core Variables . . . 20

TriBITS Package . . . 21

TriBITS Package Core Files . . . 22

TriBITS Package Core Variables . . . 25

TriBITS Subpackage . . . 27

TriBITS Subpackage Core Files . . . 28

TriBITS Subpackage Core Variables . . . 30

How is a TriBITS Subpackage is different from a TriBITS Package? . . . 30

TriBITS TPL . . . 30

3.2 Processing of TriBITS Files: Ordering and Details . . . 32

Full TriBITS Project Configuration . . . 32

Reduced Package Dependency Processing . . . 33

File Processing Tracing . . . 34
3.3 Coexisting Projects, Repositories, and Packages . . . 35
3.4 Standard and Common TPLs . . . 36
Standard TriBITS TPLs . . . 36
Common TriBITS TPLs . . . 36

4 Example TriBITS Projects **37**
4.1 TribitsHelloWorld . . . 37
4.2 TribitsExampleProject . . . 38
4.3 MockTrilinos . . . 41
4.4 ReducedMockTrilinos . . . 42
4.5 Trilinos . . . 43
4.6 The TriBITS Test Package . . . 43

5 Package Dependencies and Enable/Disable Logic **43**
5.1 Example ReducedMockTrilinos Project Dependency Structure . . . 44
5.2 TriBITS Dependency Handling Behaviors . . . 47
5.3 Example Enable/Disable Use Cases . . . 52
5.4 <Project>PackageDependencies.xml . . . 61

6 TriBITS Automated Testing **61**
6.1 Test Classifications for Repositories, Packages, and Tests . . . 62
6.2 Nested Layers of TriBITS Project Testing . . . 65
6.3 Pre-push Testing using checkin-test.py . . . 67
6.4 TriBITS Package-by-Package CTest/Dash Driver . . . 68
CTest/CDash Nightly Testing . . . 69
CTest/CDash CI Server . . . 69
6.5 TriBITS CDash Customizations . . . 69
CDash regression email addresses . . . 69

7 Multi-Repository Support **71**

8 Development Workflows **73**
8.1 Basic Development Workflow . . . 74
8.2 Multi-Repository Development Workflow . . . 74

9 Howtos **74**
9.1 How to add a new TriBITS Package . . . 74
9.2 How to add a new TriBITS Package with Subpackages . . . 74
9.3 How to add a new TriBITS Subpackage . . . 75
9.4 How to add a new TriBITS TPL . . . 75
9.5 How to use FIND_PACKAGE() for a TriBITS TPL . . . 76
9.6 How to add a new TriBITS Repository . . . 78
9.7 How to add a new TriBITS SE Package dependency . . . 78
9.8 How to add a new TriBITS TPL dependency . . . 79
9.9 How to insert a package into an upstream repo . . . 80

10 Additional Topics **81**
10.1 TriBITS Repository Contents . . . 82
TriBITS/ Directory Contents . . . 82
TriBITS/tribits/ Directory Contents . . . 82
10.2 TriBITS System Project Dependencies . . . 83
10.3 Python Support . . . 84
10.4 Project-Specific Build Reference . . . 84
10.5 Project and Repository Versioning and Release Mode . . . 85
10.6 TriBITS Environment Probing and Setup . . . 85
10.7 Configure-time System Tests . . . 86
10.8 Creating Source Distributions . . . 86
10.9 Multi-Repository Almost Continuous Integration . . . 87
ACI Introduction . . . 87
ACI Multi-Git/TriBITS Repo Integration Example . . . 88
ACI Local Sync Git Repo Setup . . . 88
ACI Integration Build Directory Setup . . . 89
ACI Sync Driver Script . . . 90
ACI Cron Job Setup . . . 92
Addressing ACI Failures and Summary . . . 92
10.10 Post-Push CI and Nightly Testing using checkin-test.py . . . 92
10.11 TriBITS Dashboard Driver . . . 93
10.12 Regulated Backward Compatibility and Deprecated Code . . . 93
Setting up support for deprecated code handling . . . 94
Deprecating code but maintaining backward compatibility . . . 94
Deprecating C/C++ classes, structs, functions, typdefs, etc. . . . 95
Deprecating preprocessor macros . . . 95
Deprecating an entire header and/or source file . . . 96
Hiding deprecated code to certify and facilitate later removal . . . 96
Hiding C/C++ entities . . . 96
Hiding entire deprecated header and source files . . . 96
Physically removing deprecated code . . . 97
Removing entire deprecated header and source files . . . 97
Removing deprecated code from remaining files . . . 98
10.13 Installation and Backward Compatibility Testing . . . 98
10.14 Wrapping Externally Configured/Built Software . . . 98
10.15 TriBITS directory snapshotting . . . 99
10.16 TriBITS Development Toolset . . . 100

11 References **100**

12 TriBITS Detailed Reference Documentation **100**
12.1 TriBITS Global Project Settings . . . 100
12.2 TriBITS Macros and Functions . . . 107
TRIBITS_ADD_ADVANCED_TEST() . . . 107
TRIBITS_ADD_DEBUG_OPTION() . . . 113
TRIBITS_ADD_EXAMPLE_DIRECTORIES() . . . 113
TRIBITS_ADD_EXECUTABLE() . . . 114
TRIBITS_ADD_EXECUTABLE_AND_TEST() . . . 117
TRIBITS_ADD_LIBRARY() . . . 118
TRIBITS_ADD_OPTION_AND_DEFINE() . . . 120
TRIBITS_ADD_SHOW_DEPRECATED_WARNINGS_OPTION() . . . 121
TRIBITS_ADD_TEST() . . . 121
TRIBITS_ADD_TEST_DIRECTORIES() . . . 128
TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES() . . . 128
TRIBITS_CONFIGURE_FILE() . . . 129
TRIBITS_COPY_FILES_TO_BINARY_DIR() . . . 130
TRIBITS_CTEST_DRIVER() . . . 132
TRIBITS_DETERMINE_IF_CURRENT_PACKAGE_NEEDS_REBUILT() . . . 133
TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS() . . . 133
TRIBITS_EXCLUDE_FILES() . . . 134
TRIBITS_FIND_MOST_RECENT_BINARY_FILE_TIMESTAMP() . . . 134
TRIBITS_FIND_MOST_RECENT_FILE_TIMESTAMP() . . . 134
TRIBITS_FIND_MOST_RECENT_SOURCE_FILE_TIMESTAMP() . . . 135
TRIBITS_INSTALL_HEADERS() . . . 136
TRIBITS_INCLUDE_DIRECTORIES() . . . 136
TRIBITS_PACKAGE() . . . 136
TRIBITS_PACKAGE_DECL() . . . 137
TRIBITS_PACKAGE_DEF() . . . 138
TRIBITS_PACKAGE_DEFINE_DEPENDENCIES() . . . 138
TRIBITS_PACKAGE_POSTPROCESS() . . . 141
TRIBITS_PROCESS_SUBPACKAGES() . . . 141
TRIBITS_PROCESS_ENABLED_TPL() . . . 141
TRIBITS_PROJECT() . . . 141
TRIBITS_PROJECT_DEFINE_EXTRA_REPOSITORIES() . . . 141
TRIBITS_PROJECT_ENABLE_ALL() . . . 143
TRIBITS_REPOSITORY_DEFINE_PACKAGES() . . . 143
TRIBITS_REPOSITORY_DEFINE_TPLS() . . . 144
TRIBITS_SET_ST_FOR_DEV_MODE() . . . 145
TRIBITS_SUBPACKAGE() . . . 145
TRIBITS_SUBPACKAGE_POSTPROCESS() . . . 145
TRIBITS_TPL_ALLOW_PRE_FIND_PACKAGE() . . . 146
TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES() . . . 146
TRIBITS_TPL_TENTATIVELY_ENABLE() . . . 148

TRIBITS_VERBOSE_PRINT_VAR() . . . 148
TRIBITS_WRITE_FLEXIBLE_PACKAGE_CLIENT_EXPORT_FILES() . . . 148
12.3 General Utility Macros and Functions . . . 149
ADD_SUBDIRECTORIES() . . . 149
ADVANCED_OPTION() . . . 150
ADVANCED_SET() . . . 150
APPEND_CMNDLINE_ARGS() . . . 150
APPEND_GLOB() . . . 150
APPEND_GLOBAL_SET() . . . 150
APPEND_SET() . . . 151
APPEND_STRING_VAR() . . . 151
APPEND_STRING_VAR_EXT() . . . 151
APPEND_STRING_VAR_WITH_SEP() . . . 151
ASSERT_DEFINED() . . . 151
COMBINED_OPTION() . . . 152
CONCAT_STRINGS() . . . 152
DUAL_SCOPE_APPEND_CMNDLINE_ARGS() . . . 152
DUAL_SCOPE_PREPEND_CMNDLINE_ARGS() . . . 152
DUAL_SCOPE_SET() . . . 153
GLOBAL_NULL_SET() . . . 153
GLOBAL_SET() . . . 153
JOIN() . . . 153
MESSAGE_WRAPPER() . . . 154
MULTILINE_SET() . . . 154
PARSE_ARGUMENTS() . . . 155
PREPEND_CMNDLINE_ARGS() . . . 157
PREPEND_GLOBAL_SET() . . . 157
PRINT_NONEMPTY_VAR() . . . 157
PRINT_NONEMPTY_VAR_WITH_SPACES() . . . 157
PRINT_VAR() . . . 157
REMOVE_GLOBAL_DUPLICATES() . . . 158
SET_AND_INC_DIRS() . . . 158
SET_CACHE_ON_OFF_EMPTY() . . . 158
SET_DEFAULT() . . . 158
SET_DEFAULT_AND_FROM_ENV() . . . 158
SPLIT() . . . 159
TIMER_GET_RAW_SECONDS() . . . 159
TIMER_GET_REL_SECONDS() . . . 159
TIMER_PRINT_REL_TIME() . . . 159
UNITTEST_COMPARE_CONST() . . . 160
UNITTEST_HAS_SUBSTR_CONST() . . . 160
UNITTEST_STRING_REGEX() . . . 160
UNITTEST_FILE_REGEX() . . . 160
UNITTEST_FINAL_RESULT() . . . 161

13 FAQ **161**

14 Appendix **161**

14.1 History of TriBITS . . . 161

14.2 Why a TriBITS Package is not a CMake Package . . . 162

14.3 Design Considerations for TriBITS . . . 162

14.4 clone_extra_repos.py --help . . . 162

14.5 gitdist --help . . . 164

14.6 snapshot-dir.py --help . . . 168

14.7 checkin-test.py --help . . . 170

14.8 install_devtools.py --help . . . 185

1 Introduction

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

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

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

2 Background

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

2.1 TriBITS Developer and User Roles

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

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

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

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

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

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

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

2.2 CMake Language Overview and Gotchas

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In this case, the misspelled variable would be caught.

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

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

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

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

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

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

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

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

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

2.3 Software Engineering Packaging Principles

The term "software engineering" package is adopted in TriBITS nomenclature in order to highlight and the role of defining and managing "Packages" according to standard software engineering packaging principles. In his book [Agile Software Development, 2003], Robert Martin defines several object-oriented (OO) software engineering principles related to packaging software which are listed below:

- *Package Cohesion OO Principles*:

1) *REP (Release-Reuse Equivalency Principle)*: The granule of reuse is the granule of release.
2) *CCP (Common Closure Principle)*: The classes in a package should be closed together against the same kinds of changes. A change that affects a closed package affects all the classes in that package and no other packages.
3) *CRP (Common Reuse Principle)*: The classes in a package are used together. If you reuse one of the classes in a package, you reuse them all.

- *Package Coupling OO Principles*:

4) *ADP (Acyclic Dependencies Principle)*: Allow no cycles in the package dependency graph.
5) *SDP (Stable Dependencies Principle)*: Depend in the direction of stability.
6) SAP (Stable Abstractions Principle): A package should be as abstract as it is stable.

Any of these six OO packaging principles (and other issues) may be considered when deciding how to partition software into different TriBITS SE Packages.

NOTE: This purpose of this TriBITS Developers Guide document is not teach basic software engineering so these various principles will not be expanded on further here. However, interested readers are strongly encouraged to read [Agile Software Development, 2003] as one of the better software engineering books out there (see http://web.ornl.gov/~8vt/readingList.html#Most_Recommended_SE_Books).

3 TriBITS Project Structure

TriBITS is a Framework, implemented in CMake, to create CMake projects. As a Framework (with a capital "F"), TriBITS defines the overall structure of a CMake build system for a project and it processes the various project-, repository-, and package-specific files in a specified order. Almost all of this processing takes place in the TRIBITS_PROJECT() macro (or macros and functions it calls). The following subsections define the essence of the TriBITS framework in some detail. Later sections cover specific topics and the various sections link to each other. Within this section, the subsection TriBITS Structural Units defines the basic units TriBITS Project, TriBITS Repository, TriBITS Package, TriBITS TPL and other related structural units. The subsection Processing of TriBITS Files: Ordering and Details defines exactly what files TriBITS processes and in what order. It also shows how to get TriBITS to show exactly what files it is processing to help in debugging issues. The subsection Coexisting Projects, Repositories, and Packages gives some of the rules and constrains for how the different structure units can co-exist in the same directories. The last two subsections in this section cover Standard TriBITS TPLs and Common TriBITS TPLs.

3.1 TriBITS Structural Units

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

First, to better 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 associated tests to help define and protect the functionality provided by the software. A package also typically defines a single integrated unit of documentation and testing (see TriBITS Automated Testing). A TriBITS package may or may not be broken down into multiple subpackages. Examples of TriBITS packages in TribitsExampleProject include `SimpleCXX`, `MixedLang` and `WithSubpackages`. (Don't confuse a TriBITS "Package" with a raw CMake "Package". A raw CMake "Package" actually maps to a TriBITS TPL; see Why a TriBITS Package is not a CMake Package.)

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

- **TriBITS SE Package**: The combined set of TriBITS Packages and TriBITS Subpackages that constitute the basic *Software Engineering* packages of a TriBITS project (see Software Engineering Packaging Principles): SE packages are the basis for setting dependencies in the TriBITS system. For example, the SE Packages provided by the top-level example package `WithSubpackages` (in order of increasing dependencies) are `WithSubpackagesA`, `WithSubpackagesB`, `WithSubpackagesC`, and `WithSubpackages` (see TribitsExampleProject).

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

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

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

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

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

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

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

The following subsections define the major structural units of a TriBITS project in more detail. Each structural unit is described along with the files and directories associated 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 example projects are a good way to start. These example projects will be referenced in the more detailed descriptions given in this document.

The last issue to touch on before getting into the detailed descriptions of the different TriBITS structural units is the issue of how CMake variables are defined and used by TriBITS. The CMake variables described in the TriBITS structural units below fall into one of two major 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 non-namespaced 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 without having to refer to any specific names.

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

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

TriBITS Project

A TriBITS Project:

- Defines the `PROJECT_NAME` CMake variable (defined in <projectDir>/ProjectName.cmake)
- Defines a complete CMake project which calls `PROJECT(${PROJECT_NAME} ...)` and can be directly configured, built, tested, installed, etc.
- Consists of one or more TriBITS Repositories (and may itself be a TriBITS Repository) which can include native and extra repositories.
- Allows for extra Repositories to be added on before or after the set of native Repositories (specified in <projectDir>/cmake/ExtraRepositoriesList.cmake or by CMake cache variables)
- Defines a default CDash server and default project name on the server (the project name on the CDash server must be the same as `${PROJECT_NAME}`).
- Defines pre-push testing standard builds (using checkin-test.py)

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

- TriBITS Project Core Files
- TriBITS Project Core Variables

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

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

These TriBITS Project files are documented in more detail below:

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

- <projectDir>/cmake/ctest/CTestCustom.cmake.in

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

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

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

A meta-project's `ProjectName.cmake` file might have a number of other variables set that define how the various different TriBITS repos are cobbled together into a single TriBITS meta-project (usually because the different TriBITS repos and packages are a bit messy and have other issues). For example, the CASL VERA TriBITS meta-project at one point had a very extensive collection of set statements in this file.

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

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

# To be safe, define your minimum CMake version.  This may be newer than the
# min required by TriBITS.
CMAKE_MINIMUM_REQUIRED(VERSION 2.8.11 FATAL_ERROR)

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

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

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

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

#
```

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

SET(${PROJECT_NAME}_TRIBITS_DIR
   "${CMAKE_CURRENT_LIST_DIR}/../.."  CACHE  STRING
  "TriBITS base directory (default assumes in TriBITS source tree)")
INCLUDE("${${PROJECT_NAME}_TRIBITS_DIR}/TriBITS.cmake")

# Set default location for header-only TPL to make easy to configure out of
# the TriBITS source tree.
SET(HeaderOnlyTpl_INCLUDE_DIRS
  "${${PROJECT_NAME}_TRIBITS_DIR}/examples/tpls/HeaderOnlyTpl"
  CACHE PATH "Default set by TriBITS/CMakeLists.txt" )

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

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

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

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

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

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

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

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

The minimum CMake version must also be declared in the top-level `CMakeLists.txt` file as shown. Explicitly setting the minimum CMake version avoids strange errors that can occur when someone tries to build the project using a version of CMake that is too old. The project should set the minimum CMake version based on the CMake features used in that project's own CMake files. The minimum CMake version required by TriBITS is defined in in the variable `TRIBITS_CMAKE_MINIMUM_REQUIRED` (the current minimum version of CMake required by TriBITS is given at in Getting set up to use CMake) . For example, the `VERA/CMakeLists.txt` file lists as its first line:

```
SET(VERA_TRIBITS_CMAKE_MINIMUM_REQUIRED 2.8.11)
CMAKE_MINIMUM_REQUIRED(VERSION ${VERA_TRIBITS_CMAKE_MINIMUM_REQUIRED}
  FATAL_ERROR)
```

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

```
INCLUDE(SetDefaultAndFromEnv)

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

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

IF (CTEST_DROP_METHOD STREQUAL "http")
  SET_DEFAULT_AND_FROM_ENV(CTEST_DROP_SITE "casl-dev.ornl.gov")
  SET_DEFAULT_AND_FROM_ENV(CTEST_PROJECT_NAME "TribitsExProj")
  SET_DEFAULT_AND_FROM_ENV(CTEST_DROP_LOCATION "/cdash/submit.php?project=TribitsExPro
  SET_DEFAULT_AND_FROM_ENV(CTEST_TRIGGER_SITE "")
  SET_DEFAULT_AND_FROM_ENV(CTEST_DROP_SITE_CDASH TRUE)
ENDIF()
```

All of the variables set in this file are directly understood by raw `ctest` and will not be explained here further (see documentation for the standard CMake module `CTest`). The usage of the function SET_DEFAULT_AND_FROM_ENV() allows the variables to be overridden both as CMake cache variables and in the environment. The latter is needed when running using `ctest` as the driver (since older versions of `ctest` did not support `-D<var-name>:<type>=<value>` command-line arguments like `cmake` does). Given that all of these variables are nicely namespaced, overriding them in the shell environment is not as dangerous as might otherwise be the case but this is what had to be done to get around limitations for older versions of CMake/CTest.

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

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

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

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

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

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

configuration = {

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

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

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

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

        }, # cmake

    } # configuration
```

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

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

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

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

There are no examples for the usage of this file in any of the TriBITS examples or test projects. However, support for this file is maintained for backward compatibility since there may be some TriBITS projects that still use it. It is recommended instead to define multiple repositories using the <projectDir>/cmake/ExtraRepositoriesList.cmake file as it allows for more flexibility in how extra repositories are specified and how they are accessed. The latter file allows the various tools to perform VC activities with these repos while "native repositories" do not.

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

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

```
TRIBITS_PROJECT_DEFINE_EXTRA_REPOSITORIES(
  ExtraRepo1  ""  GIT  someurl.com:/ExtraRepo1   ""          Continuous
  ExtraRepo2  packages/SomePackage/Blah  GIT  someurl2.com:/ExtraRepo2
```

```
                                                    NOPACKAGES  Nightly
  ExtraRepo3  ""  HG  someurl3.com:/ExtraRepo3   ""          Continuous
  ExtraRepo4  ""  SVN  someurl4.com:/ExtraRepo4  ""          Nightly
  )
```

shows the specification of both TriBITS Repositories and non-TriBITS VC Repositories. In the above file, the repositories `ExtraRepo1`, `ExtraRepo3`, and `ExtraRepo4` are both TriBITS and VC repositories that are cloned into directories under `<projectDir>` of the same names from the URLs `someurl.com:/ExtraRepo1`, `someurl3.com:/ExtraRepo3`, and `someurl4.com:/ExtraRepo4`, respectively. However, the repository `ExtraRepo2` is **not** a TriBITS Repository because it is marked as `NOPACKAGES`. In this case, it gets cloned as the directory:

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

However, the code in the tools checkin-test.py and TRIBITS_CTEST_DRIVER() will consider non-TriBITS VC repos like `ExtraRepo2` and any changes to this repository will be listed as changes to `somePackage` (see Pre-push Testing using checkin-test.py).

NOTE: This file can be overridden by setting the cache variable <Project>_EXTRAREPOS_FILE.

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

The typical usage of this file is to set the default CDash email address for all packages or override the email addresses for all of a repository's package CDash regression email addresses (see CDash regression email addresses). For example, to set the default email address for all of the packages, one would set in this file:

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

The repository email address variables ${REPOSITORY_NAME}_REPOSITORY_EMAIL_URL_ADDRESS_BASE and ${REPOSITORY_NAME}_REPOSITORY_MASTER_EMAIL_ADDRESS possibly set in the just processed <repoDir>/cmake/RepositoryDependenciesSetup.cmake files can also be overridden in this file. The CASL VERA meta-project uses this file to override several of the repository-specific email addresses for its constituent repositories.

In general, variables that affect how package dependencies are defined or affect package and TPL enable/disable logic for *only this particular project* should be defined in this file.

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

TribitsExampleProject/`cmake/CallbackDefineProjectPackaging.cmake`

provides a good example which is:

```
MACRO(TRIBITS_PROJECT_DEFINE_PACKAGING)

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

    SET(CPACK_PACKAGE_DESCRIPTION "TribitsExampleProject just shows you how to use Tri
    SET(CPACK_PACKAGE_FILE_NAME "tribitsexproj-setup-${TribitsExProj_VERSION}")
    SET(CPACK_PACKAGE_INSTALL_DIRECTORY "TribitsExProj ${TribitsExProj_VERSION}")
```

```
    SET(CPACK_PACKAGE_REGISTRY_KEY "TribitsExProj ${TribitsExProj_VERSION}")
    SET(CPACK_PACKAGE_NAME "tribitsexproj")
    SET(CPACK_PACKAGE_VENDOR "Sandia National Laboratories")
    SET(CPACK_PACKAGE_VERSION "${TribitsExProj_VERSION}")
    SET(CPACK_RESOURCE_FILE_README "${TribitsExProj_README}")
    SET(CPACK_RESOURCE_FILE_LICENSE "${TribitsExProj_LICENSE}")
    SET(${PROJECT_NAME}_CPACK_SOURCE_GENERATOR_DEFAULT "TGZ;TBZ2")
    SET(CPACK_SOURCE_FILE_NAME "tribitsexproj-source-${TribitsExProj_VERSION}")
    SET(CPACK_COMPONENTS_ALL ${TribitsExProj_PACKAGES} Unspecified)

ENDMACRO()
```

The CPack variables show above that should be defined at the project-level are described in the Official CPack Documentation.

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

<projectDir>/cmake/tribits/: [Optional] This is the typical location of the TriBITS/tribits/ source tree for projects that choose to snapshot TriBITS into their source tree. In fact, TriBITS assumes this is the default location for the TriBITS source tree if `${PROJECT_NAME}_TRIBITS_DIR` is not otherwise specified. Trilinos, for example, currently snapshots the TriBITS source tree into this directory. See TriBITS directory snapshotting for more details.

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

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

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

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

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

For documentation of the options one can change for CTest, see online CTest documentation.

TriBITS Project Core Variables The following *local variables* are defined in the top-level Project `CMakeLists.txt` file scope and are therefore accessible by all files processed by TriBITS:

`PROJECT_NAME`

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

`PROJECT_SOURCE_DIR`

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

`PROJECT_BINARY_DIR`

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

`${PROJECT_NAME}_SOURCE_DIR`

Set to the same directory as `${PROJECT_SOURCE_DIR}` automatically by the built-in `PROJECT()` command called in the top-level <projectDir>/CMakeLists.txt file..

`${PROJECT_NAME}_BINARY_DIR`

Set to the same directory as `${PROJECT_BINARY_DIR}` automatically by the built-in `PROJECT()` command called in the top-level <projectDir>/CMakeLists.txt file..

The following *cache varaibles* are defined for every TriBITS project:

`${PROJECT_NAME}_TRIBITS_DIR`

CMake cache variable that gives the path to the TriBITS implementation directory. When set to a relative path (set as type `STRING`, see below), this is taken relative to `${CMAKE_CURRENT_SOURCE_DIR}/` (the project base source dir). When an absolute path is given, it is used without modification. If this variable is not set in the <projectDir>/CMakeLists.txt file, then it will be automatically set as a `PATH` cache variable by the include of `TriBITS.cmake` by the statement

```
SET( ${PROJECT_NAME}_TRIBITS_DIR
  "${CMAKE_CURRENT_SOURCE_DIR}/cmake/tribits" CACHE PATH "...")
```

Therefore, projects that snapshot TriBITS into <projectDir>/cmake/tribits/ don't need to explicitly set `${PROJECT_NAME}_TRIBITS_DIR`. In addition, one can also point to a different TriBITS implementation just by setting the absolute path:

```
-D <Project>_TRIBITS_DIR=<some-abs-dir>
```

or to a relative path using, for example:

```
-D <Project>_TRIBITS_DIR:STRING=TriBITS/tribits
```

Note that when the `TriBITS` git repo itself is cloned by a TriBITS project, then `${PROJECT_NAME}_TRIBITS_DIR` should be set to the directory `TriBITS/tribits` (see TriBITS/tribits/) as shown above.

`${PROJECT_NAME}_ENABLE_TESTS`

CMake cache variable that if set to `ON`, then tests for all explicitly enabled packages will be turned on. This has a default value of `OFF`. This is used in logic to enable individual SE package tests (see <Project>_ENABLE_TESTS only enables explicitly enabled SE package tests).

`${PACKAGE_NAME}_ENABLE_EXAMPLES`

CMake cache variable that if set to `ON`, then examples for all explicitly enabled packages will be turned on. This has a default value of `OFF`.

The following *internal project-scope local* (non-cache) CMake variables are defined by TriBITS giving the project's TriBITS repositories.:

`${PROJECT_NAME}_NATIVE_REPOSITORIES`

The list of Native Repositories for a given TriBITS project (i.e. Repositories that are always present when configuring the Project and are managed in the same VC repo typically). This variable is set in the file <projectDir>/cmake/NativeRepositoriesList.cmake if it exists. 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 minimum a `PackagesList.cmake` file, and a `TPLsList.cmake` file (see TriBITS Repository).

`${PROJECT_NAME}_EXTRA_REPOSITORIES`

The list of Extra Repositories that the project is being configured with. This list of repositories either comes from processing the project's <projectDir>/cmake/ExtraRepositoriesList.cmake file or comes from the CMake cache variable `${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 they are processed.

TriBITS Repository

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

In short, a TriBITS Repository:

- Is a named collection of related TriBITS Packages and TPLs (defined in <repoDir>/PackagesList.cmake and <repoDir>/TPLsList.cmake respectively)
- Defines the base source and binary directories for the Repository `${REPOSITORY_NAME}_SOURCE_DIR` and `${REPOSITORY_NAME}_BINARY_DIR`.
- Defines a common set of initializations and other hooks for a all the packages in the repository.
- Typically maps to a VC (i.e. git) repository and therefore represents a unit of integration, versioning and reuse. (But core TriBITS has no dependency on any VC tool.)

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

- TriBITS Repository Core Files
- TriBITS Repository Core Variables

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

```
<repoDir>/
  PackagesList.cmake
  TPLsList.cmake
  Copyright.txt  # [Optional] Only needed if creating version header file
  Version.cmake  # [Optional] Info inserted into ${REPO_NAME}_version.h
  cmake/
     RepositoryDependenciesSetup.cmake # [Optional] CDash email addresses?
     CallbackSetupExtraOptions.cmake # [Optional] Called after main options
     CallbackDefineRepositoryPackaging.cmake # [Optional] CPack packaging
```

These TriBITS Repository files are documented in more detail below:

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

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

```
TRIBITS_REPOSITORY_DEFINE_PACKAGES(
  SimpleCxx          packages/simple_cxx         PT
  MixedLang          packages/mixed_lang         PT
  InsertedPkg        InsertedPkg                 ST
  WithSubpackages    packages/with_subpackages   PT
  WrapExternal       packages/wrap_external      ST
  )

TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS(WrapExternal Windows)
TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES(InsertedPkg)
```

Other commands that are appropriate to use in this file include TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS() and TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES(). Also, if the binary directory for any package `<packageName>` needs to be changed from the default, then the variable `<packageName>_SPECIFIED_BINARY_DIR` can be set. (see TriBITS Package == TriBITS Repository == TriBITS Project).

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

```
TRIBITS_REPOSITORY_DEFINE_PACKAGES()
```

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

<repoDir>/TPLsList.cmake: [Required] Provides the list of TPLs that are referenced as TPL dependencies in the repository's SE package's <packageDir>/cmake/Dependencies.cmake files (see TriBITS TPL). This file typically just calls the macro TRIBITS_REPOSITORY_DEFINE_TPLS() to define the TPLs along with their find modules and other properties. An example is ReducedMockTrilinos/`TPLsList.cmake` which shows:

```
TRIBITS_REPOSITORY_DEFINE_TPLS(
  MPI            "${${PROJECT_NAME}_TRIBITS_DIR}/core/std_tpls/"    PT
  BLAS           "${${PROJECT_NAME}_TRIBITS_DIR}/core/std_tpls/"    PT
  LAPACK         "${${PROJECT_NAME}_TRIBITS_DIR}/core/std_tpls/"    PT
  Boost          "${${PROJECT_NAME}_TRIBITS_DIR}/core/std_tpls/"    ST
  UMFPACK        cmake/TPLs/    ST
  AMD            cmake/TPLs/    EX
  PETSC          "${${PROJECT_NAME}_TRIBITS_DIR}/common_tpls/"    ST
  )
```

See TriBITS TPL for details on what gets defined for each TriBITS TPL once this file is processed.

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

```
TRIBITS_REPOSITORY_DEFINE_TPLS()
```

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

<repoDir>/Copyright.txt: [Optional] Gives the default copyright and license declaration for all of the software in the TriBITS repository directory `<repoDir>/`. This file is read into a string and then used to configure the repository's version header file (see Project and Repository Versioning and Release Mode). Even if a repository version header file is not produced, it is a good idea for every TriBITS repository to define this file, just for legal purposes. For a good open-source license, one should consider copying the `TriBITS/Copyright.txt` file which is a simple 3-clause BSD-like license like:

```
# @HEADER
# ************************************************************************
#
#            TriBITS: Tribal Build, Integrate, and Test System
#                    Copyright 2013 Sandia Corporation
#
# Under the terms of Contract DE-AC04-94AL85000 with Sandia Corporation,
# the U.S. Government retains certain rights in this software.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions are
# met:
#
# 1. Redistributions of source code must retain the above copyright
# notice, this list of conditions and the following disclaimer.
#
# 2. Redistributions in binary form must reproduce the above copyright
# notice, this list of conditions and the following disclaimer in the
# documentation and/or other materials provided with the distribution.
#
# 3. Neither the name of the Corporation nor the names of the
# contributors may be used to endorse or promote products derived from
# this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY SANDIA CORPORATION "AS IS" AND ANY
# EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
# IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
# PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL SANDIA CORPORATION OR THE
# CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
# EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
# PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
# LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
# NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
# SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#
# ************************************************************************
# @HEADER
```

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

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

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

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

<repoDir>/cmake/RepositoryDependenciesSetup.cmake: [Optional] If present, this file is included a single time as part of the generation of the project dependency data-structure (see Reduced Package Dependency Processing). It gets included in the order listed in ${PROJECT_NAME}_ALL_REPOSITORIES. Any local variables set in this file have project-wide scope. The primary purpose for this file is to set variables that will impact the processing of project's

package's `Dependencies.cmake` files and take care of other enable/disable issues that are not otherwise cleanly handled by the TriBITS system automatically.

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

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

Note that the prefix `${REPOSITORY_NAME}_` is used instead of hard-coding the repo name to allow greater flexibility in how meta-projects refer to this TriBITS repo.

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

A few additional variables are defined by the time this file is processed and can be used in the logic in these files. Some of the variables that should already be defined (in addition to all of the basic user TriBITS cache variables set in `TRIBITS_DEFINE_GLOBAL_OPTIONS_AND_DEFINE_EXTRA_REPOS()`) include `CMAKE_HOST_SYSTEM_NAME`, `${PROJECT_NAME}_HOSTNAME`, and `PYTHON_EXECUTABLE` (see Python Support). The types of commands and logic to put in this file include:

- Setting additional user cache variable options that are used by multiple packages in the TriBITS Repository. For example, Trilinos defines a `Trilinos_DATA_DIR` user cache variable that several Trilinos packages use to get extra test data to define extra tests.
- Disabling packages in the TriBITS Repository when conditions will not allow them to be enabled. For example, Trilinos disables the package `ForTrilinos` when `Fortran` is disabled and disables the package `PyTrilinos` when Python support is disabled.

An example of this file is:

TribitsExampleProject//`cmake/CallbackSetupExtraOptions.cmake`

which currently looks like:

```
MACRO(TRIBITS_REPOSITORY_SETUP_EXTRA_OPTIONS)

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

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

  IF ("${PYTHON_EXECUTABLE}" STREQUAL "")
    MESSAGE(
      "\n***"
      "\n*** Warning: Setting ${PROJECT_NAME}_ENABLE_WrapExternal=OFF"
      " because PYTHON_EXECUTABLE=''!"
      "\n***\n"
```

```
      )
    SET(${PROJECT_NAME}_ENABLE_WrapExternal OFF)
  ENDIF()

  ASSERT_DEFINED(${PROJECT_NAME}_ENABLE_Fortran)

  IF (NOT ${PROJECT_NAME}_ENABLE_Fortran)
    MESSAGE(
      "\n***"
      "\n*** Warning: Setting ${PROJECT_NAME}_ENABLE_MixedLang=OFF"
      " because ${PROJECT_NAME}_ENABLE_Fortran='${${PROJECT_NAME}_ENABLE_Fortran}'!"
      "\n***\n"
      )
    SET(${PROJECT_NAME}_ENABLE_MixedLang OFF)
  ENDIF()

ENDMACRO()
```

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

TribitsExampleProject/`cmake/CallbackDefineRepositoryPackaging.cmake`

provides a good example which is:

```
MACRO(TRIBITS_REPOSITORY_DEFINE_PACKAGING)

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

ENDMACRO()
```

As shown in the above example, it is important to prefix the excluded files and directories with the repository base directory `${${REPOSITORY_NAME}_SOURCE_DIR}/` since these are interpreted by CPack as regular-expressions.

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

`REPOSITORY_NAME`

The name of the current TriBITS repository. This name will be the repository name listed in <projectDir>/cmake/ExtraRepositoriesList.cmake file or if this repository directory is the project base directory, `REPOSITORY_NAME` will be set to `${PROJECT_NAME}`.

`REPOSITORY_DIR`

Path of the current Repository *relative* to the Project's base source directory ${PROJECT_NAME}_SOURCE_DIR.. This is typically just the repository name but can be an arbitrary directory if specified through the <projectDir>/cmake/ExtraRepositoriesList.cmake file.

The following project-scope (non-cache) local variables are set once the list of TriBITS repositories is processed and before any of the repository's files are processed:

`${REPOSITORY_NAME}_SOURCE_DIR`

The absolute path to the base of a given TriBITS Repository's 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 TriBITS Project (but does hard-code the repository name which is not ideal).

`${REPOSITORY_NAME}_BINARY_DIR`

The absolute path to the base of a given TriBITS Repository's 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.

The following project-level local varaibles can be defined by the project or the user to help define the what packages from the repository `${REPOSITORY_NAME}` contribute to the primary meta-project packages (PMPP):

`${REPOSITORY_NAME}_NO_PRIMARY_META_PROJECT_PACKAGES`

If set to `TRUE`, then the package's in the TriBITS repository are not considered to be part of the primary meta-project packages. This affects what packages get enabled by default when enabling all packages with setting `${PROJECT_NAME}_ENABLE_ALL_PACKAGES=ON` and what tests and examples get enabled by default when setting `${PROJECT_NAME}_ENABLE_TESTS=ON`. See TriBITS Dependency Handling Behaviors for more details.

`${REPOSITORY_NAME}_NO_PRIMARY_META_PROJECT_PACKAGES_EXCEPT`

When the above variable is set to `TRUE`, this variable is read by TriBITS to find the list of TriBITS packages selected packages in the repository `${REPOSITORY_NAME}` which are considered to be part of the set of the project's primary meta-project package when the above variable is set to `ON`. NOTE: It is not necessary to list all of the subpackages in a given parent package. Only the parent package need be listed and it will be equivalent to listing all of the subpackages. See TriBITS Dependency Handling Behaviors for more details.

The above primary meta-project variables should be set in the meta-project's <projectDir>/ProjectName.cmake file so that they will be set in all situations.

TriBITS Package

A TriBITS Package:

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

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

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

- TriBITS Package Core Files
- TriBITS Package Core Variables

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

```
<packageDir>/
  CMakeLists.txt  # Only processed if the package is enabled
  cmake/
    Dependencies.cmake  # Always processed if its repo is processed
    <packageName>_config.h.in  # [Optional], name is not fixed
```

There are a few simple rules for the location and the contents of the `<packageDir>/` directory:

- The directory `<packageDir>/` must not be a subdirectory of the package directory of any other SE package (e.g. not `pkga/pkgb`).
- All of the source files, test files, etc. for the package should be included under `<packageDir>/`.

The above rules are not needed for basic building and testing but are needed for extended features like automatically detecting when a package has changed by looking at what files have changed (see Pre-push Testing using checkin-test.py) and for creating source tarballs correctly (see Creating Source Distributions). Therefore, it would be wise to abide by the above rules when defining packages.

The following TriBITS Package files are documented in more detail below:

- <packageDir>/cmake/Dependencies.cmake
- <packageDir>/cmake/<packageName>_config.h.in
- <packageDir>/CMakeLists.txt

<packageDir>/cmake/Dependencies.cmake: [Required] Defines the dependencies for a given TriBITS (SE) package using the macro TRIBITS_PACKAGE_DEFINE_DEPENDENCIES(). This file is processed at the top-level project scope (using an `INCLUDE()`) so any local variables set will be seen by the entire project. This file is always processed, including when just building the project's dependency data-structure (see Reduced Package Dependency Processing).

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

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

An example of a package with subpackages is `WithSubpackages` which has the dependencies file:

TribitsExampleProject/`packages/with_subpackages/cmake/Dependencies.cmake`

which is:

```
TRIBITS_PACKAGE_DEFINE_DEPENDENCIES(
  SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS
    A   a   PT  REQUIRED
    B   b   ST  OPTIONAL
    C   c   ST  OPTIONAL
  )
```

`WithSubpackages` defines three subpackages which creates three new SE packages with names `WithSubpackagesA`, `WithSubpackagesB`, and `WithSubpackagesC`.

<packageDir>/cmake/<packageName>_config.h.in: [Optional] The package's configured header file. This file will contain placeholders for variables that will be substitute at configure time with TRIBITS_CONFIGURE_FILE(). This includes usage of `#cmakedefine <varName>` and other standard CMake file configuration features.

NOTE: The file name `<packageName>_config.h.in` is not at all fixed and the package can call this file anything it wants. Also, a package can configure multiple header files in different directories for different purposes using TRIBITS_CONFIGURE_FILE() or even calls to the raw CMake function CONFIGURE_FILE().

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

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

TribitsExampleProject/`packages/simple_cxx/CMakeLists.txt`

which is:

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

#
# B) Platform-specific checks
#
INCLUDE(CheckFor__int64)
CHECK_FOR___INT64(HAVE_SIMPLECXX___INT64)

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

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

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

The first command at the top of the file is a call to TRIBITS_PACKAGE() which takes the package name (`SimpleCxx` in this case) in addition to a few other options. While TriBITS obviously already knows the package name (since it read it from the <repoDir>/PackagesList.cmake file), the purpose for repeating it in this call is as documentation for the

developer's sake (and this name is checked against the expected package name). Then a set of configure-time tests are typically performed (if the package needs any of these). In this example, the existence of the C++ `__int64` data-type is checked using the module `CheckFor__int64.cmake` (which is in the `cmake/` directory of this package. (CMake has great support Configure-time System Tests.) This is followed by package-specific options. In this case, the standard TriBITS options for debug checking and deprecated warnings are added using the standard macros TRIBITS_ADD_DEBUG_OPTION() and TRIBITS_ADD_SHOW_DEPRECATED_WARNINGS_OPTION(). After all of this up-front stuff is complete (which will be present in any moderately complex CMake-configured project) the source and the test sub-directories are added that actually define the library and the tests. In this case, the standard TRIBITS_ADD_TEST_DIRECTORIES() macro is used which only conditionally adds the tests for the package.

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

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

When a TriBITS package is broken up into subpackages (see TriBITS Subpackage), its `CMakeLists.txt` file looks a little different from a package with no subpackages as shown above. The basic structure of this file for a **package with subpackages** is shown in:

TribitsExampleProject/`packages/with_subpackages/CMakeLists.txt`

which contains:

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

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

#
# C) Process the subpackages
#
TRIBITS_PROCESS_SUBPACKAGES()

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

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

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

TriBITS Package Core Variables A packages' core variables are broken down into the following categories:

- TriBITS Package Local Variables
- TriBITS Package Top-Level Local Variables
- TriBITS Package Cache Variables
- TriBITS Package Optional Dependency Macro Variables

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

`PACKAGE_NAME`

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

`PACKAGE_SOURCE_DIR`

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

`PACKAGE_BINARY_DIR`

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

`PACKAGE_NAME_UC`

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

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

`${PACKAGE_NAME}_SOURCE_DIR`

The absolute path to the package's base 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. NOTE: This variable is defined for all declared packages that exist, independent of whether they are enabled or not. This varible is set as soon as it is known if the given package exists or not.

`${PACKAGE_NAME}_BINARY_DIR`

The absolute path to the package's base 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. NOTE: This variable is **only** defined if the package is actually enabled!

`${PACKAGE_NAME}_PARENT_REPOSITORY`

The name of the package's parent repository. This can be used by a package to access information about its parent repository. For example, the variable `${${PACKAGE_NAME}_PARENT_REPOSITORY}_SOURCE_DIR` can be dereferenced and read of needed (but it is not recommended that packages be aware of their parent repository in general)..

`${PACKAGE_NAME}_TESTGROUP`

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

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

`${PROJECT_NAME}_ENABLE_${PACKAGE_NAME}`

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

`${PACKAGE_NAME}_ENABLE_${OPTIONAL_DEP_PACKAGE_NAME}`

Set to `ON` if support for the optional upstream dependent package `${OPTIONAL_DEP_PACKAGE_NAME}` is enabled in package `${PACKAGE_NAME}`. Here `${OPTIONAL_DEP_PACKAGE_NAME}` corresponds to each optional upstream SE package listed in the `LIB_OPTIONAL_PACKAGES` and `TEST_OPTIONAL_PACKAGES` arguments to the TRIBITS_PACKAGE_DEFINE_DEPENDENCIES() macro.

NOTE: It is important that the CMake code in the package's `CMakeLists.txt` files key off of this variable and **not** the project-level variable `${PROJECT_NAME}_ENABLE_${OPTIONAL_DEP_PACKAGE_NAME}` because the package-level variable `${PACKAGE_NAME}_ENABLE_${OPTIONAL_DEP_PACKAGE_NAME}` can be explicitly turned off by the user even through the packages `${PACKAGE_NAME}` and `${OPTIONAL_DEP_PACKAGE_NAME}` are both enabled at the project level! See Support for optional SE package/TPL can be explicitly disabled.

`${PACKAGE_NAME}_ENABLE_${OPTIONAL_DEP_TPL_NAME}`

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

`${PACKAGE_NAME}_ENABLE_TESTS`

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

`${PACKAGE_NAME}_ENABLE_EXAMPLES`

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

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

The following local *TriBITS Package Optional Dependency Macro Variables* are defined in the top-level project scope before a (SE) Package's `CMakeLists.txt` file is processed:

`HAVE_<PACKAGE_NAME_UC>_<OPTIONAL_DEP_PACKAGE_NAME_UC>`

Set to `ON` if support for optional upstream package `${OPTIONAL_DEP_PACKAGE}` is enabled in downstream package `${PACKAGE_NAME}` (i.e. ${PACKAGE_NAME}_ENABLE_${OPTIONAL_DEP_PACKAGE_NAME} = `ON`) and is set to `FALSE` otherwise. Here, `<PACKAGE_NAME_UC>` and `<OPTIONAL_DEP_PACKAGE_NAME_UC>` are the upper-case names for the packages `${PACKAGE_NAME}` and `${OPTIONAL_DEP_PACKAGE_NAME}`, respectively. For example, if optional support for upstream package `Triutils` is enabled in downstream package `EpetraExt` in ReducedMockTrilinos, then `EpetraExt_ENABLE_TriUtils=ON` and `HAVE_EPETRAEXT_TRIUTILS=ON`. This variable is meant to be used in:

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

in configured header files (e.g. <packageDir>/cmake/<packageName>_config.h.in). For example, for the `EpetraExt` and `Triutils` example, this would be:

```
#cmakedefine HAVE_EPETRAEXT_TRIUTILS
```

`HAVE_<PACKAGE_NAME_UC>_<OPTIONAL_DEP_TPL_NAME_UC>`

Set to `ON` if support for the optional TPL `${OPTIONAL_DEP_TPL_NAME}` is enabled in downstream package `${PACKAGE_NAME}` (i.e. ${PACKAGE_NAME}_ENABLE_${OPTIONAL_DEP_TPL_NAME} = `ON`) and is set to `FALSE` otherwise. Here, `<PACKAGE_NAME_UC>` and `<OPTIONAL_DEP_TPL_NAME_UC>` are the upper-case names for the packages `${PACKAGE_NAME}` and `${OPTIONAL_DEP_TPL_NAME}`, respectively. For example, if optional support for the TPL `Boost` is enabled in the package `Teuchos` in ReducedMockTrilinos, then `Teuchos_ENABLE_Boost=ON` and `HAVE_TUECHOS_BOOST=ON`. This variable is meant to be used in:

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

in configured header files (e.g. <packageDir>/cmake/<packageName>_config.h.in). For example, for the `Teuchos` and `Boost` example, this would be:

```
#cmakedefine HAVE_TEUCHOS_BOOST
```

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). However, this may be undesirable because it will make it hard to pull a package out of one TriBITS repository and place it in another repository for a different use. However, a package can indirectly refer to its own repository without loss of generality by reading the variable `${PACKAGE_NAME}_PARENT_REPOSITORY`. The problem is referring to other TriBITS repositories explicitly.

TriBITS Subpackage

A TriBITS Subpackage:

- Is a compartmentalization of a parent TriBITS Package according to Software Engineering Packaging Principles.
- Typically defines a set of libraries and/or header files and/or executables and/or tests with CMake build targets for building these for which TriBITS exports the list of include directories, libraries, and targets that are created (along with CMake dependencies).
- Is declared in its parent packages's <packageDir>/cmake/Dependencies.cmake file in a call to TRIBITS_PACKAGE_DEFINE_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 <packageDir>/<spkgDir>/cmake/Dependencies.cmake using the macro TRIBITS_PACKAGE_DEFINE_DEPENDENCIES().
- Can **NOT** have its own subpackages defined (only top-level packages can have subpackages).
- Is enabled or disabled along with all other subpackages in the parent package automatically if it's parent package is enabled or disabled with `${PROJECT_NAME}_ENABLE_${PARENT_PACKAGE_NAME}` set to `ON` or `OFF` respectively (see Enable/disable of parent package is enable/disable for subpackages and PARENT_PACKAGE_NAME).
- Has tests turned on automatically if `${PARENT_PACKAGE_NAME}_ENABLE_TESTS=ON`.

The contents of a TriBITS Subpackage are almost identical to those of a TriBITS Package. The differences are described below and in How is a TriBITS Subpackage is different from a TriBITS Package?.

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

- TriBITS Subpackage Core Files
- TriBITS Subpackage Core Variables

TriBITS Subpackage Core Files The set of core files for a subpackage are identical to the TriBITS Package Core Files and are:

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

(where `<packageDir> = ${${PARENT_PACKAGE_NAME}_SOURCE_DIR}` and `<spkgDir>` is the subpackage directory listed in the SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS to TRIBITS_PACKAGE_DEFINE_DEPENDENCIES()).

There are a few simple rules for the location and the contents of the `<spkgDir>/` directory:

- The relative directory `<spkgDir>/` should be a strict subdirectory of `<packageDir>/` (e.g. not `../../somewhereelse`).
- The directory `<spkgDir>/` must not be a subdirectory of the package directory of any other subpackage (e.g. not `spkga/spkgb`).
- All of the source files, test files, etc. for the subpackage should be included under `<spkgDir>/`.

The above rules are not needed for basic building and testing but are needed for extended features like automatically detecting when a package has changed by looking at what files have changed (see Pre-push Testing using checkin-test.py) and for creating source tarballs correctly (see Creating Source Distributions). Therefore, it would be wise to abide by the above rules when defining subpackages.

These TriBITS Subpackage files are documented in more detail below:

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

<packageDir>/<spkgDir>/cmake/Dependencies.cmake: [Required] The contents of this file for subpackages are identical as for top-level packages. It just contains a call to the macro TRIBITS_PACKAGE_DEFINE_DEPENDENCIES() to define this SE package's upstream TPL and SE package dependencies. A simple example is for the example subpackage `WithSubpackagesB` (declared in with_subpackages/cmake/Dependencies.cmake) with the file:

TribitsExampleProject/`packages/with_subpackages/b/cmake/Dependencies.cmake`

which is:

```
TRIBITS_PACKAGE_DEFINE_DEPENDENCIES(
  LIB_REQUIRED_PACKAGES  SimpleCxx
  LIB_OPTIONAL_PACKAGES  WithSubpackagesA  InsertedPkg
  TEST_OPTIONAL_PACKAGES  MixedLang
  )
```

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

```
'WithSubpackagesA' = 'WithSubpackages' + 'A'
```

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

TribitsExampleProject/`packages/with_subpackages/a/cmake/Dependencies.cmake`

which is:

```
TRIBITS_PACKAGE_DEFINE_DEPENDENCIES(
  LIB_REQUIRED_PACKAGES SimpleCxx
  )
```

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

```
TRIBITS_PACKAGE_DEFINE_DEPENDENCIES(
  LIB_REQUIRED_TPLS Boost
  SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS
    A   A   PT  REQUIRED
    ...
  )
```

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

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

TribitsExampleProject/`packages/with_subpackages/a/CMakeLists.txt`

which contains:

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

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

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

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

TRIBITS_ADD_TEST_DIRECTORIES(tests)

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

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

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

`PARENT_PACKAGE_NAME`

The name of the parent package.

`PARENT_PACKAGE_SOURCE_DIR`

The absolute path to the parent package's base source directory.

`PARENT_PACKAGE_BINARY_DIR`

The absolute path to the parent package's base binary directory.

How is a TriBITS Subpackage is different from a TriBITS Package? A common question this is natural to ask is how a TriBITS Subpackage is different from a TriBITS Package? They contain the same basic files (i.e. a `cmake/Dependencies.cmake` file, a top-level `CMakeLists.txt` file, source files, test files, etc.). They both are included in the list of TriBITS SE Packages and therefore can both be enabled/disabled by the user or in automatic dependency logic (see Package Dependencies and Enable/Disable Logic). The primary difference is that a subpackage is meant to involve less overhead and is to be used to partition the parent package's software into chunks according to Software Engineering Packaging Principles. Also, the dependency logic treats a parent package's subpackages as part of itself so when the parent package is explicitly enabled or disabled, it is identical to explicitly enabling or disabling all of its subpackages (see Enable/disable of parent package is enable/disable for subpackages). Also, subpackages are tested along with their peer subpackages with the parent package as part of TriBITS Package-by-Package CTest/Dash Driver. This effectively means that if a build failure is detected in any subpackage, then that will effectively disable the parent package and all of its other subpackages in downstream testing. This is a type of "all for one and all for one" when it comes to the relationship between the subpackages within a single parent package. These are some of the issues to consider when breaking up software into packages and subpackages that will be mentioned in other sections as well.

TriBITS TPL

A *TriBITS TPL*:

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

WARNING: One must be very careful to pick **Globally unique TriBITS TPL names** across all TPLs in all TriBITS repositories that ever might be cobbled together into a single TriBITS (meta) project! However, choosing TPL names is usually much easier and less risky than choosing Globally unique TriBITS package names because the native TPLs themselves tend to be uniquely named. For example, the TPL names `BLAS` and `LAPACK` are well defined in the applied math and computational science community and are not likely to clash.

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

Dependencies and Enable/Disable Logic. For example, if a TPL is explicitly disabled, all of the downstream packages that depend on that TPL will be automatically disabled as well (see TPL disable triggers auto-disables of downstream dependencies).

For each TPL referenced in a <repoDir>/TPLsList.cmake file using the macro TRIBITS_REPOSITORY_DEFINE_TPLS(), there must exist a file, typically called `FindTPL${TPL_NAME}.cmake`, that once processed, produces the variables `TPL_${TPL_NAME}_LIBRARIES` and `TPL_${TPL_NAME}_INCLUDE_DIRS`. Most `FindTPL${TPL_NAME}.cmake` files just use the function TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES() the define the TriBITS TPL. A simple example of such a file is the common TriBITS `FindTPLPETSC.cmake` module which is currently:

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

Some concrete `FindTPL${TPL_NAME}.cmake` files actually do use `FIND_PACKAGE()` and a standard CMake package find module to fill in the guts of finding at TPL which is perfectly fine. In this case, the purpose for the wrapping `FindTPL${TPL_NAME}.cmake` is to standardize the output variables `TPL_${TPL_NAME}_INCLUDE_DIRS` and `TPL_${TPL_NAME}_LIBRARIES`. For more details on properly using `FIND_PACKAGE()` to define a `FindTPL${TPL_NAME}.cmake` file, see How to use FIND_PACKAGE() for a TriBITS TPL.

Once the <repoDir>/TPLsList.cmake files are all processed, then each defined TPL `TPL_NAME` is assigned the following global non-cache variables:

`${TPL_NAME}_FINDMOD`

> Gives the location for the TPL's find module. This is set using the `FINDMOD` field in the call to TRIBITS_REPOSITORY_DEFINE_TPLS(). The final value of the variable is defined by the *last* <repoDir>/TPLsList.cmake file that is processed that declares the TPL `TPL_NAME`. For example, if `Repo1/TPLsList.cmake` and `Repo2/TPLsList.cmake` both list the TPL `SomeTpl`, then if `Repo2` is processed after `Repo1`, then `SomeTpl_FINDMOD` is determined by `Repo2/TPLsList.cmake` and the find module listed in `Repo1/TPLsList.cmake` is ignored.

`${TPL_NAME}_TESTGROUP`

> Gives the TPL's SE Package Test Group. This is set using the `CLASSIFICATION` field in the call to TRIBITS_REPOSITORY_DEFINE_TPLS(). If multiple repos define a given TPL, then the *first* <repoDir>/TPLsList.cmake file that is processed that declares the TPL `TPL_NAME` specifies the test group. For example, if `Repo1/TPLsList.cmake` and `Repo2/TPLsList.cmake` both list the TPL `SomeTpl`, then if `Repo2` is processed after `Repo1`, then `SomeTpl_TESTGROUP` is determined by `Repo1/TPLsList.cmake` and the test group in `Repo2/TPLsList.cmake` is ignored. However, if `${TPL_NAME}_TESTGROUP` is already set before the <repoDir>/TPLsList.cmake files are processed, then that test group will be used. Therefore, the project can override the test group for a given TPL if desired.

The specification given in Enabling support for an optional Third-Party Library (TPL) and TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES() describes how the a `FindTPL${TPL_NAME}.cmake` module should behave and allow users to override and specialize how a TPL's include directories and libraries are determined. However, note that the TriBITS system does not require the usage of the function `TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES()` and does not even care about the TPL module name `FindTPL${TPL_NAME}.cmake`. All that is required is that some CMake file fragment exist that once included, will define the variables `${TPL_NAME}_LIBRARIES` and `${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_FIND_INCLUDE_DIRS_AND_LIBRARIES()` function.

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

3.2 Processing of TriBITS Files: Ordering and Details

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

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

Another factor that is important to understand is the scoping in which the various files are processed (with `INCLUDE()` or `ADD_SUBDIRECTORY()`). This scoping has a large impact on the configuration of the project and what effect the processing of files and setting variables have on the project as a whole. Some of the strange scoping rules for CMake are discussed in CMake Language Overview and Gotchas and should be understood before trying to debug issues with processing. Many of the basic files are processed (included) in the base project <projectDir>/CMakeLists.txt scope and therefore any local variables set in these files are accessible to the entire CMake project (after the file is processed, of course). Other files get processed inside of functions which have their own local scope and therefore only impact the rest of the project in more purposeful ways. And of course all of the package <packageDir>/CMakeLists.txt files that are processed using `ADD_SUBDIRECTORY()` create a new local scope for that given package.

Full TriBITS Project Configuration

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

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

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

Full Processing of TriBITS Project Files:

1. Read <projectDir>/ProjectName.cmake (sets PROJECT_NAME)
2. Call `PROJECT(${PROJECT_NAME} NONE)` (sets ${PROJECT_NAME}_SOURCE_DIR and ${PROJECT_NAME}_BINARY_DIR)
3. Execute TRIBITS_PROJECT():
   1) Set PROJECT_SOURCE_DIR and PROJECT_BINARY_DIR
   2) For each `<optFile>` in ${${PROJECT_NAME}_CONFIGURE_OPTIONS_FILE}:
      * `INCLUDE(<optFile>)`
   3) Set variables `CMAKE_HOST_SYSTEM_NAME` and `${PROJECT_NAME}_HOSTNAME`
      (both of these can be overridden in the cache by the user)
   4) Find Python (sets `PYTHON_EXECUTABLE`, see Python Support)
   5) `INCLUDE(` <projectDir>/Version.cmake `)`
   6) Define primary TriBITS options and read in the list of extra repositories
      (calls `TRIBITS_DEFINE_GLOBAL_OPTIONS_AND_DEFINE_EXTRA_REPOS()`)
      * `INCLUDE(` <projectDir>/cmake/ExtraRepositoriesList.cmake `)`
   7) For each `<repoDir>` in all defined TriBITS repositories:
      * `INCLUDE(` <repoDir>/cmake/CallbackSetupExtraOptions.cmake `)`
      * Call macro `TRIBITS_REPOSITORY_SETUP_EXTRA_OPTIONS()`
   9) Call `TRIBITS_READ_PACKAGES_PROCESS_DEPENDENCIES_WRITE_XML()`:
      a) For each `<repoDir>` in all defined TriBITS repositories:

* INCLUDE (<repoDir>/PackagesList.cmake) and process list
* INCLUDE (<repoDir>/TPLsList.cmake) and process list

b) For each <repoDir> in all defined TriBITS repositories:
* INCLUDE (<repoDir>/cmake/RepositoryDependenciesSetup.cmake)

c) INCLUDE (<projectDir>/cmake/ProjectDependenciesSetup.cmake)

d) For each <packageDir> in all defined top-level packages:
* INCLUDE (<packageDir>/cmake/Dependencies.cmake)
  - Sets all package-specific options (see TriBITS Package Cache Variables)
* For each <spkgDir> in all subpackages for this package:
  * INCLUDE (<packageDir>/<spkgDir>/cmake/Dependencies.cmake)
    - Sets all subpackage-specific options

10) Adjust SE package and TPLs enables and disables
(see Package Dependencies and Enable/Disable Logic)

11) Probe and set up the environment (finds MPI, compilers, etc.)
(see TriBITS Environment Probing and Setup)

12) For <tplName> in the set of enabled TPLs:
* INCLUDE(${<tplName>_FINDMOD}) (see TriBITS TPL)

13) For each <repoDir> in all defined TriBITS repositories:
* Read <repoDir>/Copyright.txt
* INCLUDE (<repoDir>/Version.cmake)
(see Project and Repository Versioning and Release Mode)

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

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

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

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

Reduced Package Dependency Processing

In addition to the full processing that occurs as part of the Full TriBITS Project Configuration, there are also TriBITS tools that only process as subset of project's files. This reduced processing is performed in order to build up the project's package dependencies data-structure and to write the file <Project>PackageDependencies.xml. For example, the script checkin-test.py and the function TRIBITS_CTEST_DRIVER() both drive this type of processing. In particular, the CMake -P script TribitsDumpDepsXmlScript.cmake reads all of the project's dependency-related files and dumps out the <Project>PackageDependencies.xml file. This reduced processing (as executed in cmake -P TribitsDumpDepsXmlScript.cmake) is described below.

Reduced Dependency Processing of TriBITS Project:

1. Read <projectDir>/ProjectName.cmake (sets PROJECT_NAME)
2. INCLUDE (<projectDir>/cmake/ExtraRepositoriesList.cmake)
3. Call TRIBITS_READ_PACKAGES_PROCESS_DEPENDENCIES_WRITE_XML():
   a) For each <repoDir> in all defined TriBITS repositories:
      * INCLUDE (<repoDir>/PackagesList.cmake)
      * INCLUDE (<repoDir>/TPLsList.cmake)
   b) For each <repoDir> in all defined TriBITS repositories:

* INCLUDE (<repoDir>/cmake/RepositoryDependenciesSetup.cmake)

c) INCLUDE (<projectDir>/cmake/ProjectDependenciesSetup.cmake)

d) For each <packageDir> in all defined top-level packages:

* INCLUDE (<packageDir>/cmake/Dependencies.cmake)
  - Sets all package-specific options (see TriBITS Package Cache Variables)
* For each <spkgDir> in all subpackages for this package:
  * INCLUDE (<packageDir>/<spkgDir>/cmake/Dependencies.cmake)
    - Sets all subpackage-specific options

e) Write the file <Project>PackageDependencies.xml
(specified using ${PROJECT_NAME}_DEPS_XML_OUTPUT_FILE)

When comparing the above reduced dependency processing to the Full Processing of TriBITS Project Files it is important to note that that several files are **not** processed in the reduced algorithm shown above. The files that are **not** processed include <projectDir>/Version.cmake, <repoDir>/Version.cmake and <repoDir>/cmake/CallbackSetupExtraOptions.cmake (in addition to not processing any of the `CMakeLists.txt` files obviously). Therefore, one cannot put anything in these non-processed files that would impact the definition of TriBITS repositories, packages, TPLs, etc. Anything that would affect the dependencies data-structure that gets written out as <Project>PackageDependencies.xml must be contained in the files that are processed shown in the reduced processing above.

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

File Processing Tracing

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

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

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

looks something like:

```
-- File Trace: PROJECT    INCLUDE    [...]/Version.cmake
-- File Trace: REPOSITORY INCLUDE    [...]/cmake/CallbackSetupExtraOptions.cmake
-- File Trace: REPOSITORY INCLUDE    [...]/PackagesList.cmake
-- File Trace: REPOSITORY INCLUDE    [...]/TPLsList.cmake
-- File Trace: PACKAGE    INCLUDE    [...]/packages/simple_cxx/cmake/Dependencies.cmak
-- File Trace: PACKAGE    INCLUDE    [...]/packages/mixed_lang/cmake/Dependencies.cmak
-- File Trace: PACKAGE    INCLUDE    [...]/packages/with_subpackages/cmake/Dependencie
-- File Trace: PACKAGE    INCLUDE    [...]/packages/with_subpackages/a/cmake/Dependenc
-- File Trace: PACKAGE    INCLUDE    [...]/packages/with_subpackages/b/cmake/Dependenc
-- File Trace: PACKAGE    INCLUDE    [...]/packages/with_subpackages/c/cmake/Dependenc
-- File Trace: PACKAGE    INCLUDE    [...]/packages/wrap_external/cmake/Dependencies.c
```

```
-- File Trace: PROJECT    CONFIGURE  [...]/cmake/ctest/CTestCustom.cmake.in
-- File Trace: REPOSITORY READ       [...]/Copyright.txt
-- File Trace: REPOSITORY INCLUDE    [...]/Version.cmake
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/simple_cxx/CMakeLists.txt
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/simple_cxx/test/CMakeLists.txt
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/mixed_lang/CMakeLists.txt
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/mixed_lang/test/CMakeLists.txt
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/with_subpackages/CMakeLists.txt
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/with_subpackages/a/CMakeLists.txt
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/with_subpackages/a/tests/CMakeList
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/with_subpackages/b/CMakeLists.txt
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/with_subpackages/b/tests/CMakeList
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/with_subpackages/c/CMakeLists.txt
-- File Trace: PACKAGE    ADD_SUBDIR [...]/packages/with_subpackages/c/tests/CMakeList
-- File Trace: PROJECT    INCLUDE    [...]/cmake/CallbackDefineProjectPackaging.cmake
-- File Trace: REPOSITORY INCLUDE    [...]/cmake/CallbackDefineRepositoryPackaging.cma
```

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

3.3 Coexisting Projects, Repositories, and Packages

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

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

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

```
TRIBITS_REPOSITORY_DEFINE_PACKAGES(
  extraRepoOnePackage    .     ST
  )
```

(Note the dot `'.'` for the package directory.)

This is also how the real TriBITS repository and package DataTransferKit is set up (at least that is the way it was when this document was first written).

However, to maximize flexibility, it is recommended that a TriBITS package and its TriBITS repository **not** share the same directory or the same name. This allows a TriBITS repository to define more packages later.

TriBITS Package Dir == TriBITS Repository Dir == TriBITS Project Dir: In the extreme, it is possible to collapse a single TriBITS package, repository, and project into the same base source directory. They can also share the same name for the package, repository and package. One example of this is the `TriBITS` project and The TriBITS Test Package themselves, which are both rooted in the base `TriBITS/` source directory of the stand-alone TriBITS repository. There are a few restrictions and modifications needed to get this to work:

- The base `CMakeLists.txt` file must be modified to allow it to be processed both as the base project `CMakeLists.txt` file and as the package's base `CMakeLists.txt` file. In the case of `TriBITS/CMakeLists.txt`, a big if statement is used.

Other than that simple modification to the top-level `CMakeLists.txt` file, a TriBITS project, repository, and package can all be rooted in the same source directory.

The primary use case for collapsing a project, repository, and package into a single base source directory would be to support the stand-alone build of a TriBITS package as its own entity that uses an independent installation of the TriBITS (or a minimal snapshot of TriBITS). If a given TriBITS package has no required upstream TriBITS package dependencies and minimal TPL dependencies (or only uses Standard TriBITS TPLs or Common TriBITS TPLs already defined in the `tribits/core/std_tpls/` or `tribits/common_tpls/` directories), then creating a stand-alone project build of a single TriBITS package requires fairly little extra overhead or duplication.

3.4 Standard and Common TPLs

While a TriBITS Repository can define their own TPLs and their own TPL find modules (see TriBITS TPL), the TriBITS source tree contains the find modules for a few different standard TPLs and common TPLs. Standard TriBITS TPLs are integral to the TriBITS system itself while Common TriBITS TPLs are TPL that are used in several different TriBITS Repositories and are contained in TriBITS for convenience and uniformity.

Standard TriBITS TPLs

TriBITS contains find modules for a few standard TPLs integral to the TriBITS system. The standard TriBITS TPLs are contained under the directory:

```
tribits/core/std_tpls/
```

The current list of standard TriBITS TPL find modules is:

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

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

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

```
TRIBITS_REPOSITORY_DEFINE_TPLS(
  MPI   "${${PROJECT_NAME}_TRIBITS_DIR}/core/std_tpls/"  PT
  CUDA  "${${PROJECT_NAME}_TRIBITS_DIR}/core/std_tpls/"  ST
  ...
  )
```

Common TriBITS TPLs

TriBITS also contains find modules for several TPLs that are used across many independent TriBITS repositories. The goal of maintaining these under TriBITS is to enforce conformity in case these independent repositories are combined into a single meta-project.

The common TriBITS TPLs are contained under the directory:

```
tribits/common_tpls/
```

The current list of common TriBITS TPL find modules is:

```
FindTPLBinUtils.cmake
FindTPLBLAS.cmake
FindTPLBoost.cmake
FindTPLHDF5.cmake
FindTPLLAPACK.cmake
FindTPLPETSC.cmake
```

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

```
TRIBITS_REPOSITORY_DEFINE_TPLS(
  BLAS   "${${PROJECT_NAME}_TRIBITS_DIR}/common_tpls/"  PT
  LAPACK  "${${PROJECT_NAME}_TRIBITS_DIR}/common_tpls/"  PT
  ...
  )
```

By using a standard TPL definition, it is guaranteed that the TPL used will be consistent with all of the TriBITS packages that depend on these TPLs in case they are combined into a single project.

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

4 Example TriBITS Projects

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

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

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

4.1 TribitsHelloWorld

`TribitsHelloWorld` is about the simplest possible TriBITS project that you can imagine and is contained under the directory:

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

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

The directory structure for this example his shown below:

```
TribitsHelloWorld/
  CMakeLists.txt
  PackagesList.cmake
  ProjectName.cmake
```

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

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

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

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

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

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

4.2 TribitsExampleProject

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

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

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

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

This simple project shows how what is listed in files:

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

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

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

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

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

The variable `PROJECT_NAME=TribitsExProj` is used to prefix (using `"${PROJECT_NAME}_"`) all of the project's global TriBITS variables like `TribitsExProj_ENABLE_TESTS`, `TribitsExProj_ENABLE_ALL_PACKAGES`, etc.

The directory structure and key files for this example project are shown in the partial list of TribitsExampleProject Files and Directories below:

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

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

```
TRIBITS_REPOSITORY_DEFINE_PACKAGES(
  SimpleCxx          packages/simple_cxx         PT
```

```
  MixedLang           packages/mixed_lang         PT
  InsertedPkg         InsertedPkg                 ST
  WithSubpackages     packages/with_subpackages   PT
  WrapExternal        packages/wrap_external      ST
  )

TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS(WrapExternal Windows)
TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES(InsertedPkg)
```

From this file, we get the list of top-level packages `SimpleCxx`, `MixedLang`, `WithSubpackages`, and `WrapExternal` (and their base package directories and testing group, see <repoDir>/PackagesList.cmake). (NOTE: By default the package `InsertedPkg` is not defined because its directory is missing, see How to insert a package into an upstream repo.)

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

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

```
TribitsExampleProject/packages/with_subpackages/cmake/Dependencies.cmake
```

with contents:

```
TRIBITS_PACKAGE_DEFINE_DEPENDENCIES(
  SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS
    A   a   PT  REQUIRED
    B   b   ST  OPTIONAL
    C   c   ST  OPTIONAL
  )
```

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

```
TribitsExampleProject/packages/with_subpackages/a/
TribitsExampleProject/packages/with_subpackages/b/
TribitsExampleProject/packages/with_subpackages/c/
```

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

```
SimpleCxx MixedLang WithSubpackagesA WithSubpackagesB WithSubpackagesC \
  WithSubpackages WrapExternal 7
```

The above list of SE packages is printed (with the number of packages printed at the end) by TriBITS to the `cmake` stdout on the line starting with `"Final set of non-enabled SE packages:"` when no packages are enabled (see Selecting the list of packages to enable). (Note that TriBITS does not put in line-brakes with continuation characters `"\"` as shown above.) TriBITS defines enable/disable cache variables for each of these defined SE packages like `TribitsExProj_ENABLE_SimpleCxx` and `TribitsExProj_ENABLE_WithSubpackagesA`, and defines all the variables listed in TriBITS Package Cache Variables that are settable by the users or by the dependency logic described in section Package Dependencies and Enable/Disable Logic.

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

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

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

4.3 MockTrilinos

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

```
tribits/examples/MockTrilinos/
```

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

```
tribits/examples/MockTrilinos/PackagesList.cmake
```

is:

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

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

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

All of the package directories listed above have <packageDir>/cmake/Dependencies.cmake files but generally do not have <packageDir>/CMakeLists.txt files since most of usage of `MockTrilinos` just involves the testing of the algorithms and behaviors described in the section Package Dependencies and Enable/Disable Logic.

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

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

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

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

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

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

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

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

4.4 ReducedMockTrilinos

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

```
tribits/examples/ReducedMockTrilinos/
```

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

The list of packages in:

```
tribits/examples/ReducedMockTrilinos/PackagesList.cmake
```

is:

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

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

4.5 Trilinos

The real Trilinos project and repository itself is an advanced example for the usage of TriBITS. Almost every single-repository use case for TriBITS is demonstrated somewhere in Trilinos. While some of the usage of TriBITS in Trilinos may not be not exemplary (e.g., because it represents old usage, or was written by CMake/TriBITS beginners) it does represent real working usage. Given that Trilinos is a widely available software repository, anyone should be able to access a newer version of Trilinos and mine it for CMake and TriBITS examples.

4.6 The TriBITS Test Package

The last TriBITS example mentioned here is the TriBITS test package named (appropriately) `TriBITS` itself defined in the `TriBITS` repository. The directory for the `TriBITS` test package is the base TriBITS source directory `tribits`. This allows any TriBITS project to add testing for the TriBITS system by just listing the TriBITS repository in its <projectDir>/cmake/ExtraRepositoriesList.cmake file. Trilinos lists the TriBITS repository in its `ExtraRepositoriesList.cmake` file as:

> **TRIBITS_PROJECT_DEFINE_EXTRA_REPOSITORIES(** TriBITS “” GIT https://github.com/TriBITSPub/TriBITS “” Continuous ...)

No downstream TriBITS packages list a dependency on `TriBITS` in their <packageDir>/cmake/Dependencies.cmake files. Defining the `TriBITS` TriBITS package in only done for running the TriBITS tests.

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

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

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

5 Package Dependencies and Enable/Disable Logic

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

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

This section is organized and broken-down as follows. First, the subsection Example ReducedMockTrilinos Project Dependency Structure presents the ReducedMockTrilinos example project, describes its dependency structure, and uses it to begin to describe how TriBITS sets up and manages package and TPL dependencies. The packages in this ReducedMockTrilinos example project are used in the following subsections so one will be constantly referred back to

this subsection. The following subsection TriBITS Dependency Handling Behaviors defines and describes the nitty-gritty details of the TriBITS package and TPL dependency structure and the algorithms that manipulate the various package, TPL, and test enables and disables. Specific examples for the TriBITS dependency handling algorithms are given in the subsection Example Enable/Disable Use Cases. Finally, the subsection <Project>PackageDependencies.xml describes the standard XML output data-structure that gets created by TriBITS that defines a project's package and TPL dependencies.

5.1 Example ReducedMockTrilinos Project Dependency Structure

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

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

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

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

This gives the full list of top-level TriBITS packages:

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

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

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

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

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

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

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

```
TRIBITS_REPOSITORY_DEFINE_TPLS(
  MPI             "${${PROJECT_NAME}_TRIBITS_DIR}/core/std_tpls/"    PT
  BLAS            "${${PROJECT_NAME}_TRIBITS_DIR}/core/std_tpls/"    PT
  LAPACK          "${${PROJECT_NAME}_TRIBITS_DIR}/core/std_tpls/"    PT
  Boost           "${${PROJECT_NAME}_TRIBITS_DIR}/core/std_tpls/"    ST
  UMFPACK         cmake/TPLs/    ST
  AMD             cmake/TPLs/    EX
  PETSC           "${${PROJECT_NAME}_TRIBITS_DIR}/common_tpls/"    ST
  )
```

Take note of the SE Package Test Group (i.e. PT, ST, or EX) assigned to each SE package and TPL as it plays a significant role in how the TriBITS dependency system handles enables and disables.

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

ReducedMockTrilinos Dependencies:

```
Printing package dependencies ...

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

-- RTOp_LIB_REQUIRED_DEP_PACKAGES: Teuchos

-- Epetra_LIB_REQUIRED_DEP_TPLS: BLAS LAPACK
-- Epetra_LIB_OPTIONAL_DEP_TPLS: MPI

-- Triutils_LIB_REQUIRED_DEP_PACKAGES: Epetra

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

-- ThyraCoreLibs_LIB_REQUIRED_DEP_PACKAGES: Teuchos RTOp

-- ThyraGoodStuff_LIB_REQUIRED_DEP_PACKAGES: ThyraCoreLibs

-- ThyraCrazyStuff_LIB_REQUIRED_DEP_PACKAGES: ThyraGoodStuff

-- ThyraEpetra_LIB_REQUIRED_DEP_PACKAGES: Epetra ThyraCoreLibs

-- ThyraEpetraExt_LIB_REQUIRED_DEP_PACKAGES: ThyraEpetra EpetraExt

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

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

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

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

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

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

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

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

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

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

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

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

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

The above optional package-specific cache variables allow one to control whether or not support for upstream dependency X is turned on in package Y independent of whether or not X and Y are themselves both enabled. For

example, if the packages `Triutils` and `EpetraExt` are both enabled, one can explicitly disable support for the optional dependency `Triutils` in `EpetraExt` by setting `EpetraExt_ENABLE_Triutils=OFF`. One may want to do this for several reasons but the bottom line is that this gives the user more detailed control over package dependencies. See the TriBITS Dependency Handling Behaviors and Explicit disable of an optional package dependency for more discussion and examples.

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

5.2 TriBITS Dependency Handling Behaviors

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

In brief, the rules/behaviors of the TriBITS package and TPL dependency management system are:

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

In more detail, these rules/behaviors are:

0) **No circular dependencies of any kind are allowed**: The zeroth rule of the TriBITS dependency management system is that no TriBITS SE package (or its tests) can declare a dependency on a downstream SE package, period! This is one of the most basic Software Engineering Packaging Principles stated as the *ADP (Acyclic Dependencies Principle)*, i.e. "Allow no cycles in the package dependency graph". By default, the TriBITS system will check for circular dependencies and will fail the configure if any are found. For a more detailed discussion of circular package dependencies, see Design Considerations for TriBITS.

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

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

3) **SE package enable triggers auto-enables of upstream dependencies**: Any SE package <TRIBITS_PACKAGE> can be explicitly **enabled** by the user by setting the cache variable ${PROJECT_NAME}_ENABLE_<TRIBITS_PACKAGE>=ON (e.g. Trilinos_ENABLE_EpetraExt=ON). When an SE package is enabled in this way, the TriBITS system will try to enable all of the required upstream SE packages and TPLs defined by the package (specified in its Dependencies.cmake file). If an enabled SE package can't be enabled and has to be disabled, either a warning is printed or processing will stop with an error (depending on the value of ${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES, see Disables trump enables where there is a conflict). In addition, if ${PROJECT_NAME}_ENABLE_ALL_OPTIONAL_PACKAGES=ON, then TriBITS will try to enable all of the specified optional PT SE packages as well (and also optional upstream ST SE packages if ${PROJECT_NAME}_SECONDARY_TESTED_CODE=ON). For an example, see Explicit enable of a package, its tests, an optional TPL, with ST enabled.

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

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

6) **EX TPLs given default unset enable/disable state**: A TriBITS TPL <TRIBITS_TPL> with testing group EX, is given an **unset enable/disable state by default** (i.e. TPL_ENABLE_<TRIBITS_TPL>="", same as for PT and EX TPLs). For an example, see Default configure with no packages enabled on input. This is different behavior than for EX SE packages described above which provides an initial hard disable. However, since TriBITS will never automatically enable an optional TPL (see Optional TPLs only enabled explicitly by the user) and since only downstream EX SE packages are allowed to have a required dependencies on an EX TPL, there is no need to set the default enable for an EX TPL to OFF. If an EX SE package has a required dependency on an EX TPL, just enabling the EX SE package should automatically enable the EX TPL as described in Required TPLs are auto-enabled for enabled SE packages.

7) **Required TPLs are auto-enabled for enabled SE packages**: All TPLs listed as required TPL dependencies for the final set of enabled SE packages are **set to enabled** (i.e. TPL_ENABLE_<TRIBITS_TPL>=ON), unless the listed TPLs are already explicit disabled (in which case the SE package would be disabled or an error would occur, see Disables trump enables where there is a conflict). For example, if the Teuchos package is enabled, then that will trigger the enable of its required TPLs BLAS and LAPACK. For an example, see Explicit enable of a package and its tests.

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

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

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

    - If `${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES=ON`, then TriBITS will disable the explicit enable and continue on. In the above example, TriBITS will override `Trilinos_ENABLE_RTOp=ON` and set `Trilinos_ENABLE_RTOp=OFF` and print a verbose warning to the `cmake` stdout.

    - If `${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES=OFF`, then TriBITS will generate a detailed error message printed to `cmake` stdout and then abort processing. In the above example, TriBITS will report that `RTOp` is enabled but the required SE package `Teuchos` is disabled and therefore `RTOp` can't be enabled and processing must stop.

    For an example of both behaviors, see Conflicting explicit enable and disable.

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

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

    (Note that `Trilinos_ENABLE_ThyraCrazyStuff` is **not** set to `ON` because it is already set to `OFF` by default, see EX SE packages disabled by default.) Likewise, explicitly setting `Trilinos_ENABLE_Thyra=OFF` is equivalent to explicitly setting all of the `Thyra` subpackages to `OFF` at the outset. For a `PT` example, see Explicit enable of a package and its tests. For a `ST` example, see Explicit enable of a package, its tests, an optional TPL, with ST enabled.

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

13) **Support for optional SE package/TPL is enabled by default**: For an SE package <TRIBITS_PACKAGE> with an optional dependency on an upstream SE package or TPL <TRIBITS_DEP_PACKAGE_OR_TPL>, TriBITS will automatically set the intra-enable variable <TRIBITS_PACKAGE>_ENABLE_<TRIBITS_DEP_PACKAGE_OR_TPL>=ON if <TRIBITS_PACKAGE> and <TRIBITS_DEP_PACKAGE_OR_TPL> are **both enabled**. For example, if the packages Triutils and EpetraExt are both enabled, then TriBITS will automatically set EpetraExt_ENABLE_Triutils=ON by default and enables support for Triutils in EpetraExt. Likewise, if Teuchos and the optional TPL Boost are both enabled, then TriBITS will automatically set Teuchos_ENABLE_Boost=ON by default. This is obviously the logical behavior. See the full context for these examples in Explicit enable of a package, its tests, an optional TPL, with ST enabled.

14) **Support for optional SE package/TPL can be explicitly disabled:** Even though TriBITS will automatically set <TRIBITS_PACKAGE>_ENABLE_<TRIBITS_DEP_PACKAGE_OR_TPL>=ON by default if <TRIBITS_PACKAGE> and <TRIBITS_DEP_PACKAGE_OR_TPL> are both enabled at the project level (as described above), the user can explicitly set <TRIBITS_PACKAGE>_ENABLE_<TRIBITS_DEP_PACKAGE_OR_TPL>=OFF which will turn off optional package-level support for the upstream SE package or TPL <TRIBITS_DEP_PACKAGE_OR_TPL> in the downstream SE package <TRIBITS_PACKAGE>. For example, the user can enable EpetraExt and Triutils at the project level, but set EpetraExt_ENABLE_Triutils=OFF which will turn off package-level support for Triutils in the EpetraExt package. Likewise, the user can enable Teuchos, Epetra and the optional TPL Boost, but set Epetra_ENABLE_Boost=ON. This would provide support for Boost in Teuchos but not in Epetra. For examples, see Explicit disable of an optional package dependency and Explicit disable of an optional TPL dependency.

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

16) **ST SE packages only auto-enabled if ST code is enabled**: TriBITS will only enable an optional ST SE package when ${PROJECT_NAME}_ENABLE_ALL_OPTIONAL_PACKAGES=ON if ${PROJECT_NAME}_SECONDARY_TESTED_CODE=ON is also set. Otherwise, when an optional ST upstream dependent SE package is not enabled due to ${PROJECT_NAME}_SECONDARY_TESTED_CODE=OFF, then a one-line warning is printed to stdout. For example, if the EpetraExt package is enabled but ST code is not enabled, then the optional SE package Triutils will not be enabled (for an example, see Explicit enable of a package and its tests). However, when ${PROJECT_NAME}_SECONDARY_TESTED_CODE=ON and EpetraExt is enabled, then Triutils will be enabled too (for an example, see Explicit enable of a package, its tests, an optional TPL, with ST enabled). The TriBITS default is ${PROJECT_NAME}_ENABLE_ALL_OPTIONAL_PACKAGES=ON. This helps to avoid problems when users try to set a permutation of enables/disables which is not regularly tested.

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

18) **<Project>_ENABLE_ALL_PACKAGES enables all PT (cond. ST) SE packages**: Setting the user cache-variable `${PROJECT_NAME}_ENABLE_ALL_PACKAGES=ON` will result in the enable of all `PT` SE packages when `${PROJECT_NAME}_SECONDARY_TESTED_CODE=OFF` and all `PT` and `ST` SE packages when `${PROJECT_NAME}_SECONDARY_TESTED_CODE=ON`. For an example, see Enable all packages. When the project is a meta-project, this will only enable the project's primary meta-project packages (PMPP). That is, a package will only be enabled due to `${PROJECT_NAME}_ENABLE_ALL_PACKAGES=ON` when its parent repository does not have ${REPOSITORY_NAME}_NO_PRIMARY_META_PROJECT_PACKAGES set to `TRUE`. However, if:

    `${REPOSITORY_NAME}_NO_PRIMARY_META_PROJECT_PACKAGES=TRUE`

    then the SE package may be enabled if it (or its parent package) is listed in ${REPOSITORY_NAME}_NO_PRIMARY_META_PROJECT_PACKAGES_EXCEPT.

19) **<Project>_ENABLE_TESTS only enables explicitly enabled SE package tests**: Setting `${PROJECT_NAME}_ENABLE_TESTS=ON` will **only enable tests for explicitly enabled SE packages** on input. For example, configuring with `Trilinos_ENABLE_RTOp=ON` and `Trilinos_ENABLE_TESTS=ON` will only result in the enable of tests for `RTOp`, not `Teuchos` (even through TriBITS will enable `Teuchos` because it is a required dependency of `RTOp`). See an example, see Explicit enable of a package and its tests. When the project is a meta-project, this will only enable tests for the project's primary meta-project packages (PMPP). The uses the same logic as for `${PROJECT_NAME}_ENABLE_ALL_PACKAGES`, see <Project>_ENABLE_ALL_PACKAGES enables all PT (cond. ST) SE packages.

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

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

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

TriBITS prints out a lot of information about the enable/disable logic as it applies the above rules/behaviors. For a large TriBITS project with lots of packages, this can produce a lot of output to stdout. One just needs to understand what TriBITS is printing out and where to look in the output for different information. The examples in the section Example Enable/Disable Use Cases show what this output looks like for the various enable/disable scenarios and tries to explains in more detail the reasons for why the given behavior is implemented the way that it is. Given this output, the rule definitions given above, and the detailed Example Enable/Disable Use Cases, one should always be able to figure out exactly why the final set of enables/disables is the way it is, even in the largest and most complex of TriBITS projects. (NOTE: The same can *not* be said for many other large software configuration and deployment systems where basic decisions about what to enable and disable are hidden from the user and can be very difficult to track down and debug.)

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

`${REPOSITORY_NAME}_NO_IMPLICIT_PACKAGE_ENABLE`

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

`${REPOSITORY_NAME}_NO_IMPLICIT_PACKAGE_ENABLE_EXCEPT`

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

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

5.3 Example Enable/Disable Use Cases

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

The use cases covered are:

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

All of these use cases and more can be easily run from the command-line by first setting:

$ export REDUCED_MOCK_TRILINOS=<base-dir>/tribits/examples/ReducedMockTrilinos

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

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

before each run to clear the CMake cache.

These use cases are now described in detail below.

Default configure with no packages enabled on input

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

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

which produces the relevant dependency-related output:

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

...

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

...

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

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

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

Explicit enable of a package and its tests

One of the most typical use cases is for the user to explicitly enable one or more top-level TriBITS package and enable its tests. This configuration would be used to drive local development on a specific set of packages (i.e. tests do not need to be enabled for packages not being changed).

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

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

which produces the relevant dependency-related output:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
Processing enabled TPL: BLAS
Processing enabled TPL: LAPACK

Configuring individual enabled Trilinos packages ...

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

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

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

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

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

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

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

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

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

Configuring individual enabled Trilinos packages ...

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

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

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

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

Consider the configure of the `ReducedMockTrilinos` project with:

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

which produces the relevant dependency-related output:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

-- Setting Teuchos_ENABLE_Boost=ON since TPL_ENABLE_Boost=ON

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

Getting information for all enabled TPLs ...

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

Configuring individual enabled Trilinos packages ...

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

A few more behaviors of the TriBITS system that this particular configuration use-case shows are described below.

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

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

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

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

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

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

Explicit disable of a package

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

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

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

which produces the relevant dependency-related output:

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

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

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

Configuring individual enabled Trilinos packages ...

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

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

Conflicting explicit enable and disable

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

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

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

which produces the relevant dependency-related output:

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

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

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

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

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

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

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

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

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

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

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

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

Explicit enable of an optional TPL:

ToDo: Set Trilinos_ENABLE_Thyra=ON and TPL_ENABLE_MPI=ON

Explicit disable of an optional TPL:

ToDo: Set Trilinos_ENABLE_Thyra=ON and TPL_ENABLE_MPI=OFF

Explicit disable of a required TPL

ToDo: Set Trilinos_ENABLE_Epetra=ON and Trilinos_ENABLE_BLAS=OFF

Explicit enable of a subpackage

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

Explicit enable of an optional package dependency

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

Explicit disable of an optional package dependency

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

Explicit enable of an optional TPL dependency

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

Explicit disable of an optional TPL dependency

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

Explicit enable of a package and downstream packages and tests

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

Enable all packages

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

Enabling all `PT` packages the with:

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

produces the relevant dependency-related output:

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

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

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

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

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

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

```
Processing enabled TPL: BLAS
Processing enabled TPL: LAPACK

Configuring individual enabled Trilinos packages ...

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

As shown above, only the `PT` SE packages get enabled. To also enable the `ST` packages as well, one additionally set `${PROJECT_NAME}_SECONDARY_TESTED_CODE=ON` at configure time.

5.4 <Project>PackageDependencies.xml

The TriBITS CMake configure system can write out the project's package dependencies into a file `<Project>Dependencies.xml` (or any name one wants to give it). This file is used by a number of the TriBITS SE-related tools. The structure of this XML file, showing one of the more interesting mock packages from the MockTrilinos project is shown below:

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

This XML file contains the names, directories, testing groups (`type`), CDash email address, and all of the SE package and TPL dependencies for every SE package in the TriBITS project. There are several python tools under `tribits/ci_support/` that read in this file and use the created data-structure for various tasks. A TriBITS project configure can create this file as a byproduct (see ???), or the CMake -P script `TribitsDumpDepsXmlScript.cmake` can be used to create this file on the fly without having to configure a TriBITS project (see the `checkin-test.out` log file generated by checkin-test.py for an example of how to run this script to generate the `<Project>Dependencies.xml` file).

6 TriBITS Automated Testing

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

The following subsections describe several aspects to the TriBITS support for testing. The subsection Test Classifications for Repositories, Packages, and Tests defines the different types of test-related classifications that are

defined by TriBITS. These different test classifications are then used to define a number of different standard Nested Layers of TriBITS Project Testing which include different types of CI testing as well as nightly and other tests. One of the most important types of CI testing, pre-push testing, is then described in more detail in the subsection Pre-push Testing using checkin-test.py. The subsection TriBITS Package-by-Package CTest/Dash Driver describes the usage of the advanced TRIBITS_CTEST_DRIVER() function to do incremental project testing of a projects using advanced `ctest -S` scripts. The final subsection TriBITS CDash Customizations describes how projects can use a CDash server to more effectively display test results and provide notifications for failures that are compartmentalized for a large project.

6.1 Test Classifications for Repositories, Packages, and Tests

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

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

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

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

Repository Test Classification

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

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

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

- Repositories marked **Experimental** match the standard CTest dashboard type `Experimental`. Repositories marked as `Experimental` are not processed by default as part of any automated testing described in the subsection Nested Layers of TriBITS Project Testing (except for perhaps some experimental builds). The main reason that an extra repo may be marked as `Experimental` is that it may only contain `EX` SE packages and therefore none of these packages would be enabled by default anyway. Also, a repo may be marked as `Experimental` if it is developed in a very sloppy way such that one cannot even assume that the repository's

<repoDir>/PackagesList.cmake, <repoDir>/TPlsList.cmake, and <packageDir>/cmake/Dependencies.cmake files are not without errors. Since these files are always processed if the repository is included, then any errors in these files will cause the entire configure to fail!

SE Package Test Group

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

Every TriBITS SE Package and TriBITS TPL is assigned a test group. These test groups are for *Primary Tested* (PT) code, *Secondary Tested* (ST) code, and *Experimental* (EX) code. The test group defines *what* SE packages and TPL get selected (or are excluded from being selected) to include in a given build for testing-related purposes. SE packages may also conditionally build in additional code based on the testing group. The detailed rules for when an SE package or TPL are selected or excluded from the build based on the test group is given in TriBITS Dependency Handling Behaviors. We only summarize those rules here.

More detailed descriptions of the test groups are given below.

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

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

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

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

- The top-level TriBITS Package's test group is assigned using the `CLASSIFICATION` field in the macro call TRIBITS_REPOSITORY_DEFINE_PACKAGES() in its parent repository's <repoDir>/PackagesList.cmake file.

- A TriBITS Subpackage's test group is assigned using the `CLASSIFICATIONS` field of the SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS argument in the macro call TRIBITS_PACKAGE_DEFINE_DEPENDENCIES() in its parent package's <packageDir>/cmake/Dependencies.cmake file.

- A TriBITS TPL's test group is assigned using the `CLASSIFICATION` field in the macro call TRIBITS_REPOSITORY_DEFINE_TPLS() in its parent repository's <repoDir>/TPLsList.cmake file.

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

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

In addition to just selecting PT and ST SE packages as a whole, a TriBITS PT SE package can also contain conditional code and test directories that get enabled when ${PROJECT_NAME}_SECONDARY_TESTED_CODE=ON and therefore represents more ST code. The package's <packageDir>/CMakeLists.txt files can contain simple if statements and can use the TRIBITS_SET_ST_FOR_DEV_MODE() function to automatically select extra code to enable when ST is enabled or when the project is in release mode.

Test Test Category

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

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

The different test test categories are described below in more detail:

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

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

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

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

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

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

6.2 Nested Layers of TriBITS Project Testing

Now that the different types of Test Classifications for Repositories, Packages, and Tests have been defined, this section describes how these different test-related classifications are used to select repositories, packages (and code) and tests to run in the standard project testing processes. More than any other section in this document, this section will describe and assume a certain class of software development processes (namely agile processes) where testing and *continuous integration* (CI) are critical components. However, detailed descriptions of these processes are deferred to the later sections Pre-push Testing using checkin-test.py and TriBITS Package-by-Package CTest/Dash Driver.

The standard TriBITS-defined project testing processes are:

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

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

Pre-Push CI Testing

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

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

Typically a TriBITS project will define a "standard development environment" which is comprised of a standard compiler (e.g. GCC 4.6.1), TPL versions (e.g. OpenMPI 1.4.2, Boost 4.9, etc.), and other tools (e.g. cmake 2.8.11, git 1.8.2, etc.). This standard development environment is expected to be used to test changes to the project's code before any push. By using a standard development environment, if the code builds and all the tests pass for the "default" pre-push builds for one developer, then that maximizes the probability that the code will also build and all tests will pass for every other developer using the same development environment. This is critical to keep the development team maximally productive. Portability is also important for most projects but portability testing is best done in a secondary feedback look using Nightly Testing builds. TriBITS has some support for helping to set up a standard software development environment as described in section TriBITS Development Toolset.

The basic assumption of all CI processes (including the one described here) is that if anyone pulls the project's development sources at any time, then all of the code will build and all of the tests will pass for the "default" build cases. For a TriBITS project, this means that the project's `--default-builds` (see above) will all pass for every `PT`

package. All of these software development processes make this basic assumption and agile software development methods fall apart if this is not true.

Post-Push CI Testing

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

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

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

Nightly Testing

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

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

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

Weekly Testing

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

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

Project developer teams should strive to limit the number of test cases that are marked as `WEEKLY` since these tests will

not get run nightly and developers will tend to never enable them when doing more extensive testing using `--st-extra-builds` with the checkin-test.py script in extended pre-push testing.

Performance Testing

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

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

6.3 Pre-push Testing using checkin-test.py

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

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

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

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

configuration = {

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

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

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

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

        }, # cmake

    } # configuration
```

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

Note that both of the default builds shown above, including the `MPI_DEBUG` build, actually set optimized compiler flags with `-DCMAKE_BUILD_TYPE:STRING=RELEASE`. What makes the `MPI_DEBUG` build a "debug" build is turning on optional run-time debug-mode checking, not disabling optimized code. This is important so that the defined tests run fast. For most projects, the default pre-push builds should **not** be used to debug code by running a debugger. Instead, they are designed to test changes to the project's code efficiently before pushing changes. A development team should not have to test the chosen compiler's ability to generate non-optimized debug code and suffer slower test times before pushing.

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

The `checkin-test.py` script is a fairly sophisticated piece of software that is well tested and very robust. The level of testing of this tool is likely greater than any of the software that it will be used to test (unless the project is a real-time flight control system or nuclear reactor control system or something). This is needed so as to provide confidence in the developers that the tool will only push their changes if everything checks out as it should. There are a lot of details and boundary cases that one has to consider and a number of use cases that need to be supported by such a tool. For more detailed documentation, see checkin-test.py --help.

Note that the `checkin-test.py` script can also be used to implement "poor-man's" post-push testing processes as described in Post-Push CI and Nightly Testing using checkin-test.py. However, most software projects will want to go with the more elaborate and more feature-full CTest/CDash system described in TriBITS Package-by-Package CTest/Dash Driver.

6.4 TriBITS Package-by-Package CTest/Dash Driver

The TriBITS system uses a sophisticated and highly customized CTest driver script to test TriBITS projects and submit results to a CDash server. The primary code for driving this is contained in the CTest function

TRIBITS_CTEST_DRIVER() contained in the file `TribitsCTestDriverCore.cmake`. This script loops through all of the specified TriBITS packages for a given TriBITS project and does a configure, built, and test and then submits results to the specified CDash server incrementally. If the configure or library build of any upstream TriBITS package fails, then that TriBITS package is disabled in all downstream TriBITS package builds so as not to propagate already known failures. Each TriBITS top-level package is assigned its own CDash regression email address (see CDash regression email addresses) and each package configure/build/test is given its own row for the build in the CDash server. A CTest script using TRIBITS_CTEST_DRIVER() is run in one of three different modes. First, it can run standard once-a-day, from-scratch builds as described in CTest/CDash Nightly Testing. Second, it can run as a CI server as described in CTest/CDash CI Server. Third, it can run in experimental mode testing a local repository using the TriBITS-defined make dashboard target.

CTest/CDash Nightly Testing

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

CTest/CDash CI Server

When a TriBITS ctest driver script is used in continuous integration (CI) mode, it starts every day with a clean from-scratch build and then performs incremental rebuilds as new commits are pulled from the master branch in the main repository(s). In this mode, a continuous loop is performed after the initial baseline build constantly pulling commits from the master git repository(s). If any package changes are detected (looking at git file diffs), then the tests and examples for those packages and all downstream packages are enabled and run using a reconfigure/rebuild. Since all of the upstream package libraries are already built, this rebuild and retest can take place in a fraction of the time of a complete from-scratch build and test of the project.

6.5 TriBITS CDash Customizations

CDash is not currently designed to accommodate multi-package, multi-repository VC projects in the way they are supported by TriBITS. However, CDash provides some ability to customize a CDash project and submits to address missing features. Each TriBITS package is given a CTest/CDash "Label" with the name of the TriBITS package. CDash will then aggregate the different package configure/build/test runs for each package into aggregated "builds". The commits pulled for each of the extra VC repos listed in the <projectDir>/cmake/ExtraRepositoriesList.cmake file are shown in an uploaded CDash "Notes" file for each TriBITS package configure/build/test submit. This uploaded "Notes" file also contains a cleaned up version of the CMakeCache.txt file as well as a copy of the `ctest -s` script that ran the case.

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

CDash regression email addresses

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

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

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

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

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

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

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

6) **${PROJECT_NAME}_PROJECT_MASTER_EMAIL_ADDRESS** (set in <projectDir>/cmake/ProjectDependenciesSetup.cmake): A single default email address for all packages specified at the Project level (e.g. `my-project-regression@some.project.gov`). If not already set, this variable will be set to `${REPOSITORY_NAME}_REPOSITORY_MASTER_EMAIL_ADDRESS` for the first repository processed that has this set. Every meta-project should set this variable so that it will be the default email address for any new package added.

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

If a TriBITS project does not use CDash, then no email address needs to be assigned to packages at all (and therefore none of the above variables need be set).

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

All of the email dependency management logic must be accessible by just running the macro:

```
TRIBITS_READ_PACKAGES_PROCESS_DEPENDENCIES_WRITE_XML()
```

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

In review, the precedence order for how regression email addresses are selected for a given package is:

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 for a given package.

What the above setup does is it results in the TriBITS system (in the TRIBITS_CTEST_DRIVER() function called in a `ctest -S` script) creating a file called `CDashSubprojectDependencies.xml` (which contains the list of TriBITS packages, which CDash calls "subprojects", and email address for each package to send regression emails to) and that file gets sent to the CDash server. CDash then takes this file and creates, or updates, a set of CDash users (same name and password as the email list address) and sets up a mapping of Labels (which are used for TriBITS package

names) to CDash user emails addresses. CDash is automatically set up to process this XML file and create and update CDash users. There are several consequences of this implementation of which project maintainers need to be aware.

First, **one should not list the email address for a CDash user account already on CDash**. This is because labels will be added for the TriBITS packages that this email address is associated with and CDash emails will not be sent out for any other TriBITS packages, no matter what setting that CDash user account has. Therefore, one should only list email addresses as CDash regression email lists that are not already CDash user accounts and wish to be maintained separately. For example, if there is an email list that one wants to have CDash emails sent to but is already a CDash user account, then one can create another email list (e.g. using Mailman) which can then be registered with the TriBITS packages in the `CDashSubprojectDependencies.xml` file and then that new email list forward email to the target email list.

Second, the CDash implementation currently is not set up to remove labels from existing users when an email address is disassociated with a TriBITS package in the `CDashSubprojectDependencies.xml` file. Therefore, **if one changes a TriBITS package's CDash regression email address then one needs to manually remove the associated labels from the old email address**. CDash will not remove them automatically. Otherwise, email will continue to be sent to that email address for that package.

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

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

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

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

Hopefully that should be enough information to manage the mapping of CDash regression email lists to TriBITS packages for single and multi-repository TriBITS projects.

7 Multi-Repository Support

TriBITS has built-in support for projects involving multiple TriBITS Repositories which contain multiple TriBITS Packages. The basic configuration, build, and test of such projects requires only raw CMake/CTest, just like any other CMake project (see TriBITS System Project Dependencies). Every TriBITS project automatically supports tacking on add-on TriBITS packages and TPLs through the ${PROJECT_NAME}_EXTRA_REPOSITORIES cmake cache variable as described in Enabling extra repositories with add-on packages. In addition, a TriBITS project can be set up to pull in other TriBITS Repositories using the <projectDir>/cmake/ExtraRepositoriesList.cmake file. A special form of this is a TriBITS Meta-Project that contains no native packages or TPLs of its own. The ability to create meta-projects out of individual TriBITS repositories allows TriBITS to be used to provide coordinated builds (or meta-builds) of large aggregations of software.

To help set up a full-featured development environment (i.e. not just the basic configure, build, test, and install) for TriBITS projects with multiple repositories, TriBITS provides some extra development tools implemented using Python which are provided in the "extended" parts of TriBITS (see TriBITS/tribits/ Directory Contents). The primary tools supporting multi-repository projects are the Python scripts clone_extra_repos.py, gitdist, and checkin-test.py.

To demonstrate, consider the TriBITS meta-project with the following `ExtraRepositoriesList.cmake` file:

```
TRIBITS_PROJECT_DEFINE_EXTRA_REPOSITORIES(
  ExtraRepo1  ""  GIT  git@someurl.com:ExtraRepo1  ""  Continuous
  ExtraRepo2  "ExtraRepo1/ExtraRepos2"  GIT  git@someurl.com:ExtraRepo2
                                                NOPACKAGES  Continuous
  ExtraRepo3  ""  GIT  git@someurl.com:ExtraRepo3  ""  Nightly
)
```

Once cloned, the directories would be laid out as:

```
MetaProject/
  .git/
  .gitignore
  ExtraRepo1/
    .git/
    ExrraRepo2/
      .git/
  ExtraRepo3/
    .git/
```

The tool **clone_extra_repos.py** is used to clone the extra repositories for a multi-repositories TriBITS proejct. It reads the repository URLs and destination directories from the file <projectDir>/cmake/ExtraRepositoriesList.cmake and does the clones. For example, to clone all the repos for the `MetaProject` project, one would use the commands:

```
$ git clone git@someurl.com:MetaProject
$ cd MetaProject/
$ ./cmake/tribits/ci_support/clone_extra_repos.py
```

which produces the output like:

```
...

**
** Clone the selected extra repos:
**

Cloning repo ExtraRepo1 ...

Running: git clone git@someurl.com:ExtraRepo1 ExtraRepo1

Cloning repo ExtraRepo2 ...

Running: git clone git@someurl.com:ExtraRepo2 ExtraRepo1/ExtraRepo2

Cloning repo ExtraRepo3 ...

Running: git clone git@someurl.com:ExtraRepo3 ExtraRepo3
```

See clone_extra_repos.py --help for more details.

Once cloned, one needs to work with the multiple repositories to perform basic VC operations. For this, TriBITS provides the tool **gitdist** which is a simple stand-alone Python script that distributes a git command across a set of git repos. This tool is not specific to TriBITS but it is very useful for dealing with TriBITS projects with multiple repositories. It only requires that a base git repo and a set of zero or more git repos cloned under it.

To use `gitdist` with this aggregate meta-project, one would first set up the file `MetaProject/.gitdist` (or a version controlled `MetaProject.gitdist.default` file) which would contain the lines:

```
ExtraRepo1
ExtraRepo1/ExtraRepo2
ExtraRepo3
```

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

```
/ExtraRepo1/
/ExtraRepo1/ExtraRepo2/
/ExtraRepo3/
```

To use `gitdist`, one would put `gitdist` into their path and also set up the command-line shell aliases `gitdist-status` and `gitdist-mod` (see gitdist --help).

Some of the aggregate commands that one would typically run under the base `MetaProject/` directory are:

```
# See status of all repos at once
gitdist-status

# Pull updates to all
gitdist pull

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

The script `gitdist` is provided under TriBITS directory:

```
cmake/tribits/python_utils/gitidst
```

and can be installed by the install_devtools.py script (see TriBITS Development Toolset). See gitdist --help for more details.

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

To keep track of compatible versions of the git repos, TriBITS provides support for a `<Project>RepoVersion.txt` file. Any TriBITS project can generate this file automatically by setting the option ${PROJECT_NAME}_GENERATE_REPO_VERSION_FILE. For the above example `MetaProject`, this file looks like:

```
*** Base Git Repo: MetaProject
e102e27 [Mon Sep 23 11:34:59 2013 -0400] <author0@someurl.com>
First summary message
** Git Repo: ExtraRepo1
b894b9c [Fri Aug 30 09:55:07 2013 -0400] <author1@someurl.com>
Second summary message
** Git Repo: ExtraRepo1/ExtraRepo2
97cf1ac [Thu Dec 1 23:34:06 2012 -0500] <author2someurl.com>
Third summary message
** Git Repo: ExtraRepo3
cd4a3af [Mon Mar 9 19:39:06 2013 -0400] <author3someurl.com>
Fourth summary message
```

This file gets created in the build directory, gets echoed in the configure output, gets installed into the install directory, and get added to the source distributions tarball. It also gets pushed up to CDash for all automated builds. The tool gitdist can then use this file to checkout and tag compatible versions, difference two versions of the meta-project, etc. (see gitdist --help for more details on git operations).

The TriBITS approach to managing multiple VC repos described above works well for around 20 or 30 VC repos but is likely not a good solution for many more git repos. For larger numbers of VC repos, one should consider nested integration creating snapshot git repos (e.g. using the tool snapshot-dir.py) that aggregate several related repositories into a single git repo. Another approach might be to use git submodules. (However, note that the TriBITS tools and processes described here are **not** currently set up to support aggregate VC repos that use git submodules.) The design decision with TriBITS was to explicitly handle the different git VC repos by listing them in the <projectDir>/cmake/ExtraRepositoriesList.cmake file and then using the simple, easy to understand, tools clone_extra_repos.py and gitdist. There are advantages and disadvantages to explicitly handling the different git repos as is currently employed by the TriBITS software development tools verses using git submodules. It is possible that TriBITS will add support for aggregate git repos using git submodules in the future but only if there are important projects that choose to use them. The discussion of these various approaches and strategies to dealing with aggregate repos is beyond the scope of this document.

8 Development Workflows

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

8.1 Basic Development Workflow

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

8.2 Multi-Repository Development Workflow

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

9 Howtos

This section provides short, succinct lists of the steps to accomplish a few common tasks. Extra details are referenced.

9.1 How to add a new TriBITS Package

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

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

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

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

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

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

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

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

9.2 How to add a new TriBITS Package with Subpackages

Adding a new package with subpackages is similar to adding a new regular package described in How to add a new TriBITS Package. Again, it is recommended that one copies an example package from TribitsExampleProject. For example, one could copy files and directories from the example package `WithSubpackages`.

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

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

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

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

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

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

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

9.3 How to add a new TriBITS Subpackage

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

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

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

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

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

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

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

9.4 How to add a new TriBITS TPL

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

To add a new TriBITS TPL, do the following:

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

2) Create the TPL find module, e.g. `<repoDir>/tpls/FindTPL<tplName>.cmake` (see TriBITS TPL and TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES() for details). List the default required header files and/or libraries that must be provided by the TPL. NOTE: This find module can also (optionally) use `FIND_PACKAGE(<tplName> ...)` for the default find operation. For details, see How to use FIND_PACKAGE() for a TriBITS TPL.

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

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

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

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

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

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

7) Use the TPL in the packages that define the dependency on the new TPL, configure, test, etc.

9.5 How to use FIND_PACKAGE() for a TriBITS TPL

When defining a `FindTPL<tplName>.cmake` file, it is possible (and encouraged) to utilize `FIND_PACKAGE(<tplName> ...)` to provide the default find operation. In order for the resulting `FindTPL<tplName>.cmake` to behave consistently between all the various TriBITS TPLs (and allow the standard TriBITS TPL find overrides) one must use the TriBITS function TRIBITS_TPL_ALLOW_PRE_FIND_PACKAGE() in combination with the function TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES(). The basic form of the resulting TriBITS TPL module file `FindTPL<tplName>.cmake` looks like:

```
# First, set up the variables for the (backward-compatible) TriBITS way of
# finding <tplName>.  These are used in case FIND_PACKAGE(<tplName> ...) is
# not called or does not find <tplName>.  Also, these variables need to be
# non-null in order to trigger the right behavior in the function
# TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES().
SET(REQUIRED_HEADERS <header0> <header1> ...)
SET(REQUIRED_LIBS_NAMES <libname0> <libname1> ...)

# Second, search for <tplName> components (if allowed) using the standard
# FIND_PACKAGE(<tplName> ...).
TRIBITS_TPL_ALLOW_PRE_FIND_PACKAGE(<tplName>  <tplName>_ALLOW_PREFIND)
IF (<tplName>_ALLOW_PREFIND)
  MESSAGE("-- Using FIND_PACKAGE(<tplName> ...) ...")
  FIND_PACKAGE(<tplName>)
  IF (<tplName>_FOUND)
    # Tell TriBITS that we found <tplName> and there no need to look any further!
    SET(TPL_<tplName>_INCLUDE_DIRS ${<tplName>_INCLUDE_DIRS} CACHE PATH "...")
    SET(TPL_<tplName>_LIBRARIES ${<tplName>_LIBRARIES} CACHE FILEPATH "...")
    SET(TPL_<tplName>_LIBRARY_DIRS ${<tplName>_LIBRARY_DIRS} CACHE PATH "...")
  ENDIF()
ENDIF()

# Third, call TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES()
TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES( <tplName>
  REQUIRED_HEADERS ${REQUIRED_HEADERS}
  REQUIRED_LIBS_NAMES ${REQUIRED_LIBS_NAMES}
  )
```

```
# NOTE: If FIND_PACKAGE(<tplName> ...) was called and successfully found
# <tplName>, then TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES() will use the
# already-set variables and just print them out.  This is the final "hook"
# into the TriBITS TPL system.
```

With this approach, the `FindTPL<tplName>.cmake` module preserves all of the user behavior described in Enabling support for an optional Third-Party Library (TPL) for overriding what TPL componets to look for, where to look and finally to override what is actually used. That is, if the user sets the cache variables `TPL_<tplName>_INCLUDE_DIRS`, `TPL_<tplName>_LIBRARIES`, or `TPL_<tplName>_LIBRARY_DIRS`, then they should be used without question (which is why the `SET( ... CACHE ...)` calls in the above example do not use `FORCE`).

If one wants to skip and ignore the standard TriBITS TPL override variables `<tplName>_INCLUDE_DIRS`, `<tplName>_LIBRARY_NAMES`, or `<tplName>_LIBRARY_DIRS`, then one can set:

```
SET(<tplName>_FORCE_PRE_FIND_PACKAGE TRUE CACHE BOOL
  "Always first call FIND_PACKAGE(<tplName> ...) unless explicit override")
```

at the top of the `FindTPL<tplName>.cmake` and it will ignore these variables. This avoids name classes with the variables `<tplName>_INCLUDE_DIRS` and `<tplName>_LIBRARY_DIRS` which are often used in the concrete CMake `Find<tplName>.cmake` module files themselves.

For a slightly more complex (but real-life) example, see `FindTPLHDF5.cmake` which is:

```
#######################################################################
# See associated tribits/Copyright.txt file for copyright and license! #
#######################################################################

#
# First, set up the variables for the (backward-compatible) TriBITS way of
# finding HDF5.  These are used in case FIND_PACKAGE(HDF5 ...) is not called
# or does not find HDF5.  Also, these variables need to be non-null in order
# to trigger the right behavior in the function
# TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES().
#

SET(REQUIRED_HEADERS hdf5.h)
SET(REQUIRED_LIBS_NAMES hdf5)

IF (HDF5_REQUIRE_FORTRAN)
  SET(REQUIRED_LIBS_NAMES ${REQUIRED_LIBS_NAMES} hdf5_fortran)
ENDIF()

IF (TPL_ENABLE_MPI)
  SET(REQUIRED_LIBS_NAMES ${REQUIRED_LIBS_NAMES} z)
ENDIF()

#
# Second, search for HDF5 components (if allowed) using the standard
# FIND_PACKAGE(HDF5 ...).
#
TRIBITS_TPL_ALLOW_PRE_FIND_PACKAGE(HDF5  HDF5_ALLOW_PREFIND)
IF (HDF5_ALLOW_PREFIND)

  MESSAGE("-- Using FIND_PACKAGE(HDF5 ...) ...")

  SET(HDF5_COMPNENTS C)
  IF (HDF5_REQUIRE_FORTRAN)
    LIST(APPEND HDF5_COMPNENTS Fortran)
  ENDIF()
```

```
  FIND_PACKAGE(HDF5 COMPONENTS ${HDF5_COMPNENTS})

  IF (HDF5_FOUND)
    # Tell TriBITS that we found HDF5 and there no need to look any further!
    SET(TPL_HDF5_INCLUDE_DIRS ${HDF5_INCLUDE_DIRS} CACHE PATH
      "HDF5 include dirs")
    SET(TPL_HDF5_LIBRARIES ${HDF5_LIBRARIES} CACHE FILEPATH
      "HDF5 libraries")
    SET(TPL_HDF5_LIBRARY_DIRS ${HDF5_LIBRARY_DIRS} CACHE PATH
      "HDF5 library dirs")
  ENDIF()

ENDIF()

#
# Third, call TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES()
#
TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES( HDF5
  REQUIRED_HEADERS ${REQUIRED_HEADERS}
  REQUIRED_LIBS_NAMES ${REQUIRED_LIBS_NAMES}
  )
# NOTE: If FIND_PACKAGE(HDF5 ...) was called and successfully found HDF5, then
# TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES() will use the already-set
# variables TPL_HDF5_INCLUDE_DIRS and TPL_HDF5_LIBRARIES and then print them
# out (and set some other standard variables as well).  This is the final
# "hook" into the TriBITS TPL system.
```

Note that some specialized `Find<tplName>.cmake` modules do more than just return a list of include directories and libraries. Some, like `FindQt4.cmake` also return other variables that are used in downstream packages. therefore, in these cases, `FIND_PACKAGE(Qt4 ...)` must be called on every configure. In specialized cases such as this, one must write a more specialized `FindTPL<tplName>.cmake` file and can't use the TRIBITS_TPL_ALLOW_PRE_FIND_PACKAGE() function like shown shown above. Such find modules cannot completely adhere to the standard behavior described in Enabling support for an optional Third-Party Library (TPL).

9.6 How to add a new TriBITS Repository

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

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

2) Add an ignore for the extra repo name to the base project's git repo's `.gitignore` file (see Multi-Repository Support). Commit this file.

3) Set up the new package dependencies for the new package in the new TriBITS repo or make other adjustments.

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

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

9.7 How to add a new TriBITS SE Package dependency

It is often the case where one will want to add a new dependency for an existing downstream package to an existing upstream TriBITS SE Package. This can either be a required dependency or an optional dependency. Here, we will refer to the downstream SE package as `<packageName>` with base directory `<packageDir>` and will refer to the upstream package as `<upstreamPackageName>`.

The process for adding a new dependency to an existing upstream SE package is as follows:

1) **Add the name of the upstream package to the downstream package's Dependencies.cmake file:** Add `<upstreamPackagename>` to the call of TRIBITS_PACKAGE_DEFINE_DEPENDENCIES() in the downstream package's <packageDir>/cmake/Dependencies.cmake file. If this is to be a required library dependency, then `<upstreamPackagename>` is added to the `LIB_REQUIRED_PACKAGES` argument. Alternatively, if this is to be an optional library dependency, then `<upstreamPackagename>` is added to the `LIB_OPTIONAL_PACKAGES` argument. (For example, see the file `packages/EpetaExt/cmake/Dependencies.cmake` file in the ReducedMockTrilinos project.) If only the test and/or example sources, and not the package's core library sources, will have the required or optional dependency, then `<upstreamPackagename>` is added to the arguments `TEST_REQUIRED_PACKAGES` or `TEST_OPTIONAL_PACKAGES`, respectively.

2) **For an optional dependency, add 'HAVE_' preprocessor macro to the package's configured header file:** If this is an optional dependency, typically a C/C++ processor macro will be added to the package's configured <packageDir>/cmake/<packageName>_config.h.in file using the line:

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

(see HAVE_<PACKAGE_NAME_UC>_<OPTIONAL_DEP_PACKAGE_NAME_UC>.)

Warning, do not add optional defines for tests/examples to configured header files: If this is a test-only and/or example-only dependency then please do **not** add a `#cmakedefine` to the package's core <packageDir>/cmake/<packageName>_config.h.in file. Instead, add the `#cmakedefine` line to a configured header that is only included by sources for the tests/examples or just add a define on the compiler command line (see the `DEFINES` argument to TRIBITS_ADD_LIBRARY() and TRIBITS_ADD_EXECUTABLE(), but see the warning about problems with `ADD_DEFINITIONS()` in Miscellaneous Notes (TRIBITS_ADD_LIBRARY())). We don't want the package's header files to change or libraries to have to be rebuilt if tests/examples get enabled or disabled. Otherwise, the TriBITS Package-by-Package CTest/Dash Driver process will result in unnecessary rebuilds of software over and over again.

3) **Use the features of the upstream package in the source files of the downstream package sources and/or tests/examples:** Usage of the features of the upstream package `<upstreamPackageName>` in the downstream package `<packageName>` will typically involve adding `#include <upstreamPackageName>_<fileName>` in the package's C/C++ source (or test/example) files (or the equivalent in Fortran). If it is an optional dependency, then these includes will typically be protected using preprocessor ifdefs, for example, as:

```
#include "<packageName>_config.h"

#if HAVE_<PACKAGE_NAME_UC>_<OPTIONAL_DEP_PACKAGE_NAME_UC>
#  include "<upstreamPackageName>_<fileName>"
#endif
```

NOTE: TriBITS will automatically add the include directories for the upstream package to the compile lines for the downstream package source builds and will add the libraries for the upstream package to the link lines to the downstream package library and executable links. See documentation in the functions TRIBITS_ADD_LIBRARY() and TRIBITS_ADD_EXECUTABLE(), and the `DEPLIBS` argument to these functions, for more details.

9.8 How to add a new TriBITS TPL dependency

It is often the case where one will want to add a new dependency for an existing downstream package to an existing upstream TriBITS TPL. This can either be a required dependency or an optional dependency. Here, we will refer to the downstream SE package as `<packageName>` with base directory `<packageDir>` and will refer to the upstream TPL as `<tplName>`.

The process for adding a new dependency to an existing upstream TPL is as follows:

1) **Add the name of the upstream TPL to the downstream package's Dependencies.cmake file:** Add `<tplName>` to the call of TRIBITS_PACKAGE_DEFINE_DEPENDENCIES() in the downstream

package's <packageDir>/cmake/Dependencies.cmake file. If this is to be a required library dependency, then `<tplName>` is added to the `LIB_REQUIRED_TPLs` argument. Alternatively, if this is to be an optional library dependency, then `<tplName>` is added to the `LIB_OPTIONAL_TPL` argument. (For example, see the file `packages/Teuchos/cmake/Dependencies.cmake` file in the ReducedMockTrilinos project.) If only the test and/or example sources, and not the package's core library sources, will have the required or optional dependency, then `<tplName>` is added to the arguments `TEST_REQUIRED_TPLs` or `TEST_OPTIONAL_TPLS`, respectively.

2) **For an optional dependency, add 'HAVE_' preprocessor macro to the package's configured header file:** If this is an optional dependency, typically a C/C++ processor macro will be added to the package's configured <packageDir>/cmake/<packageName>_config.h.in file using the line:

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

(see HAVE_<PACKAGE_NAME_UC>_<OPTIONAL_DEP_TPL_NAME_UC>.)

WARNING: If this is a test-only and/or example-only dependency then please do **not** add a `#cmakedefine` to the package's core <packageDir>/cmake/<packageName>_config.h.in file. See Warning, do not add optional defines for tests/examples to configured header files.

3) **Use the features of the upstream TPL in the source files of the downstream package sources and/or tests/examples:** Usage of the features of the upstream package `<tplName>` in the downstream package `<packageName>` will typically involve adding `#include <tplName>_<fileName>` in the package's C/C++ source (or test/example) files (or the equivalent in Fortran). If it is an optional dependency, then these includes will typically be protected using preprocessor ifdefs, for example, as:

```
#include "<packageName>_config.h"

#if HAVE_<PACKAGE_NAME_UC>_<OPTIONAL_DEP_TPL_NAME_UC>
#  include "<upstreamPackageName>_<fileName>"
#endif
```

NOTE: TriBITS will automatically add the include directories for the upstream TPL to the compile lines for the downstream package source builds and will add the libraries for the upstream TPL to the link lines to the downstream package library and executable links. See documentation in the functions TRIBITS_ADD_LIBRARY() and TRIBITS_ADD_EXECUTABLE(), and the `DEPLIBS` argument to these functions, for more details.

9.9 How to insert a package into an upstream repo

Sometimes it is desired to insert a package from a downstream VC repo into an upstream TriBITS Repository in order for one or more packages in the upstream repo to define a dependency on that package. The way this is supported in TriBITS is to just list the inserted package into the `PackagesList.cmake` file of the upstream TriBITS repo after the packages it depends on and before the packages that will use it then call the TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES() function to allow the package to be missing. This is demonstrated in TribitsExampleProject with the package `InsertedPkg` which is **not** included in the default `TribitsExampleProject` source tree. The TribitsExampleProject/`PackagesList.cmake` file looks like:

```
TRIBITS_REPOSITORY_DEFINE_PACKAGES(
  SimpleCxx          packages/simple_cxx         PT
  MixedLang          packages/mixed_lang         PT
  InsertedPkg        InsertedPkg                 ST
  WithSubpackages    packages/with_subpackages   PT
  WrapExternal       packages/wrap_external      ST
  )

TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS(WrapExternal Windows)
TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES(InsertedPkg)
```

In this example, InsertedPkg has a required dependency on on SimpleCxx and the SE package WithSubpackagesB has an optional dependency on InsertedPkg. Therefore, the inserted package InsertedPkg has upstream and downstream dependencies on packages in the TribitsExampleProject repo.

The function TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES() tells TriBITS to treat InsertedPkg the same as any other package if the directory TribitsExampleProject/InsertedPkg exists or to completely ignore the package InsertedPkg otherwise. In addition, TriBITS will automatically disable of all downstream package dependencies for the missing package (and print a note about the disables). NOTE: By default TriBITS will silently ignore missing inserted packages and disable optional support for the missing package. To see what packages are missing and being ignored, configure with:

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

The way one would set up TribitsExampleProject to enable InsertedPkg, if these were in separate VC (e.g. git) repos for example, would be to do:

```
$ git clone <some-url-base>/TribitsExampleProject
$ cd TribitsExampleProject
$ git clone <some-other-url-base>/ExteranlPkg
$ echo /InsertedPkg/ >> .git/info/excludes
```

Then, when you configure TribitsExampleProject, the package InsertedPkg would automatically appear and could then be enabled or disabled like any other TriBITS package. The TriBITS test Tribits_TribitsExampleProject_InsertedPkg demonstrates this.

Assuming that one would put the (new) external package in a separate VC repo, one would perform the following steps:

1) Pick a name for the new inserted external package <insertedPackageName> (see Globally unique TriBITS package names). (NOTE: The external package may already exist in which case the name would already be selected).

2) Create a new VC repo containing the new package <insertedPackageName> or add it to an existing extra VC repo. (Or, add the new package to an existing downstream repo but don't add it to the PackagesList.cmake file to downstream repo. That would define the package twice!)

3) Clone the downstream VC repo under the upstream TriBITS repository. (This is currently needed since TriBITS only allows package dirs to be contained under the repository directory.)

4) Insert the package into the <repoDir>/PackagesList.cmake file as described in How to add a new TriBITS Package except one must also call TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES(<insertedPackageName>) as described above.

5) Flesh out the new package and use it in downstream SE packages just like it was any other package. But note that any downstream SE package that has a required dependency on <insertedPackageName> will always be hard disabled when the source for <insertedPackageName> is missing.

6) When configuring and building to get the package working, add -D<insertedPackageName>_ALLOW_MISSING_EXTERNAL_PACKAGE=FALSE so that TriBITS will catch mistakes in specifying the package directory. Otherwise, to see notes about ignoring missing inserted/external packages, set the variable -D<Project>_WARN_ABOUT_MISSING_EXTERNAL_PACKAGES=TRUE and TriBITS will print warnings about missing external packages.

10 Additional Topics

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

10.1 TriBITS Repository Contents

The TriBITS git repository is organized as a TriBITS Project, TriBITS Repository, and TriBITS Package all in the same base directory. The base contents are described in the file:

```
TriBITS/README.DIRECTORY_CONTENTS.rst
```

The part of TriBITS that is installed or snapshotted in contained in the subdirectory TriBITS/tribits/ and is described in the following section.

TriBITS/ Directory Contents

This base directory for TriBITS acts as a TriBITS Project, a TriBITS Repository, and a TriBITS Package. As such, it contains the standard files that are found in a TriBITS Project, Repository, and Package:

```
ProjectName.cmake    # PROJECT_NAME=TriITS
CMakeLists.txt       # PROJECT_NAME = PACKAGE_NAME = TriBITS
PackagesList.cmake   # Lists just "TriBITS . PT"
TPLsList.cmake       # Lists only MPI
cmake/               # Dependencies.cmake, etc.
```

The core functionality of TriBITS is provided in the directory:

tribits/: The part of TriBITS that CMake projects use to access TriBITS functionality and assimilate the the TriBITS framework. It also contains basic documentation and examples. Files and directories from here are what get installed on the system or are snapshotted into `<projectDir>/cmake/tribits/`. Each TriBITS Project decides what parts of it wants to install or shapshot using the script `tribits/snapshot_tribits.py` (which takes arguments for what dirs to snapshot, see below). This directory contains no tests at all. All of the tests for TriBITS are in the `test/` directory (see below). The breakdown of the contents of `tribits/` are described in the file `tribits/README.DIRECTORY_CONTENTS.rst`.

The following directories are not snapshotted into `<projectDir>/cmake/tribits/` by the script `tribits/snapshot_tribits.py`:

test/: Contains all of the automated tests for TriBITS as part of the TriBITS "TriBITS" package. When doing development, these tests are critical.

dev_testing/: Contains scripts that support the development of the TriBITS system itself in various contexts.

common_tools/: Contains misc utilities that are not central to the TriBITS system but are very helpful to keep around and do not take up too much space.

refactoring/: Some scripts and other files that have aided in various refactorings of TriBITS and are used to upgrade client TriBITS projects.

TriBITS/tribits/ Directory Contents

This directory contains the implementation for the varous parts of TriBITS that are used by TriBITS projects to implement TriBITS functionality. It also contains basic documentation in the subdirectory doc/ that is very close to the TriBITS implementation. Files and directories from here are what get installed on the system or will be snapshotted into `<projectDir>/cmake/tribits/`. Each TriBITS Project decides what parts of TriBITS it wants to install or shapshot using the script `tribits/snapshot_tribits.py` (which takes arguments for what dirs to snapshot). This directory contains no tests at all. All of the tests for TriBITS are in the `test/` directory in the parent TriBITS repository.

The breakdown of the contents of this directory are described below:

TriBITS.cmake: The one file that needs to be included in order to use TriBITS in a CMake project. This one file insulates clients from future TriBITS refactorings of TriBITS.

Version.cmake: Version of TriBITS. This gets included by TriBITS.cmake

core/: Core TriBITS package-based architecture for CMake projects. This only depends on raw CMake and contains just the minimal support for building, testing, installing, and deployment. Only depends on CMake and nothing else.

python_utils/: Some basic Python utilities that are not specific to TriBITS but are used in TriBITS CI and testing support software. There are some very useful python scripts here like `gitdist` and `snapshot-dir.py`.

ci_support/: Support code for pre-push continuous integration testing. This contains the `checkin-test.py` script and its supporting Python modules.

ctest_driver/: Support for package-by-package testing driven by CTest submitting to CDash (to CDash project `<Project>`). This contains the file `TribitsCTestDriveCore.cmake` and some supporting modules.

dashboard_driver/: TriBITS Dashboard Driver system which uses CTest to drive individual project builds in parallel and submits results to a separate `<Project>Driver` CDash project. WARNING: This was written by a contractor and is least well tested, more confusing, and least desirable part of TriBITS. If you have a better way to manage multiple builds (e.g. Jenkins) then use that instead.

common_tpls/: TPLs that are very common and are used by several different TriBITS projects but are not built into the TriBITS system itslef. Having some of these common TPLs in a central location enhances uniformity, reuse, and makes it easier to pull TriBITS packages out of a repo and build them independently.

doc/: Basic TriBITS documentation built using docutils. The generated documents are stored in the git repo but instead are built on command using the `doc/build_docs.sh` script.

examples/: Example TriBITS projects and TPLs. These can be copied out and used as examples

devtools_install/: Basic install scripts for tools like CMake, GCC, OpenMPI, MPICH, Git, etc. By default, these all download tarballs from the `github.com/TriBITSPub/` site and the repos are named `devtools-<toolname>-<version>-base`. This makes it easy to set up a new dev environment for projects that uses TriBITS (or don't use TriBITS for that matter).

win_interface/: Some non-Windows C header files ported to Windows to make porting to Windows easier.

The script `snapshot_tribits.py` install the different pieces for of this `tribits/` directory into a project's `<projectDir>/cmake/tribits/` subdirectory. It supports the argument `--components` with values `core`, `python_utils`, `ci_support`, `ctest_driver`, `dashboard_driver`, `common_tpls`, `doc`, `examples`, `win_interface`, and `devtools_install`. These snapshot components have the dependencies:

- `core` => (external CMake)
- `python_utils` => (external Python 2.4)
- `win_interface` => (external C compiler)
- `TriBITS.cmake` => `core`
- `ci_support` => `core`, `python_utils`
- `ctest_driver` => `core`, `ci_support`
- `dashboard_driver` => `ctest_driver`
- `common_tpls` => `core`
- `examples` => (external tribits installation)
- `doc` => `core`, `ci_support`, `examples`
- `devtools_install` => `python_utils`

10.2 TriBITS System Project Dependencies

The core TriBITS system itself (see `tribts/core/` in TriBITS/tribits/) which is used to configure, built, test, create tarballs, and install software has no dependencies other than a basic installation of CMake (which typically includes the executables `cmake`, `ctest`, and `cpack`). Great effort has been expended to implement all of this core functionality of TriBITS just using raw CMake. That means that anyone who needs to configure, build, and install software that uses TriBITS just needs a compatible CMake implementation. CMake is becoming iniquitous enough that many machines will already have a current-enough version of CMake installed by default on their systems and therefore no one will need to download or install any extra software when building and installing a project that uses TriBITS (assuming the necessary compilers etc. required by the project are also installed). If a current-enough version of CMake is not installed on a given system, it is easy to download the source code and all it needs is a basic C++ compiler to build and install.

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

While the TriBITS Core functionality to configure, build, test, and install software is written using only raw CMake, the more sophisticated development tools needed to implement the full TriBITS development environment require Python 2.4 (or higher, but not Python 3.x) (see Python Support). Python is needed for tools like checkin-test.py and gitdist. In addition, these python tools are used in TRIBITS_CTEST_DRIVER() to drive automated testing and submits to CDash. Also note that `git` is the chosen version control tool for the TriBITS software development tools and all the VC-related functionality in TriBITS. But none of this is required for doing the most basic building, testing, or installation of a project using TriBITS Core.

10.3 Python Support

TriBITS Core does not require anything other than raw CMake. However, Python Utils, TriBITS CI Support, and other extended TriBITS components require Python. These extra TriBITS tools only require Python 2.4+. By default, when a TriBITS project starts to configure using CMake, it will try to find Python 2.4+ on the system (see Full Processing of TriBITS Project Files). If Python is found, it will set the global cache variable `PYTHON_EXECUTABLE`. If it is not found, then it will print a warning and `PYTHON_EXECUTABLE` will be empty. With this default behavior, if Python is found, then the TriBITS project can use it. Otherwise, it can do without it.

While the default behavior for finding Python described above is useful for many TriBITS project (such as Trilinos), some TriBITS projects need different behavior such as:

1. The TriBITS project may not ever use Python so there is no need to look for it at all. In this case, the TriBITS project would set ${PROJECT_NAME}_USES_PYTHON to `FALSE`.
2. Some TriBITS projects require Python and should not even configure if it can't be found. In this case, the TriBITS project would set ${PROJECT_NAME}_REQUIRES_PYTHON to `TRUE`.
3. Some TriBITS projects may require a version of Python more recent than 2.4. In this case, the TriBITS project would set PythonInterp_FIND_VERSION to some value higer than `2.4`. For example, may newer systems have Python 2.6.6 or higher versions installed by default and projects developed on such a system typically requires this version or higher.

10.4 Project-Specific Build Reference

If a project that uses TriBITS is going to have a significant 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 reference document in restructured text (RST) format and with HTML and LaTeX/PDF outputs. This document are generally created in the base project source tree and given then name `<Project>BuildReference.[rst,html,pdf]`. This document consists of two parts. One part is a generic template document:

```
tribits/doc/TribitsBuildReferenceBody.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:

```
<projectDir>/cmake/<Project>BuildReferenceTemplate.rst
```

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

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

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

```
<projectDir>/
  <Project>BuildReference.rst
  <Project>BuildReference.html
  <Project>BuildReference.pdf
```

For a simple example of this, see:

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

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

10.5 Project and Repository Versioning and Release Mode

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

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

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

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

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

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

10.6 TriBITS Environment Probing and Setup

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

Probe and set up the environment:

- Set `CMAKE_BUILD_TYPE`
- Set up for MPI (MPI compilers, etc.)
- Set up C, C++, and Fortran compilers
- Find Perl (sets `PERL_EXECUTABLE`)
- Determine mixed language C/Fortran linking
- Set up C++11, OpenMP, and Windows support

- Find Doxygen (sets `DOXYGEN_EXECUTABLE`)
- Perform some other configure-time tests (see `cmake` output)

At the completion of this part of the processing, the TriBITS CMake project is ready to compile code. All of the major variables set as part of this process are printed to the `cmake` stdout.

10.7 Configure-time System Tests

CMake has good support for defining configure-time checks of the system to help in configuring the project. One can check for whether a header file exists, if the compiler supports a given data-type or language feature, or perform almost any other type of check that one can imagine that can be done using the configured compilers, libraries, system tools, etc. An example was given in TribitsExampleProject. Just follow that example, look at some of the built-in CMake configure-time test modules, and consult provided on-line CMake documentation in order to learn how to create a configure-time test for almost anything.

10.8 Creating Source Distributions

The TriBITS system uses CMake's built-in CPack support to create source distributions in a variety of zipped and tarred formats. (Note that the term "source tarball" or just "tarball" may be used below but should be interpreted as "source distribution".) TriBITS will automatically add support for CPack when the variable ${PROJECT_NAME}_ENABLE_CPACK_PACKAGING is set to `ON`. The commands for creating a source distribution are described in Creating a tarball of the source tree using the built-in `package_source` build target. The value added by TriBITS is that TriBITS will automatically exclude the source for any defined packages that are not enabled and TriBITS provides a framework for systematically excluding files and directories from individual repositories and packages. In addition, the source for non-enabled subpackages can also be excluded depending on the value of ${PROJECT_NAME}_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION. All of this allows one to create distributions which only includes subsets of a larger project (even a single package in some cases).

Unlike other build systems (like autotools), CMake will put **EVERYTHING** into the source distribution (e.g. tarball) that it sitting in the source tree by default. Therefore, setting up for a source distribution usually means deciding what extra files and directories should be excluded. Beyond the directories for non-enabled packages, further files can be selected to be excluded on a package-by-package based and at the repository level (see below).

Individual packages can list additional files/directories under the package's source tree to be excluded from the project's source distribution using a call to TRIBITS_EXCLUDE_FILES() in their <packageDir>/CMakeLists.txt file. Note that if the package is not enabled, these excludes will never get added! That is no problem if these excludes only apply to the given package since TriBITS will add an exclude for the entire package but is a problem if a package lists excludes for files outside of the package's source tree.

Additional files and entire directories can also be excluded at the repository level by listing them in the <repoDir>/cmake/CallbackDefineRepositoryPackaging.cmake file which typically just appends the built-in CMake variable `CPACK_SOURCE_IGNORE_FILES`. However, if a repository is not processed, this file is never processed and therefore no files will be excluded from a repository sitting in the source tree that is not processed (see below).

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

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

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

There are a few points of caution to note about creating source distributions.

NOTE: It is worth stressing again that **EVERY** file that is in the source tree will be included in the source distribution (tarball) unless there is an exclude regex matching it appended to the variable `CPACK_SOURCE_IGNORE_FILES`. TriBITS can only add excludes for defined non-enabled packaged. Every other file listed in the source tree will be included in the tarball.

NOTE: The entries in `CPACK_SOURCE_IGNORE_FILES` are interpreted as **REGULAR EXPRESSIONS** not globs so if you add `"someFile.*"` as an exclude, it will exclude every file in the entire source tree that has `"someFile"` in the name! This is because, in regex terminology, the trailing `".*"` means "match any character zero or more times"

and "someFile" can match anywhere in the file name path. Also, note that if you add in an exclude like "*.pyc" (i.e. trying to exclude all of the generated Python bite code files) that it will exclude every file that has "pyc" in the name and **not** just those with the file extension "pyc". For example, the exclude ".pyc" would exclude the files "puppyc", "lpycso", etc. If you want to exclude all files with extension "pyc", you have to add the exclude regex ".*[.]pyc$"! One's lack of understanding of this fact will cost someone hours of lost time debugging what happens when random files are missing when one tries to configure what is left. Somethings, what is left will actually configure and might almost build!

NOTE: As warned in TriBITS Package Core Files and TriBITS Subpackage Core Files, SE Packages must have directories that are strictly independent of the directories of other SE packages. If they don't, then the source directory for an enabled package will get excluded from the source distribution if its directory is under the directory of a package that is not enabled. For example, if PackageA is enabled but its package directory packageb/packagea/ is under the package directory packageb/ for the disabled package PackageB, then every file and directory under packageb/ will be excluded from the source distribution (tarball), including everything under packageb/packagea/! It would be too expensive to put in an automated check for cases like this so package developers should just take care not to nest the directories of packages inside of each other to avoid problems like this.

NOTE: Extra repositories that are sitting in the source tree but not processed by TriBITS for some reason (e.g. due to explicitly listing in the variable ${PROJECT_NAME}_EXTRA_REPOSITORIES only a subset of the repositories listed in <projectDir>/cmake/ExtraRepositoriesList.cmake) will get added to the source distribution in full by default even though there are no enabled packages from these repos. These non-processed repo dirs are like any other random directory sitting in the source tree, they will get copied over into the source distribution!.

NOTE: When debugging tarball creation problems, always configure with the variable <Project>_DUMP_CPACK_SOURCE_IGNORE_FILES=ON. If you don't see a regex listed for the file or directory you expect to be excluded, then that file/directory it will be included in the source distribution!

10.9 Multi-Repository Almost Continuous Integration

The checkin-test.py script can be used to the implement staged integration of the various repositories in a multi-repo TriBITS project (see Multi-Repository Support) . This is referred to here as Almost Continuous Integration (ACI). The basic concept of Almost Continuous Integration (ACI) is defined and described in the paper [Integration Strategies for CSE, 2009].

This topic is broken down into the following subsections:

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

ACI Introduction

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

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

The following subsections describe how to use the checkin-test.py script to implement an ACI process for a given set of git/TriBITS repositories and also provides a little background and context behind ACI.

ACI Multi-Git/TriBITS Repo Integration Example

In order to set up the context for the ACI process, consider the following simple TriBITS project with two extra repositories:

```
BaseProj/
    ExtraRepo1
    ExtraRepo2
```

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

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

  `url1.gov:/git/BaseProj`

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

  `url2.gov:/git/ExtraRepo1`

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

  `url3.gov:/git/ExtraRepo2`

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

Integration project mirrored git repos:

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

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

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

ACI Local Sync Git Repo Setup

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

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

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

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

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

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

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

This gives the remotes:

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

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

ACI Integration Build Directory Setup

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

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

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

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

ACI Sync Driver Script

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

```
#!/bin/bash -e

EXTRA_ARGS=$@

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

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

cd $SYNC_BASE_DIR/BUILDS/CHECKIN

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

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

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

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

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

`--extra-pull-from=integrate-from:master`

> This option instructs the `checkin-test.py` script to pull and merge in commits that define the integration. This same remote name and remote branch name has to be the same in all git repos (todo: remove this requirement). If it is not, then this option can't be used and instead the wrapper script should do the pulls up front manually before calling the `checkin-test.py` script. The disadvantage of doing the pulls manually is that if they fail for some reason, they will not be seen by the `checkin-test.py` script and no notification email would go out. However, integrating `master` branches in different git repos is a very common use case when good Lean/Agile CI practices are used by all of the projects.

`--abort-gracefully-if-no-changes-to-push`

> The option `--abort-gracefully-if-no-changes-to-push` makes the `checkin-test.py` script gracefully terminate without sending out any emails if after all the pulls, there are no local changes to push to the 'origin' repos. This can happen, for example, if no commits were pushed to the main development git repo for `ExtraRepo1` at `url2.gov:/git/ExtraRepo1` since the last time this sync process was run. This avoids getting confusing and annoying emails like `"PUSH FAILED"`. The reason this option is not generally needed for local developer usage of the `checkin-test.py` script is that in general a developer will not run the `checkin-test.py` script with `--push` unless they have made

local changes; it just does not make any sense at all to do that and if they do by accident, they should get an error email. However, for an automated ACI sync process, there is no easy way to know a priori if changes need to be synced so the script supports this option to deal with that case gracefully.

`--send-email-to=base-proj-integrators@url4.gov`

The results of the builds will be sent this email address. If you only want an email sent when a push actually happens, you can set `--send-email-to=''` and rely on `--send-email-to-on-push`.

`--send-email-to-on-push=base-proj-integrators@url4.gov`

A confirmation and summary email will be sent to this address if the push happens. This can be a different email address than set by the `--send-email-to` option. It is highly recommended that a mail list be used for this email address since this will be the only more permanent logging of the ACI process.

`--no-append-test-results --no-rebase`

These options are needed to stop the `checkin-test.py` script from modifying the commits being tested and pushed from one public git repo to another. The option `--no-append-test-results` is needed to instruct the `checkin-test.py` script to *NOT* amend the last commit with the test results. The option `--no-rebase` is needed to avoid rebasing the new commits pulled. While the default behavior of the `checkin-test.py` script is to amend the last commit message and rebase the local commits (which is considered best practice when testing local commits), this is a very bad thing to do when a ACI sync server is only testing and moving commits between public repos. Amending the last commit would change the SHA1 of the commit (just as a rebase would) and would fork the history and mess up a number of workflows that otherwise should work smoothly. Since an email logging what was tested will go out if a push happens due to the `--send-email-to-on-push` argument, there is no value in appending the test results to the last commit pulled and merged (which will generally not be a merge commit but just a fast-forward). There are cases, however, where appending the test results in an ACI process might be acceptable but they are not discussed here.

`--do-all --push -j16`

These are standard options that always get used when invoking the `checkin-test.py` script and need no further explanation.

`--wipe-clean`

This option is added if you want to make the sync server more robust to changes that might require a clean configure from script. If you care more about using less computer resources and testing that rebuilds work smoothly, remove this option.

Note that the option `--enable-packages` is not set in the above invocation of the `checkin-test.py` script, and therefore the `checkin-test.py` script will decide on its own what packages to test just based on what packages have changed files in the `ExtraRepo1` git/TriBITS extra repository. This is the preferred way to go since any affected packages will automatically be enabled as determined by the TriBITS package dependency structure and therefore this driver script will require no modifications if the dependency structure changes over time. However, there are cases and various reasons where the exact list of packages to be tested (or not tested) should be specified using the `--enable-packages` option (or the `--disable-packages` option, respectively). However, these should not be needed with well structured portable TriBITS repos and packages.

The sync script can be created and tested locally to ensure that it works correctly first, before setting it as a cron job as described next.

Note, if using this in a continuous sync server that runs many times in a day in a loop, you also want to set the option `--abort-gracefully-if-no-updates` in addition to the option `--abort-gracefully-if-no-changes-to-push`. That is because if the updated repos are in a broken state such that there are always local changes at every CI iteration (because they have not been pushed to origin), you don't want to do a new CI build unless something has changed that would otherwise perhaps make the error go away. That allows the CI server to sit ready to try out any change that gets pulled that might allow the integrated build to work and then push the updates.

ACI Cron Job Setup

Once the sync script `sync_ExtraRepo1.sh` has been locally tested, then it should be committed to a version control git repo and then run automatically as a cron job. For example, the cron script shown below would fire off the daily ACI process at 8pm local time every night:

```
# ----------------- minute (0 - 59)
# |  -------------- hour (0 - 23)
# |  |  ----------- day of month (1 - 31)
# |  |  |  -------- month (1 - 12)
# |  |  |  |  ----- day of week (0 - 7) (Sunday=0 or 7)
# |  |  |  |  |
# *  *  *  *  *  command to be executed
 00 20  *  *  *  ~/sync_base_dir/sync_ExtraRepo1.sh &> ~/sync_base_dir/sync_ExtraRepo1
```

In the above crontab file (set with `'crontab -e'` or `'crontab my-crontab-file.txt'`), the script:

```
~/sync_base_dir/sync_ExtraRepo1.sh
```

is assumed to be a soft symbolic link to some version controlled copy of the ACI sync script. For example, it might make sense for this script to be version controlled in the `BaseProj` repo and therefore the symbolic link would be created with something like:

```
$ cd ~/sync_base_dir/
$ ln -s BaseProj/sampleScripts/sync_ExtraRepo1.sh .
```

Such a setup would ensure that sync scripts would always be up-to-date due to the git pulls part of the ACI process.

Addressing ACI Failures and Summary

After the above cron job starts running (setup described above), the `checkin-test.py` script will send out emails to the email addresses passed into the underlying `checkin-test.py` script. If the emails report an update, configure, build, or test failure, then someone will need to log onto the machine where the ACI sync server is running and investigate what went wrong, just like they would if the were running the `checkin-test.py` script for testing locally modified changes before pushing.

In the above example, only a single git/TriBITS repo is integrated in this ACI sync scripts. For a complete system, other ACI sync scripts would be written to sync the two other git/TriBITS repos in order to maintain some independence. Or, a single ACI sync script that tries to update all three git/TriBITS repos at could would be written and used. The pattern of integrations chosen will depend many different factors and these patterns can change over time according to current needs and circumstances.

In summary, the checkin-test.py script can be used to set up robust and effective Almost Continuous Integration (ACI) sync servers that can be used to integrate large collections of software in logical configurations at any logical frequency. Such an approach, together with the practice of Regulated Backward Compatibility and Deprecated Code, can allow very large collections of software to be kept integrated in short time intervals using ACI.

10.10 Post-Push CI and Nightly Testing using checkin-test.py

While the post-push CI and Nightly testing processes using `ctest -S` scripts using TRIBITS_CTEST_DRIVER() (see TriBITS Package-by-Package CTest/Dash Driver) which posts results to a CDash server (see TriBITS CDash Customizations) is a very attractive testing system with many advantages, setting up a CDash server can be a bit difficult and a CDash server can require non-trivial storage and CPU resources (due to the MySQL DB of test results) and requires some amount of extra maintenance. As an intermediate approach, one can consider just using the project's checkin-test.py script to implement basic post-push CI and/or Nightly testing servers using simple cron jobs and a some other helper scripts. The `checkin-test.py` script will robustly pull new commits, configure the project, build, run tests, and send out emails with results and pass/fail. A bunch of builds can be run at once using multiple builds specified in the `--default-builds`, `--st-extra-builds`, and `--extra-build` arguments, or different invocations of the `checkin-test.py` script can be run in parallel for better machine utilization.

What one gives up with this approach over the full-blow CTest/CDash implementation is:

1) Test results over multiple builds/days are not archived in a database for later retrieval and query. Only the latest build's detailed results are accessible.

2) Detailed test results (pass or fail) cannot be easily looked up on a website. Only detailed results from the last build can be found and one has to go to the machine where the build was run from to see the results.

The process for setting up a basic nightly build using `checkin-test.py` as a cron job is a subset of the steps needed to set up a driver and cron job for the more complex ACI process described in ACI Sync Driver Script and ACI Cron Job Setup. Setting up a CI server using `checkin-test.py` is a little more involved than setting up a nightly build (and less desirable because it blows away old CI iteration results pretty fast) but can be done using existing tools.

For smaller, private, less-critical projects with few developers, setting up CI and Nightly testing using `checkin-test.py` may be quite adequate. In fact, post-push testing processes implemented with `checkin-test.py` are much more solid and feature-full than have been employed in many software projects that we have seen over the years that were larger, more public, had many developers, and and were quite important to many users and development teams.

10.11 TriBITS Dashboard Driver

TriBITS also includes a system based on CMake/CTest/CDash to drive the builds of a TriBITS project and post results to another CDash project. This system is contained under the directory:

```
tribits/ctest/tdd/
```

If the TriBITS project name is `<projectName>`, the TDD driver CDash project is typically called `<projectName>Driver`. Using CTest to drive the Nightly builds of a TriBITS project makes sense because CTest can run different builds in parallel, can time-out builds that are taking too long, and will report results to a dashboard and submit notification emails when things fail. However, this is the most confusing and immature part of the TriBITS system. The TriBITS Package-by-Package CTest/Dash Driver system using the TRIBITS_CTEST_DRIVER() can be used without this TriBITS Dashboard Driver (TDD) system.

10.12 Regulated Backward Compatibility and Deprecated Code

The motivation and ideas behind *Regulated Backward Compatibility* and deprecated code are described in the TriBITS Lifecycle Model document. Here, the details of the implementation in TriBITS are given and how transitions between non-backward compatible major versions is accomplished.

This section describes the process for deprecating code within a major backward-compatible version sequence and finally removing deprecated code when transitioning to a new major version X to X+1 for the semantic versioning numbering scheme X.Y.Z. This information is given as a process with different phases.

The processing for managing deprecated code is as follows and more details are given below:

- Setting up support for deprecated code handling
- Deprecating code but maintaining backward compatibility
  - Deprecating C/C++ classes, structs, functions, typdefs, etc.
  - Deprecating preprocessor macros
  - Deprecating an entire header and/or source file
- Hiding deprecated code to certify and facilitate later removal
  - Hiding C/C++ entities
  - Hiding entire deprecated header and source files
- Physically removing deprecated code
  - Removing entire deprecated header and source files
  - Removing deprecated code from remaining files

Setting up support for deprecated code handling

Setting up support for managing deprecated code in a TriBITS package requires just two simple changes to the TriBITS-related files in a package. First, the top-level package <packageDir>/CMakeLists.txt file needs to have a call to:

```
TRIBITS_ADD_SHOW_DEPRECATED_WARNINGS_OPTION()
```

Second, the package's configure header:

```
<packageDir>/cmake/<PACKAGE_UCNAME>_config.h.in
```

file needs to have:

```
@<PACKAGE_UCNAME>_DEPRECATED_DECLARATIONS@
```

where `<PACKAGE_UCNAME>` is the upper-case name of the TriBITS package.

That is all that is needed to provide support for TriBITS deprecated code.

When a TriBITS project is configured with:

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

by default, all packages will show deprecated warnings. These deprecated warnings can be turned on and off on a package-by-package basis by setting:

```
-D<PackageName>_SHOW_DEPRECATED_WARNINGS=[ON|OFF]
```

This gives developers and users a little extra control over what deprecated warnings are shown.

In addition, deprecated code can be hidden from the build to help certify that downstream code is clean by configuring with:

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

This will remove the deprecated code from the build and install (see details below) so that other software in the TriBITS project can be shown to build clean without deprecated code and so that outside client code can be shown to be clean of deprecated code.

As with deprecated warnings, showing or hiding deprecated code can be controlled on a package-by-package basis by setting:

```
-D<PackageName>_HIDE_DEPRECATED_CODE=[ON|OFF]
```

In this case, hiding deprecated code on a package-by-package basis may not work because deprecated code in a downstream package might rely on deprecated code in an upstream package (which might have its deprecated code hidden).

Deprecating code but maintaining backward compatibility

One of the most important aspects of the TriBITS Lifecycle Model for later-stage Production Growth and Production Maintenance code is to provide backward compatibility between a continuous stream of versions of the software within a major version number X in the version numbering scheme X.Y.Z. In all cases, if a piece of client code builds and runs correctly against version X0.Y0.Z0, it should also build, without modification, against versions X0.Y1,Z1 for all Y1 >= Y0 and all Z1 and up to (but not including) X0+1.0.0. There are many types of constructs that one will want to deprecate and therefore later remove. When deprecating code, one wants to give users compile-time warnings of the usage of deprecated features so that they know what they need to remove. One also wants to allow them to certify that their code is free of deprecated warnings by hiding deprecated code. Below, the different types of entities that one wants to deprecate and how to support hiding code (which also facilitates removing it later) are described.

Deprecating C/C++ classes, structs, functions, typdefs, etc. To deprecate standard C/C++ constructs, one can just use the standard TriBITS compile-time macro `<PACKAGE_UCNAME>_DEPRECATED` which is properly ifdefed by the TriBITS system to add a GCC/Intel deprecated attribute or not. For example, one would deprecate standard C/C++ constructs for the package `SomePackage` with:

```
// Deprecate a class (or struct)
class SOMEPACKAGE_DEPRECATED SomeClass { ... };

// Deprecate a function
SOMEPACKAGE_DEPRECATED int someFunc(...);

// Deprecate a typedef
SOMEPACKAGE_DEPRECATED typedef someTypeDef int;
```

The GCC (version 3.1 and newer) and Intel C and C++ compilers both support adding extra attributes including the `__deprecated__` attribute. When this attribute is applied to a given C/C++ entity, it produces a compiler warning that can be searched for in the compiler output and elevated to an error (when `-Werror` is also passed to the compiler).

In addition to the basic deprecated warning, one can also add an optional deprecated warning message using the macro `<PACKAGE_UCNAME>_DEPRECATED_MSG()`. For example, if a new function is replacing an old function, one might use:

```
// Deprecated old unsafe function taking raw pointers
SOMEPACKAGE_DEPRECATED_MSG(
  "Please use the safe someFunc(const Ptr<const std::string>&) instead!")
void someFunc(const std::string *str);

// New version take does not take raw pointers (and is therefore safer)
void someFunc(const Teuchos::Ptr<const std::string> &str);
```

Then, if user code calls the version `someFunc(const std::string*)` they will get the string:

```
"Please use the safe someFunc(const Ptr<const std::string>&) instead!"
```

printed as well by the compiler. Note that the custom message will only be printed for GCC versions 4.5 and newer. If an older version of GCC is used, then the message string is ignored.

Deprecating preprocessor macros A C/C++ preprocessor macro is not an entity seen by the C/C++ compiler and therefore cannot directly take advantage of a feature such as the `__deprecated__` attribute of the GCC/Intel compilers. However, in some cases, for function-like macros can such as:

```
// The definition of the macro
#define SOME_OLD_FUNC_MACRO(ARG1, ARG2, ...) ...
...
// A use of the macro
SOME_OLD_FUNC_MACRO(a, b, ...)
```

there is a strategy where one can define the macro to call a dummy deprecated function such as with:

```
SOMEPACKAGE_DEPRECATED
inline void SOME_OLD_FUNC_MACRO_is_deprecated()
{}

// The definition of the macro
#define SOME_OLD_FUNC_MACRO(ARG1, ARG2, ...) \
  { \
    SOME_OLD_FUNC_MACRO_is_deprecated(); \
    ... \
  }
```

```
...

// A use of the macro
SOME_OLD_FUNC_MACRO(a, b, ...)
```

In the above example, client code calling `SOME_OLD_FUNC_MACRO()` will now result in a deprecated compiler warning which should make it clear what is being deprecated.

Note that this approach will not work in cases were a macro is only providing a single value such as a constant (but one should not use macros for providing constants anyway).

Deprecating an entire header and/or source file There are times when one wants to deprecate an entire set of files and all of the contents in those files. In addition to deprecating the contents of the files one will want to deprecate the entire file as well. There are a few steps to this. First, one wants to put a warning in the file such as:

```
#ifdef __GNUC__
#  warning "This header <THIS_HEADER> is deprecated!  Use <NEW_HEADER> instead!"
#endif
```

The above `ifdef` on `__GNUC__` is needed in order avoid the non-standard `#warning` preprocessor directive on non-compliant compilers (but should work on all later version GCC and Intel compilers).

Hiding deprecated code to certify and facilitate later removal

In addition to adding deprecation warnings at preprecessing or compile-time, it is also highly desirable to allow the deprecated code to be removed from the build to help certify that client code indeed no longer needs the deprecated code. The following subsections describe how to hide deprecated code from existing files and how to hide deprecated files entirely.

Hiding C/C++ entities In the case when various C/C++ entities will be removed from an existing file, but the file will remain, then the deprecated code can be ifdefed out, for the package `SomePackage` for example, using:

```
#ifndef SOMEPACKAGE_HIDE_DEPRECATED_CODE
  // Deprecate a class (or struct)
  class SOMEPACKAGE_DEPRECATED SomeClass { ... };
  ...
#endif /* SOMEPACKAGE_HIDE_DEPRECATED_CODE */
```

In this way, when the CMake variable `SomePackae_HIDE_DEPRECATED_CODE=ON`, then the deprecated code will be completely removed resulting in compile errors for any downstream code still using them.

Hiding entire deprecated header and source files In order to hide entire deprecated header and source files when the CMake variable `<PackageName>_HIDE_DEPRECATED_CODE=ON` is set, one needs to move the headers and sources to another directory and provide for conditional inclusion in the TriBITS build of the library. For example, suppose one wants to remove support for the deprecated files `SomeOldStuff.hpp` and `SomeOldStuff.cpp`. In this case, one would move the files onto a new `deprecated/` sub-directory and then write the `CMakeLists.txt` file like:

```
SET(HEADERS "")
SET(SOURCES "")

SET_AND_INC_DIRS(DIR ${CMAKE_CURRENT_SOURCE_DIR})
APPEND_GLOB(HEADERS ${DIR}/*.hpp)
APPEND_GLOB(SOURCES ${DIR}/*.cpp)

IF (NOT ${PACKAGE_NAME}_HIDE_DEPRECATED_CODE)
  INCLUDE_DIRECTORIES(${CMAKE_CURRENT_SOURCE_DIR}/deprecated)
  APPEND_SET(HEADERS
    SomeOldStuff.hpp
```

```
    )
  APPEND_SET(SOURCES
    SomeOldStuff.cpp
    )
ENDIF()

...

TRIBITS_ADD_LIBRARY(
  <LIBRARY_NAME>
  HEADERS ${HEADERS}
  SOURCES ${SOURCES}
  )
```

In this way, when `${PACKAGE_NAME}_HIDE_DEPRECATED_CODE=TRUE`, then the directory for the deprecated headers will not be in the include path (so downstream clients will not even be able to see them) and they will not be installed so external clients will not be able to see them either. However, when `${PACKAGE_NAME}_HIDE_DEPRECATED_CODE=FALSE`, then these files will be included in the build and include path and downstream clients can use them.

Once these files need to be permanently removed, one just then needs to remove them from the version control repository (i.e. `git rm <files_to_remove>`) and them remove them from the above `CMakeLists.txt` code.

Physically removing deprecated code

The final step in the code deprecation cycle is to actually remove the deprecated code. This is necessary to clean house, remove clutter and finally get the payoff in the reduction of technical debt that occurs when removing what is no longer needed or wanted.

It is recommended to remove deprecated files first, then remove deprecated file fragments from remaining files. Also, it is recommended to create git commits after each step.

It is also recommended that some time before deprecated code is actually removed, that a TriBITS repo change the default of `<Project>_HIDE_DEPRECATED_CODE` from `OFF` to `ON` so that downstream clients will see the effects of hiding the deprecated code before the code is actually removed. In fact, this default should be changed several days to a week or more before the code is actually removed. This way, downstream code developers will get a "shock" about removal of the deprecated code but can manually configure with `-D<Project>_HIDE_DEPRECATED_CODE=OFF` to keep building in the short-term until they can remove their usage of deprecated code.

Removing entire deprecated header and source files To remove entire deprecated header and source files one just needs to first remove them from the version control repository and local directories (e.g. `git rm deprecated/*`) and then remove any traces of them from the `CMakeLists.txt` file. For the example in Hiding entire deprecated header and source files, one would just remove the files `SomeOldStuff.hpp` and `SomeOldStuff.cpp` from the `CMakeLists.txt` file leaving:

```
IF (NOT ${PACKAGE_NAME}_HIDE_DEPRECATED_CODE)
  INCLUDE_DIRECTORIES(${CMAKE_CURRENT_SOURCE_DIR}/deprecated)
  APPEND_SET(HEADERS
    )
  APPEND_SET(SOURCES
    )
ENDIF()
```

Since more files may be deprecated later, it may be a good idea to leave the machinery for conditionally including deprecated files by leaving the above empty CMake code or just commenting it out.

To find which `CMakeLists.txt` files need to be modified, do a search like:

```
$ find . -name CMakeLists.txt -exec grep -nH HIDE_DEPRECATED_CODE {} \;
```

After removing the files, create a local commit of the removed files and the updated `CMakeLists.txt` files before removing deprecated fragments from the source files. In other words, do:

```
$ emacs -nw CMakeLists.txt  # Remove the references to the deprecated files
$ git rm SomeOldStuff.hpp SomeOldStuff.cpp
$ git commit -m "Removing deprecated files"
```

Removing deprecated code from remaining files The deprecated ifdefed blocks described in Hiding C/C++ entities can be removed manually but it is generally preferred to use a tool. One simple tool that can do this is called `unifdef`, that can be downloaded and it is documented at:

http://dotat.at/prog/unifdef/

Just download, build, and install the program `unifdef` (see `unifdef/INSTALL` in untarred source) and then run it as described below. In the example below, assume the program is installed in the user's home directory under:

```
~/install/bin/unifdef
```

For a given TriBITS package, the program is then run as:

```
$ find . -name "*pp" -exec ~/install/bin/unifdef \
  -DSomePackage_HIDE_DEPRECATED_CODE {} -o {} \;
```

After the above command runs, look at the diffs to make sure the ifdef deprecated code blocks were removed correctly. For example, run:

```
$ git diff -- .
```

If the diffs look correct, commit the changes:

```
$ git commit -m "Removing deprecated code blocks" -- .
```

Then test everything and push using the checkin-test.py script.

After that, all deprecated code is removed and the next period of incremental change and deprecation begins.

10.13 Installation and Backward Compatibility Testing

TriBITS has some built-in support for installation testing and backward compatibility testing. The way it works is that one can install the headers, libraries, and executables for a TriBITS project and then configure the tests and examples in the TriBITS project against the installed headers/libraries/executables. In this mode, the TriBITS project's libraries and executables are **not** build and the header file locations to local source are not included.

When the same version of the project sources are used to build the tests/examples against the installed headers/libraries/executables, then this constitutes *installation testing*. When an older version of the project is used to build and run tests and examples against the headers/libraries/executables for a version of the project, then this constitutes *backward compatibility testing* which also includes installation testing of course.

ToDo: Describe how the installation testing and backward compatibility testing process works with some examples.

ToDo: Discuss how this fits into the TriBITS lifecycle model.

10.14 Wrapping Externally Configured/Built Software

It is possible to take an external piece of software that uses any arbitrary build system and wrap it as a TriBITS package and have it integrate in with the package dependency infrastructure. The TribitsExampleProject package `WrapExternal` shows how this can be done. Support for this in TriBITS is slowly evolving but some key TriBITS features that have been added to support the arbitrary case include:

- TRIBITS_DETERMINE_IF_CURRENT_PACKAGE_NEEDS_REBUILT(): Uses brute-force searches for recently modified files in upstream SE packages to determine if the external piece of software needs to be rebuilt.

- TRIBITS_WRITE_FLEXIBLE_PACKAGE_CLIENT_EXPORT_FILES(): Write an export makefile or a `XXXConfig.cmake` file for usage by the wrapped externally configured and built software.

While it is possible to wrap nearly any externally configured and built piece of software as a TriBITS package, in most cases, it is usually better to just create a TriBITS build system for the software. For projects that use a raw CMake build system, a TriBITS build system can be created side-by-side with the existing raw CMake build using a number of approaches. The common approach that is not too invasive is to create a `CMakeLists.tribits.txt` file along side every native `CMakeLists.txt` file in the external software project and have the native `CMakeLists.txt` file defined like:

```
IF (DOING_A_TRIBITS_BUILD)
  INCLUDE("${CMAKE_CURRENT_SOURCE_DIR}/CMakeLists.tribits.txt")
  RETURN()
ENDIF()

# Rest of native CMakeLists.txt file ...
```

Experience from the CASL VERA project has shown that, overall, there is less hassle, less work, and better portability when creating a native TriBITS build, even if it is a secondary build system for a given piece of software.

The decision whether to just wrap the build system for the existing software or to create a (secondary) TriBITS build system for it depends on a number of factors.

Note that while it is generally recommended to create a TriBITS build for an existing piece of software, it is generally not recommended with switch over all of the tests to use CTest (unless the existing software is going to ditch their current test driver and reporting system). Instead, the external software's native test system can just be called by the wrapped TriBITS package in one or more CTest tests.

10.15 TriBITS directory snapshotting

Some TriBITS projects choose to snapshot the TriBITS/tribits/ directory source tree into their project's source tree, typically under <projectDir>/cmake/tribits/. The independent `TriBITS/tribts/` source tree contains the script `snapshot_tribits.py` (calls snapshot-dir.py) that allows one to update the snapshot of the TriBITS source tree as simply as:

```
$ cd <projectDir>/cmake/tribits/
$ <some-base-dir>/TriBITS/tribits/snapshot_tribits.py
```

This will create a git commit in the local `<projectDir>/` git repo that looks like:

```
Automatic snapshot commit from tribits at f8c1682

Origin repo remote tracking branch: 'casl-dev-collab/tribits_reorg_26'
Origin repo remote repo URL: 'casl-dev-collab = git@casl-dev:collaboration/TriBITS'

At commit:

f8c1682 Assert TriBITS min CMake version in TriBITS itself
Author: Roscoe A. Bartlett <bartlettra@ornl.gov>
Date: Fri Dec 5 05:40:49 2014 -0500
```

This, of course, assumes that `<projectDir>/` is a local git repo (or is in local git repo). If that is not the case, then one cannot use the script `snapshot_tribits.py` or must use it with the `--skip-commit` option.

See snapshot-dir.py --help for more details. Note the guidance on using a different branch for the snapshot sync followed by a merge. This allows for one to maintain local changes to TriBITS and use git to manage the merges. However, this will increase the changes of merge conflicts so one should consider just directly snapshottting into the master branch to avoid merge conflicts.

10.16 TriBITS Development Toolset

Most TriBITS projects need git, a compiler (e.g. GCC), MPI, and a number of other standard TPLs and other tools in order to develop on and test the project code. To this end, TriBITS contains some helper scripts for downloading, configuring, building, and installing packages like git, cmake, GCC, MPICH, and others needed to set up a development environment for a typical computational science software project. These tools are used to set up development environments on new machines for projects like Trilinos and CASL VERA. Scripts with names like `install-gcc.py` are defined which pull sources from public git repos then configure, build, and install into specified installation directories.

The script **install_devtools.py** is provided in the directory:

```
tribits/devtools_install/
```

To use this script, one just needs to create some scratch directory like:

```
$ mkdir scratch
$ cd scratch/
```

then install the tools using, for example:

```
$ install_devtools.py --install-dir=~/install/tribits_devtools \
  --parallel=16 --do-all
```

Then to access installed development environment, one just needs to source the script:

```
~/install/tribits_devtools/load_dev_env.sh
```

and then the installed versions of GCC, MPICH, CMake, and gitdist are placed in one's path.

See install_devtools.py --help for more details.

11 References

Martin, Robert. *Agile Software Development (Principles, Patterns, and Practices)*. Prentice Hall. 2003.

Bartlett, Roscoe. *Integration Strategies for Computational Science & Engineering Software*. 2009-0655, Second International Workshop on Software Engineering for Computational Science and Engineering, 2009. http://web.ornl.gov/~8vt/CSE_SoftwareIntegration_Strategies.pdf.

SCALE: A Comprehensive Modeling and Simulation Suite for Nuclear Safety Analysis and Design, ORNL/TM-2005/39, Version 6.1, Oak Ridge National Laboratory, Oak Ridge, Tennessee, June 2011. Available from Radiation Safety Information Computational Center at Oak Ridge National Laboratory as CCC-785. http://scale.ornl.gov/

LiveV https://github.com/lifev/cmake

12 TriBITS Detailed Reference Documentation

The following subsections contain detailed reference documentation for the various TriBITS variables, functions, and macros that are used by TriBITS projects that TriBITS Project Developers need to know about. Variables, functions and macros that are used only internally in TriBITS are generally not documented here (see the TriBITS `*.cmake` source files).

12.1 TriBITS Global Project Settings

TriBITS defines a number of global project-level settings that can be set by the user and can have their default determined by each individual TriBITS project. If a given TriBITS project does not define its own default, a reasonable default is set by the TriBITS system automatically. These options are defined and are set, for the most part, in the internal TriBITS function `TRIBITS_DEFINE_GLOBAL_OPTIONS_AND_DEFINE_EXTRA_REPOS()` in the TriBITS CMake code file `TribitsGlobalMacros.cmake` which gets called inside of the TRIBITS_PROJECT()

macro. That function and that file are the definitive source the options that a TriBITS project takes and what the default values are but we strive to document them here as well. Many of these global options (i.e. cache variables) such as `${PROJECT_NAME}_<SOME_OPTION>` allow the project to define a default by setting a local variable `${PROJECT_NAME}_<SOME_OPTION>_DEFAULT` as:

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

either in its top-level `CMakeLists.txt` file or in its `ProjectName.cmake` file (depends on what variable it is as to where it should be set). If `${PROJECT_NAME}_<SOME_OPTION>_DEFAULT` is not set by the project, then TriBITS provides a reasonable default value. The TriBITS code that uses these defaults 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 an option name, for example 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 `${PROJECT_NAME}_<SOME_OPTION>_DEFAULT` before this code executes, then `${${PROJECT_NAME}_<SOME_OPTION>_DEFAULT}` will be used as the default for the cache variable `${PROJECT_NAME}_<SOME_OPTION>` which, of course, can be overridden by the user when calling `cmake` in a number of ways.

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

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

- ${PROJECT_NAME}_CPACK_SOURCE_GENERATOR
- ${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES
- ${PROJECT_NAME}_ELEVATE_ST_TO_PT
- ${PROJECT_NAME}_ENABLE_CPACK_PACKAGING
- ${PROJECT_NAME}_ENABLE_CXX11
- ${PROJECT_NAME}_ENABLE_CXX
- ${PROJECT_NAME}_ENABLE_C
- ${PROJECT_NAME}_ENABLE_DEVELOPMENT_MODE
- ${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES
- ${PROJECT_NAME}_ENABLE_Fortran
- ${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES
- ${PROJECT_NAME}_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION
- ${PROJECT_NAME}_GENERATE_EXPORT_FILE_DEPENDENCIES
- ${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS
- ${PROJECT_NAME}_REQUIRES_PYTHON
- ${PROJECT_NAME}_SHOW_TEST_START_END_DATE_TIME
- ${PROJECT_NAME}_TEST_CATEGORIES

- ${PROJECT_NAME}_TPL_SYSTEM_INCLUDE_DIRS
- ${PROJECT_NAME}_TRACE_ADD_TEST
- ${PROJECT_NAME}_USE_GNUINSTALLDIRS
- ${PROJECT_NAME}_USES_PYTHON
- MPI_EXEC_MAX_NUMPROCS
- PythonInterp_FIND_VERSION

These options are described below.

${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 overridden 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}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES

If ${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES=ON (the TriBITS default value), then any explicitly enabled packages that have disabled upstream required packages or TPLs will be disabled. If ${PROJECT_NAME}_DISABLE_ENABLED_FORWARD_DEP_PACKAGES=OFF, then an configure error will occur. For more details also see TribitsBuildReference and Disables trump enables where there is a conflict. A project can define a different default value by setting:

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

${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 distinction between `PT` and `ST` code is not helpful or the assignment of `PT` and `ST` packages in a repository is not appropriate with respect to the outer meta-project. An example project like this CASL VERA. Changing the default to `ON` allows any and 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 run 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_DEFAULT ON)
```

${PROJECT_NAME}_ENABLE_CXX11

If `${PROJECT_NAME}_ENABLE_CXX11` is `ON`, then C++ compiler options that turn on C++11 support will be searched for. By default, TriBITS sets this to `OFF` for all systems. However, if project requires C++11 support by default, then the project should set the default:

SET(${PROJECT_NAME}_ENABLE_CXX11_DEFAULT TRUE)

${PROJECT_NAME}_ENABLE_CXX

If ${PROJECT_NAME}_ENABLE_CXX is ON, then C++ language support for the project will be enabled and the C++ compiler must be found. By default, TriBITS sets this to ON for all systems. A project never requires C++ can set this to off by default by setting:

SET(${PROJECT_NAME}_ENABLE_CXX_DEFAULT FALSE)

${PROJECT_NAME}_ENABLE_C

If ${PROJECT_NAME}_ENABLE_C is ON, then C language support for the project will be enabled and the C compiler must be found. By default, TriBITS sets this to ON for all systems. A project never requires C can set this to off by default by setting:

SET(${PROJECT_NAME}_ENABLE_C_DEFAULT FALSE)

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

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

NOTE: It is usually not a good idea to always force off C, 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}_ENABLE_DEVELOPMENT_MODE

The variable ${PROJECT_NAME}_ENABLE_DEVELOPMENT_MODE switches the TriBITS project from development mode to release mode. The default for this variable ${PROJECT_NAME}_ENABLE_DEVELOPMENT_MODE_DEFAULT should be set in the project's <projectDir>/Version.cmake file and switched from ON to OFF when creating a release (see Project and Repository Versioning and Release Mode). When ${PROJECT_NAME}_ENABLE_DEVELOPMENT_MODE is ON, several other variables are given defaults appropriate for development mode. For example, ${PROJECT_NAME}_ASSERT_MISSING_PACKAGES is set to ON by default in development mode but is set to OFF by default in release mode. In addition, strong compiler warnings are enabled by default in development mode but are disabled by default in release mode. This variable also affects the behavior of TRIBITS_SET_ST_FOR_DEV_MODE().

${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES

If ${PROJECT_NAME}_ENABLE_EXPORT_MAKEFILES is ON, then Makefile.export.<PackageName> will get created at configure time in the build tree and installed into the install tree. See TribitsBuildReference 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 libraries and headers are not actually installed (see ${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS)

${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 that 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}_ENABLE_INSTALL_CMAKE_CONFIG_FILES

If `${PROJECT_NAME}_ENABLE_INSTALL_CMAKE_CONFIG_FILES` is set to `ON`, then `<PackageName>Config.cmake` files are created at configure time in the build tree and installed into the install tree. These files are used by external CMake projects to pull in the list of compilers, compiler options, include directories and libraries. The TriBITS default is `ON` but 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 `<PackageName>Config.cmake` files if it was only installing and providing executables (see ${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS).

${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}_GENERATE_EXPORT_FILE_DEPENDENCIES

If `${PROJECT_NAME}_GENERATE_EXPORT_FILE_DEPENDENCIES` is `ON`, then the data-structures needed to generate `Makefile.export.<PackageName>` and `<PackageName>Config.cmake` are created. These data structures are also needed in order to generate export makefiles on demand using the function TRIBITS_WRITE_FLEXIBLE_PACKAGE_CLIENT_EXPORT_FILES(). The default in TriBITS is to turn this `ON` automatically by default if `${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}_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() or TRIBITS_INSTALL_HEADERS() will be installed (unless options are passed into TRIBITS_ADD_LIBRARY() that disable installs). If set to `OFF`, then headers and libraries will *not* be installed by default and only `INSTALLABLE` executables added with TRIBITS_ADD_EXECUTABLE() will be installed. However, as described in TribitsBuildReference, shared libraries will always be installed if enabled since they are needed by the installed executables.

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

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

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

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

${PROJECT_NAME}_REQUIRES_PYTHON

If the TriBITS project requires Python, set:

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

in the <projectDir>/ProjectName.cmake file (See Python Support). The default is implicitly `FALSE`.

${PROJECT_NAME}_SHOW_TEST_START_END_DATE_TIME

The cache variable `${PROJECT_NAME}_SHOW_TEST_START_END_DATE_TIME` determines if the start and end date/time for each advanced test (i.e. added with TRIBITS_ADD_ADVANCED_TEST()) is printed or not with each test. The TriBITS default is `OFF` but a TriBITS project can change this default by setting:

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

The implementation of this feature currently uses `EXECUTE_PROCESS(date)` and therefore will only work on Linux/Unix/Mac systems and not Windows systems.

NOTE: In a future version of CTest, this option may turn on start and end date/time for regular tests added with TRIBITS_ADD_TEST() (which uses a raw command with `ADD_TEST()`).

${PROJECT_NAME}_TEST_CATEGORIES

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

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

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

${PROJECT_NAME}_TPL_SYSTEM_INCLUDE_DIRS

If `${PROJECT_NAME}_TPL_SYSTEM_INCLUDE_DIRS` is set to `TRUE`, then the `SYSTEM` flag will be passed into the `INCLUDE_DIRECTORIES()` command for TPL include directories for every TPL for every package, by default. On some systems this will result in include directories being passed to the compiler with `-isystem` instead of `-I`. This helps to avoid compiler warning coming from TPL header files for C and C++. However, with CMake version 3.2 and less, this also results in `-isystem` being passed to the Fortran compiler (e.g. gfortran) as well. This breaks the reading of Fortran module files (perhaps a bug in gfortran). Because of this issue with Fortran, the TriBITS default for this option is set to `FALSE` but a project can override the default using:

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

(This would be a good default if the project has not Fortran files or has not Fortran files that use modules provided by TPLs).

However, if a package or subpackage sets:

```
SET(${PACKAGE_NAME}_SKIP_TPL_SYSTEM_INCLUDE_DIRS  TRUE)
```

in its CMakeLists.txt files before the TRIBITS_ADD_LIBRARY() or TRIBITS_ADD_EXECUTABLE() commands are called in that package, then SYSTEM will **not** be passed into INCLUDE_DIRECTORIES() for TPL include dirs. This is how some TriBITS packages with Fortran files that use Fortran modules avoid passing in -isystem to the Fortran compiles and thereby avoid the defect with gfortran described above. If CMake version 3.3 or greater is used, this variable is not required.

NOTE: Currently, a TriBITS SE package must have a direct dependency on a TPL to have -isystem added to a TPL's include directories on the compile lines for that package. That is, the TPL must be listed in the LIB_REQUIRED_TPLS or LIB_OPTIONAL_TPLS arguments passed into the TRIBITS_PACKAGE_DEFINE_DEPENDENCIES() function in the SE package's <packageDir>/cmake/Dependencies.cmake file. In addition, to have -isystem added to the include directories for a TPL when compiling the tests for an SE package, it must be listed in the TEST_REQUIRED_TPLS or TEST_OPTIONAL_TPLS arguments. This is a limitation of the TriBITS implementation that will be removed in a future version of TriBITS.

${PROJECT_NAME}_TRACE_ADD_TEST

If ${PROJECT_NAME}_TRACE_ADD_TEST is set to TRUE, then a single line will be printed for each call to TRIBITS_ADD_TEST() and TRIBITS_ADD_ADVANCED_TEST() for if the test is added or not and if not then why. The default is set based on the value of ${PROJECT_NAME}_VERBOSE_CONFIGURE but a project can override the default by setting:

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

${PROJECT_NAME}_USE_GNUINSTALLDIRS

If ${PROJECT_NAME}_USE_GNUINSTALLDIRS is set to TRUE, then the default install paths will be determined by the standard CMake module GNUInstallDirs. Otherwise, platform independent install paths are used by default.

A project can use the paths given the cmake module GNUInstallDirs by default by setting:

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

in the project's top-level <projectDir>/CMakeLists.txt file or its <projectDir>/ProjectName.cmake file. The default is FALSE.

${PROJECT_NAME}_USES_PYTHON

If the TriBITS project can use Python, but does not require it, set:

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

in the <projectDir>/ProjectName.cmake file (see Python Support). The default for a TriBITS project is implicitly TRUE. To explicitly state that Python is never needed, set:

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

MPI_EXEC_MAX_NUMPROCS

The variable 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 load that can be accommodated by a given machine (or class of machines). For example if a given machine has 64 cores, a reasonable number for MPI_EXEC_MAX_NUMPROCS_DEFAULT is 64.

PythonInterp_FIND_VERSION

Determines the version of Python that is looked for. TriBITS requires at least version "2.4". A particular TriBITS project can require a higher version of TriBITS and this is set using, for example:

SET(PythonInterp_FIND_VERSION_DEFAULT "2.6.6")

in the <projectDir>/ProjectName.cmake file (See Python Support). The default is version "2.4". The user can force a more recent version of Python by configuring with, for example:

```
-D PythonInterp_FIND_VERSION="2.7.3"
```

12.2 TriBITS Macros and Functions

The following subsections give detailed documentation for the CMake macros and functions that make up the core TriBITS system. These are what are used by TriBITS project developers in their `CMakeLists.txt` and other files. All of these functions and macros should be automatically available when processing the project's and package's variables files if used properly. Therefore, no explicit `INCLUDE()` statements should be needed other than the initial include of the `TriBITS.cmake` file in the top-level <projectDir>/CMakeLists.txt file so the command TRIBITS_PROJECT() can be executed.

TRIBITS_ADD_ADVANCED_TEST()

Function that creates an advanced test defined by stringing together one or more executable and/or command invocations that is run as a `cmake -P` script with very flexible pass/fail criteria.

Usage:

```
TRIBITS_ADD_ADVANCED_TEST(
  <testNameBase>
  TEST_0 (EXEC <execTarget0> | CMND <cmndExec0>) ...
  [TEST_1 (EXEC <execTarget1> | CMND <cmndExec1>) ...]
  ...
  [TEST_N (EXEC <execTargetN> | CMND <cmndExecN>) ...]
  [OVERALL_WORKING_DIRECTORY (<overallWorkingDir> | TEST_NAME)]
  [SKIP_CLEAN_OVERALL_WORKING_DIRECTORY]
  [FAIL_FAST]
  [RUN_SERIAL]
  [KEYWORDS <keyword1> <keyword2> ...]
  [COMM [serial] [mpi]]
  [OVERALL_NUM_MPI_PROCS <overallNumProcs>]
  [OVERALL_NUM_TOTAL_CORES_USED <overallNumTotalCoresUsed>]
  [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> ...]
  [TIMEOUT <maxSeconds>]
  [ADDED_TEST_NAME_OUT <testName>]
  )
```

This function allows one to add a single CTest test that is actually a sequence of one or more separate commands strung together in some way to define the final pass/fail. One will want to use this function to add a test instead of TRIBITS_ADD_TEST() when one needs to run more than one command, or one needs more sophisticated checking of the test result other than just grepping STDOUT (e.g. by running separate post-processing programs to examine output files).

Each atomic test case is either a package-built executable or just a basic command. An atomic test command block `TEST_<idx>` (i.e. `TEST_0`, `TEST_1`, ...) takes the form:

```
TEST_<idx>
   (EXEC <exeRootName> [NOEXEPREFIX] [NOEXESUFFIX] [ADD_DIR_TO_NAME]
          [DIRECTORY <dir>]
      | CMND <cmndExec>)
   [ARGS <arg1> <arg2> ... <argn>]
   [MESSAGE "<message>"]
   [WORKING_DIRECTORY <workingDir>]
   [SKIP_CLEAN_WORKING_DIRECTORY]
   [NUM_MPI_PROCS <numProcs>]
   [NUM_TOTAL_CORES_USED <numTotalCoresUsed>]
   [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())
- TEST_<idx> Test Blocks and Arguments (TRIBITS_ADD_ADVANCED_TEST())
- Overall Pass/Fail (TRIBITS_ADD_ADVANCED_TEST())
- Argument Parsing and Ordering (TRIBITS_ADD_ADVANCED_TEST())
- Implementation Details (TRIBITS_ADD_ADVANCED_TEST())
- Setting Additional Test Properties (TRIBITS_ADD_ADVANCED_TEST())
- Running multiple tests at the same time (TRIBITS_ADD_ADVANCED_TEST())
- Disabling Tests Externally (TRIBITS_ADD_ADVANCED_TEST())
- Debugging and Examining Test Generation (TRIBITS_ADD_ADVANCED_TEST())

Overall Arguments (TRIBITS_ADD_ADVANCED_TEST())

Below, some of the overall arguments are described. The rest of the overall arguments that control overall pass/fail are described in Overall Pass/Fail (TRIBITS_ADD_ADVANCED_TEST()). (NOTE: All of these arguments must be listed outside of the `TEST_<idx>` blocks, see Argument Parsing and Ordering (TRIBITS_ADD_ADVANCED_TEST())).

`<testNameBase>`

The base name of the test (which will have `${PACKAGE_NAME}_` prepended to the name, see <testName> below) 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 to this function.

`OVERALL_WORKING_DIRECTORY <overallWorkingDir>`

If specified, then the working directory `<overallWorkingDir>` (relative or absolute path) 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 `<testName>`. By default, if the directory `<overallWorkingDir>` exists before the test runs, it will be deleted and created again. If one wants to preserve the contents of this directory between test runs then set `SKIP_CLEAN_OVERALL_WORKING_DIRECTORY`. Using a separate test directory is a good option to use if the commands create intermediate files and one wants to make sure they get deleted before the test cases are run again. It is also important to create a separate test directory if multiple tests are defined in the same `CMakeLists.txt` file that read/write files with the same name.

`SKIP_CLEAN_OVERALL_WORKING_DIRECTORY`

If specified, then `<overallWorkingDir>` will **not** be deleted if it already exists.

`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 MPI processes that each executable command runs on. If `<overallNumProcs>` 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. For MPI builds with all `TEST_<IDX> CMND` blocks, `<overallNumProcs>` is used to set the property `PROCESSORS`. (see Running multiple tests at the same time (TRIBITS_ADD_ADVANCED_TEST())). **WARNING!** If just running a serial script or other command, then the property `PROCESSORS` will still get set to `${OVERALL_NUM_MPI_PROCS}` so in order to avoid CTest unnecessarily reserving `${OVERALL_NUM_MPI_PROCS}` processes for a serial non-MPI test, then one must leave off `OVERALL_NUM_MPI_PROCS` or explicitly pass in `MPI_EXEC_DEFAULT_NUMPROCS 1`!

`OVERALL_NUM_TOTAL_CORES_USED <overallNumTotalCoresUsed>`

Used for `NUM_TOTAL_CORES_USED` if missing in a `TEST_<IDX>` block.

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

Gives the Test 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 variables will be set before calling the test. This is set using the built-in CTest test property `ENVIRONMENT`.

`TIMEOUT <maxSeconds>`

If passed in, gives maximum number of seconds the test will be allowed to run before being timed-out (see TRIBITS_ADD_TEST()). This is for the full CTest test, not individual `TEST_<idx>` commands!

`ADDED_TEST_NAME_OUT <testName>`

If specified, then on output the variable `<testName>` will be set with the name of the test passed to `ADD_TEST()`. Having this name allows the calling `CMakeLists.txt` file access and set additional test propeties (see Setting additional test properties (TRIBITS_ADD_ADVANCED_TEST())).

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

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

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

If `EXEC` is specified, then `<exeRootName>` gives the root name of an executable target that will be run as the command. The full executable name and path is determined in exactly the same way it is in the TRIBITS_ADD_TEST() function (see Determining the Executable 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 maximum number of MPI processes allowed is less than this number of MPI processes, then the test will *not* be run. Note that `EXEC <exeRootName>` when `NOEXEPREFIX` and `NOEXESUFFIX` are specified is basically equivalent to `CMND <cmndExec>` except that in an MPI build, `<exeRootName>` is always run using MPI. In this case, one can pass in `<exeRootName>` to any command one would like and it will get run with MPI in MPI mode just link any other MPI-enabled built executable.

`CMND <cmndExec>`

If `CMND` is specified, then `<cmndExec>` gives the executable for a command to be run. In this case, MPI will never be used to run the executable even when configured in MPI mode (i.e. `TPL_ENABLE_MPI=ON`). If one wants to run an arbitrary command using MPI, use `EXEC <fullPathToCmndExec> NOEXEPREFIX NOEXESUFFIX` instead. **WARNING:** If you want to run such tests using valgrind, you have to use the raw executable as the `<cmndExec>` argument and *not* the script. For example, if you have a python script `my_python_test.py` with `/usr/bin/env pyhton` at the top, you can't just use:

```
CMND <path>/my_python_test.py ARGS <arg0> <arg1> ...
```

The same goes for Perl or any other scripting language.

Instead, you have to use:

```
CMND ${PYTHON_EXECUTABLE} ARGS <path>/my_python_test.py <arg0> <arg1> ...
```

By default, the output (stdout/stderr) for each test command is captured and is then echoed to stdout for the overall test. This is done in order to be able to grep the result to determine pass/fail.

Other miscellaneous arguments for each `TEST_<idx>` block include:

`DIRECTORY <dir>`

If specified, then the executable is assumed to be in the directory given by relative `<dir>`. See TRIBITS_ADD_TEST().

`MESSAGE "<message>"`

If specified, then the string in `"<message>"` will be printed 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> (relative or absolute) will be created and the test will be run from within this directory. If the directory <workingDir> exists before the test runs, it will be deleted and created again. If one wants to preserve the contents of this directory between test blocks, then one needs to set SKIP_CLEAN_WORKING_DIRECTORY. Using a different 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.

SKIP_CLEAN_WORKING_DIRECTORY

If specified, then <workingDir> will **not** be deleted if it already exists.

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>.

NUM_TOTAL_CORES_USED <numTotalCoresUsed>

If specified, gives the total number of processes used by this command/executable. If this is missing, but NUM_MPI_PROCS <numProcs> is specified, then <numProcs> is used instead. If NUM_TOTAL_CORES_USED is missing BUT OVERALL_NUM_TOTAL_CORES_USED <overallNumTotalCoresUsed> is, then <overallNumTotalCoresUsed> is used for <numTotalCoresUsed>. This argument is used for test scripts/executables that use more cores than MPI processes (i.e. <numProcs>) and its only purpose is to inform CTest and TriBITS of the maximum number of cores that are used by the underlying test executable/script. When <numTotalCoresUsed> is greater than ${MPI_EXEC_MAX_NUMPROCS}, then the test will not be added. Otherwise, the CTest property PROCESSORS is set to the max over all <numTotalCoresUsed> so that CTest knows how to best schedule the test w.r.t. other tests on a given number of available processes.

OUTPUT_FILE <outputFile>

If specified, then stdout and stderr for the test case will be sent to <outputFile>. By default, the contents of this file will **also** be printed to STDOUT unless NO_ECHO_OUT is passed as well.

NOTE: Contrary to CMake documentation for EXECUTE_PROCESS(), STDOUT and STDERR may not get output in the correct order interleaved correctly, even in serial without MPI. Therefore, you can't write any tests that depend on the order of STDOUT and STDERR output in relation to each other. Also note that all of STDOUT and STDERR will be first read into the CTest executable process main memory before the file <outputFile> is written. Therefore, don't run executables or commands that generate massive amounts of console output or it may exhaust main memory. Instead, have the command or executable write directly to a file instead of going through STDOUT.

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 or commands returns a non-zero value to the shell. However, a test case can also be defined to pass based on:

PASS_ANY

If specified, the test command will be assumed to pass regardless 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 will be assumed to pass if it matches the given regular expression. Otherwise, it is assumed to fail. TIPS: Replace ';' with '[;]' or CMake will interpretet this as a array eleemnt boundary. To match '.', use '[.]'.

PASS_REGULAR_EXPRESSION_ALL "<regex1>" "<regex2>" ... "<regexn>"

If specified, the test command will be assumed to pass 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. NOTE: It is critical that you replace ';' with '[;]' or CMake will interpretet this as a array eleemnt boundary.

`FAIL_REGULAR_EXPRESSION "<regex>"`

If specified, the test command 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 will be assumed to pass if the string expression "Final Result: PASSED" is found in the output for the test.

All of the arguments for a test block `TEST_<idx>` must appear directly below their `TEST_<idx>` argument and before the next test block (see Argument Parsing and Ordering (TRIBITS_ADD_ADVANCED_TEST())).

Overall Pass/Fail (TRIBITS_ADD_ADVANCED_TEST())

By default, the overall test will be assumed to pass if it prints:

```
"OVERALL FINAL RESULT: TEST PASSED (<testName>)"
```

However, this can be changed by setting one of the following optional arguments:

`FINAL_PASS_REGULAR_EXPRESSION <regex>`

If specified, the test will be assumed to pass if the output matches `<regex>`. Otherwise, it will be assumed to fail.

`FINAL_FAIL_REGULAR_EXPRESSION <regex>`

If specified, the test will be assumed to fail if the output matches `<regex>`. Otherwise, it will be assumed to fail.

Argument Parsing and Ordering (TRIBITS_ADD_ADVANCED_TEST())

The basic tool used for parsing the arguments to this function is the macro PARSE_ARGUMENTS() which has a certain set of behaviors. The parsing using PARSE_ARGUMENTS() is actually done in two phases. There is a top-level parsing of the "overall" arguments listed in Overall Arguments (TRIBITS_ADD_ADVANCED_TEST()) that also pulls out the test blocks. Then there is a second level of parsing using `PARSE_ARGUMENTS()` for each of the `TEST_<idx>` blocks. Because of this usage, there are a few restrictions that one needs to be aware of when using `TRIBITS_ADD_ADVANCED_TEST()`. This short sections tries to explain the behaviors and what is allowed and what is not allowed.

For the most part, the "overall" arguments and the arguments inside of any individual `TEST_<idx>` blocks can be listed can appear in any order but there are restrictions related to the grouping of overall arguments and `TEST_<idx>` blocks which are as follows:

- The `<testNameBase>` 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 (e.g. 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 listed inside of a `TEST_<idx>` block but otherwise can be listed before or after all of the `TEST_<idx>` blocks. (NOTE: The current implementation will actually allow overall arguments to be listed after all of the local arguments before the next TEST_<idx> block but this is confusing and will not be allowed in a future implementation).

Other than that, the keyword arguments 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 -P` script with the name `<testName>.cmake` gets created in the current binary directory that then gets added to CTest using:

```
ADD_TEST(<testName> cmake [other options] -P <testName>.cmake)
```

This `cmake -P` script then runs the various test cases and checks the pass/fail for each case to determine overall pass/fail and implement other functionality described above.

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_TESTS_PROPERTIES(<testName> ...)`, where `<testName>` is returned using the `ADDED_TEST_NAME_OUT <testName>` argument. 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()`. For example:

```
TRIBITS_ADD_ADVANCED_TEST_TEST( someTest ...
  ADDED_TEST_NAME_OUT  someTest_TEST_NAME )

IF (someTest_TEST_NAME)
  SET_TESTS_PROPERTIES( ${someTest_TEST_NAME}
    PROPERTIES ATTACHED_FILES someTest.log )
ENDIF()
```

where the test writes a log file `someTest.log` that we want to submit to CDash also.

Runnning multiple tests at the same time (TRIBITS_ADD_ADVANCED_TEST())

Just as with TRIBITS_ADD_TEST(), setting `NUM_MPI_PROCS <numProcs>` or `OVERALL_NUM_MPI_PROCS <numOverallProcs>` or `NUM_TOTAL_CORES_USED <numTotalCoresUsed>` or `OVERALL_NUM_TOTAL_CORES_USED <overallNumTotalCoresUsed>` will set the `PROCESSORS` CTest property to allow CTest to schedule and run mutiple tests at the same time when `'ctest -j<N>'` is used (see Running multiple tests at the same time (TRIBITS_ADD_TEST())).

Disabling Tests Externally (TRIBITS_ADD_ADVANCED_TEST())

The test can be disabled externally by setting the CMake cache variable `<testName>_DISABLE=TRUE`. This allows tests to be disabled on a case-by-case basis. The name `<testName>` must be 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 what tests get added and if not then why, configure with `${PROJECT_NAME}_TRACE_ADD_TEST=ON`. That will print one line per show that the test got added and if not then why the test was not added (i.e. due to `COMM`, `OVERALL_NUM_MPI_PROCS`, `NUM_MPI_PROCS`, `CATEGORIES`, `HOST`, `XHOST`, `HOSTTYPE`, or `XHOSTTYPE`).

Likely the best way to debugging test generation using this function is to examine the generated file `<testName>.cmake` in the current binary directory (see Implementation Details (TRIBITS_ADD_ADVANCED_TEST())) and the generated `CTestTestfile.cmake` file that should list this test case.

TRIBITS_ADD_DEBUG_OPTION()

Add the standard cache variable option `${PACKAGE_NAME}_ENABLE_DEBUG` for the package.

Usage:

```
TRIBITS_ADD_DEBUG_OPTION()
```

This option is given the default `${${PROJECT_NAME}_ENABLE_DEBUG}` and if true, will set the variable `HAVE_${PACKAGE_NAME_UC}_DEBUG` (to be used in the package's configured header file). This macro is typically called in the package's <packageDir>/CMakeLists.txt file.

TRIBITS_ADD_EXAMPLE_DIRECTORIES()

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

Usage:

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

This macro typically is called from the top-level <packageDir>/CMakeLists.txt file for which all subdirectories are all "examples" according to standard package layout.

This macro can be called several times within a package as desired to break up example directories any way one would like.

Currently, 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 future in order to modify behavior related to adding tests and examples in a uniform way.

TRIBITS_ADD_EXECUTABLE()

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

Usage:

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

Sections:

- Formal Arguments (TRIBITS_ADD_EXECUTABLE())
- Executable and Target Name (TRIBITS_ADD_EXECUTABLE())
- Additional Executable and Source File Properties (TRIBITS_ADD_EXECUTABLE())
- Install Target (TRIBITS_ADD_EXECUTABLE())

Formal Arguments (TRIBITS_ADD_EXECUTABLE())

`<exeRootName>`

The root name of the exectuable (and CMake target) (see Executable and Target Name (TRIBITS_ADD_EXECUTABLE())). This must be the first argument.

`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's 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 can contain the relative path or absolute path). 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(<exeTargetName> ... )`. After calling this function, the properties of the source files can be altered using the built-in CMake command `SET_SOURCE_FILE_PROPERTIES()`.

`DIRECTORY <dir>`

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

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

Gives the Test 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()).

`TESTONLYLIBS <lib0> <lib1> ...`

Specifies extra test-only libraries defined in this CMake project 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 can *NOT* be listed! TriBITS automatically links non `TESTONLY` libraries in this package and upstream packages to the executable. The only libraries that should be listed in this argument are either `TESTONLY` libraries. The include directories for each test-only library 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 `TESTONLYLIBS ... <libi> ...` and that is it!

`IMPORTEDLIBS <lib0> <lib1> ...`

Specifies extra external libraries that will be linked to the executable using `TARGET_LINK_LIBRARY()`. This can only be used for libraries that are built external from this CMake project and are not provided through a proper TriBITS TPL. The latter usage of passing in external libraries is not recommended. External libraries should be handled as declared TriBITS TPLs. So far, the only case where `IMPORTEDLIBS` has been shown to be necessary is to pass in the standard C math library `m`. In every other case, a TriBITS TPL should be used instead.

`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 built-in CMake target property `LINKER_LANGUAGE`. By default, CMake chooses the compiler to be used as the linker based on file extensions. The most typical use case for this option 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 executable into the `${CMAKE_INSTALL_PREFIX}/bin/` directory (see Install Target (TRIBITS_ADD_EXECUTABLE())).

`ADDED_EXE_TARGET_NAME_OUT <exeTargetName>`

If specified, then on output the variable `<exeTargetName>` will be set with the name of the executable target passed to `ADD_EXECUTABLE(<exeTargetName> ... )`. Having this name allows the calling `CMakeLists.txt` file access and set additional target propeties (see Additional Executable and Source File Properties (TRIBITS_ADD_EXECUTABLE())).

Executable and Target Name (TRIBITS_ADD_EXECUTABLE())

By default, the full name of the executable and target name is:

```
<exeTargetName> = ${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:

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

If the option `NOEXEPREFIX` is passed in, then the prefix `${PACKAGE_NAME}_` is removed.

The executable suffix `${${PROJECT_NAME}_CMAKE_EXECUTABLE_SUFFIX}` will be added to the actual executable file name 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 executable 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). For general utility executables on Linux/Unix systems, `NOEXEPREFIX` and `NOEXESUFFIX` should be passed in. In this case, one must be careful to pick `<exeRootName>` that will be sufficiently globally unique. Please use common sense when picking non-namespaced names.

Additional Executable and Source File Properties (TRIBITS_ADD_EXECUTABLE())

Once ADD_EXECUTABLE(<exeTargetName> ...) is called and this function exists, one can set and change properties on the <exeTargetName> executable target using the built-in SET_TARGET_PROPERTIES() command as well as properties on any of the source files listed in SOURCES using the built-in SET_SOURCE_FILE_PROPERTIES() command just like in any CMake project. IF the executable is added, its name will be returned by the argument ADDED_EXE_TARGET_NAME_OUT <exeTargetName>. For example:

```
TRIBITS_ADD_EXECUTABLE( someExe ...
  ADDED_EXE_TARGET_NAME_OUT  someExe_TARGET_NAME )

SET_TARGET_PROPERTIES( ${someExe_TARGET_NAME}
  PROPERTIES  LINKER_LANGUAGE  CXX )
```

Install Target (TRIBITS_ADD_EXECUTABLE())

If INSTALLABLE is passed in, then an install target using the built-in CMake command INSTALL(TARGETS <exeTargetName> ...) 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, see Setting the install prefix at configure time) .

TRIBITS_ADD_EXECUTABLE_AND_TEST()

Add an executable and a test (or several tests) all in one shot (just calls TRIBITS_ADD_EXECUTABLE() followed by TRIBITS_ADD_TEST()).

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>]
  [TESTONLYLIBS <lib0> <lib1> ...]
  [IMPORTEDLIBS <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>]
  [ADDED_EXE_TARGET_NAME_OUT <exeTargetName>]
  [ADDED_TESTS_NAMES_OUT <testsNames>]
  )
```

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 accepted by which functions.

Arguments that are specific to this function and not directly passed on to 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. These are just passed in through the `XHOST` argument to `TRIBITS_ADD_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. These are just passed in through the `XHOSTTYPE` argument to `TRIBITS_ADD_TEST()`.

This is the function to use for simple test executables that you want to run that either takes no arguments or just a simple set of arguments passed in through `ARGS`. For more flexibility, just use `TRIBITS_ADD_EXECUTABLE()` followed by `TRIBITS_ADD_TEST()`.

TRIBITS_ADD_LIBRARY()

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

Usage:

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

Sections:

- Formal Arguments (TRIBITS_ADD_LIBRARY())
- Include Directories (TRIBITS_ADD_LIBRARY())
- Install Targets (TRIBITS_ADD_LIBRARY())
- Additional Library and Source File Properties (TRIBITS_ADD_LIBRARY())
- Miscellaneous Notes (TRIBITS_ADD_LIBRARY())

Formal Arguments (TRIBITS_ADD_LIBRARY())

`<libBaseName>`

Required base name of the library. The name of the actual libray name will be prefixed by `${${PROJECT_NAME}_LIBRARY_NAME_PREFIX}` to produce:

```
<libTargetName> = ${${PROJECT_NAME}_LIBRARY_NAME_PREFIX}<libBaseName>
```

This is the name passed to `ADD_LIBRARY(<libTargetName> ...)`. The name is *not* prefixed by the package 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 (e.g. on Linux prefix = `'lib'`, postfix = `'.so'` for shared lib and postfix = `'.a'` static lib) (see documentation for the built-in CMake command `ADD_LIBRARY()`.

`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. This 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 Studio). By default, these headers will be installed (see Install Targets (TRIBITS_ADD_LIBRARY())).

`HEADERS_INSTALL_SUBDIR <headerssubdir>`

Optional subdirectory that the headers will be installed under the standard installation directory. If `<headerssubdir>!=""`, then the headers will be installed under `${PROJECT_NAME}_INSTALL_INCLUDE_DIR}/<headerssubdir>`. Otherwise, they will be installed under `${PROJECT_NAME}_INSTALL_INCLUDE_DIR}/`. Install Targets (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 for `ADD_LIBRARY()`).

`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(<libTargetName> ...)` so that CMake knows about the dependency structure of the libraries within this SE package. **NOTE:** One must **not** list libraries in other upstream TriBITS SE Packages or libraries built externally from this TriBITS CMake project in `DEPLIBS`. The TriBITS system automatically handles linking to libraries in upstream TriBITS SE packages. External libraries need to be listed in the `IMPORTEDLIBS` argument instead if they are not already specified automatically using a TriBITS TPL.

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

List of dependent libraries built externally from this TriBITS CMake project. These libraries are passed into `TARGET_LINK_LIBRARIES(<libTargetName> ...)` so that CMake knows about the dependency. However, note that external libraries are often better handled as TriBITS TPLs. A well constructed TriBITS package and library should never have to use this option! So far, the only case where `IMPORTEDLIBS` has been shown to be necessary is to pass in the standard C math library `m`. In every other case, a TriBITS TPL should be used instead.

`TESTONLY`

If passed in, then `<libTargetName>` 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 `<libTargetName>_INCLUDE_DIR` which will be used in TRIBITS_ADD_EXECUTABLE() when a test-only library is linked in through its `DEPLIBS` argument.

`NO_INSTALL_LIB_OR_HEADERS`

If specified, then no install targets will be added for the library `<libTargetName>` 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 (see Standard TriBITS TPLs). 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 may occur about not knowing about CUDA_ALL_LIBRARY().

ADDED_LIB_TARGET_NAME_OUT <libTargetName>

If specified, then on output the variable <libTargetName> will be set with the name of the library passed to ADD_LIBRARY(). Having this name allows the calling CMakeLists.txt file access and set additional target propeties (see Additional Library and Source File Properties (TRIBITS_ADD_LIBRARY())).

Include Directories (TRIBITS_ADD_LIBRARY())

Any base directories for the header files listed in the arguments HEADERS or NOINSTALLHEADERS should be passed into the standard CMake command INCLUDE_DIRECTORIES() *before* calling this function. These include directories will then be added to current packages list of include directories ${PACKAGE_NAME}_INCLUDE_DIRS which is then exported to downstream SE packages..

Install Targets (TRIBITS_ADD_LIBRARY())

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

By default, an install target for the headers listed in HEADERS will get added using INSTALL(FILES <h0> <h1> ...), but only if TESTONLY and NO_INSTALL_LIB_OR_HEADERS are not passed in as well. Also, the install target for the headers will not get added if ${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS is FASLE. If this install target is added, then the headers get installed into the flat directory ${${PROJECT_NAME}_INSTALL_INCLUDE_DIR}/ (default is ${CMAKE_INSTALL_PREFIX}/include/, see Setting the install prefix at configure time). If HEADERS_INSTALL_SUBDIR is set, then the headers will be installed under ${${PROJECT_NAME}_INSTALL_INCLUDE_DIR}/<headerssubdir>/.

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(<libTargetName> ... <src0> <src1> ...) is called, one can set and change properties on the <libTargetName> library target using the built-in CMake command SET_TARGET_PROPERTIES() as well as set and change properties on any of the source files listed in SOURCES using the built-in CMake command SET_SOURCE_FILE_PROPERTIES() just like in any CMake project. For example:

```
TRIBITS_ADD_LIBRARY( somelib ...
  ADDED_LIB_TARGET_NAME_OUT  somelib_TARGET_NAME )

SET_TARGET_PROPERTIES( ${somelib_TARGET_NAME}
  PROPERTIES  LINKER_LANGUAGE  CXX )
```

Miscellaneous Notes (TRIBITS_ADD_LIBRARY())

WARNING: Do **NOT** use the built-in CMake command ADD_DEFINITIONS() to add defines -D<someDefine> to the compile command line that will affect any of the header files in the package! These CMake-added 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 TriBITS packages) compiles that may include these header files. To add defines that affect header files, please use a configured header file (see TRIBITS_CONFIGURE_FILE()).

TRIBITS_ADD_OPTION_AND_DEFINE()

Add an option and a define variable in one shot.

Usage:

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

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

```
#cmakedefine <macroDefineName>
```

TRIBITS_ADD_SHOW_DEPRECATED_WARNINGS_OPTION()

Add the standard option `${PACKAGE_NAME}_SHOW_DEPRECATED_WARNINGS` for the package.

Usage:

```
TRIBITS_ADD_SHOW_DEPRECATED_WARNINGS_OPTION()
```

This macro should be called in the package's <packageDir>/CMakeLists.txt'_ file. This option is given the default value `${${PROJECT_NAME}_SHOW_DEPRECATED_WARNINGS}`. This option is then looked for in TRIBITS_CONFIGURE_FILE() to add macros to add deprecated warnings to deprecated parts of a package.

TRIBITS_ADD_TEST()

Add a test or a set of tests for a single executable or command using CTest `ADD_TEST()`.

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 <numMpiProcs>]
  [NUM_TOTAL_CORES_USED <numTotalCoresUsed>]
  [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>]
  [ADDED_TESTS_NAMES_OUT <testsNames>]
  )
```

Sections:

- Formal Arguments (TRIBITS_ADD_TEST())
- Determining the Executable or Command to Run (TRIBITS_ADD_TEST())
- Determining the Full Test Name (TRIBITS_ADD_TEST())
- Adding Multiple Tests (TRIBITS_ADD_TEST())

- Determining Pass/Fail (TRIBITS_ADD_TEST())
- Setting additional test properties (TRIBITS_ADD_TEST())
- Running multiple tests at the same time (TRIBITS_ADD_TEST())
- Debugging and Examining Test Generation (TRIBITS_ADD_TEST())
- Disabling Tests Externally (TRIBITS_ADD_TEST())

Formal Arguments (TRIBITS_ADD_TEST())

`<exeRootName>`

The name of the executable or path to the executable to run for the test (see Determining the Executable 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 assumed **not** to be prepended to `<exeRootName>` (see Determining the Executable or Command to Run (TRIBITS_ADD_TEST())).

`NOEXESUFFIX`

If specified, then the postfix `${${PROJECT_NAME}_CMAKE_EXECUTABLE_SUFFIX}` is assumed **not** to be post-pended to `<exeRootName>` (see Determining the Executable or Command to Run (TRIBITS_ADD_TEST())).

`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 passed to `ADD_TEST()` will always be prefixed as `${PACKAGE_NAME}_<testRootName>`. 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. See Determining the Full Test Name (TRIBITS_ADD_TEST()).

`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. See Determining the Full Test Name (TRIBITS_ADD_TEST()).

`DIRECTORY <dir>`

If specified, then the executable is assumed to be in the directory given 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 binary directory `${CMAKE_CURRENT_BINARY_DIR}`. See Determining the Executable or Command to Run (TRIBITS_ADD_TEST()).

`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 Determining the Full Test Name (TRIBITS_ADD_TEST())). The directory name will have the package's base directory stripped off so only the unique part of the test directory will be used. All directory separators `"/"` 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 GPUs) 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 `<fullTestName>_xy` where `xy` = `00`, `01`, `02`, and so on. **WARNING:** When defining multiple tests it is preferred 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 one can give a meaningful name to each test case and one can specify multiple arguments without having to quote them and one 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, determines 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. That is if `COMM mpi` is passed in, then the test will **not** be added if `TPL_ENABLE_MPI=OFF`. Likewise, if `COMM serial` is passed in, then the test will **not** be added if `TPL_ENABLE_MPI=ON`. If `COMM serial mpi` or `COMM mpi serial` is passed in, then the value of `TPL_ENABLE_MPI` does not determine if the test is added or not.

`NUM_MPI_PROCS <numMpiProcs>`

If specified, gives the number of MPI processes used to run the test with the MPI exec program `${MPI_EXEC}`. If `<numMpiProcs>` is greater than `${MPI_EXEC_MAX_NUMPROCS}` then the test will be excluded. If not specified, then the default number of processes for an MPI build (i.e. `TPL_ENABLE_MPI=ON`) will be `${MPI_EXEC_DEFAULT_NUMPROCS}`. For serial builds (i.e. `TPL_ENABLE_MPI=OFF`), this argument is ignored. This will also be set as the built-in test property `PROCESSORS` if `NUM_TOTAL_CORES_USED` is not specified.

`NUM_TOTAL_CORES_USED <numTotalCoresUsed>`

If specified, gives the total number of processes or cores that is reported to CTest as the built-in CTest `PROCESSORS` property. If this is not specified, then `PROCESSORS` is specified by the argument `NUM_MPI_PROCS <numMpiProcs>`. This argument is used for test scripts/executables that use more cores than MPI processes (i.e. `<numMpiProcs>`) and its only purpose is to inform CTest and TriBITS of the maximum number of processes or cores that are used by the underlying test executable/script. When specified, if `<numTotalCoresUsed>` is greater than `${MPI_EXEC_MAX_NUMPROCS}`, then the test will not be added. Otherwise, the CTest property `PROCESSORS` is set to `<numTotalCoresUsed>` so that CTest knows how to best schedule the test w.r.t. other tests on a given number of available processes. See Running multiple tests at the same time (TRIBITS_ADD_TEST()).

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

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

match ${PROJECT_NAME}_TEST_CATEGORIES equal to CONTINUOUS, NIGHTLY, and WEEKLY. When CATEGORIES contains NIGHTLY it will match ${PROJECT_NAME}_TEST_CATEGORIES equal to NIGHTLY and WEEKLY. When CATEGORIES contains PERFORMANCE it will match ${PROJECT_NAME}_TEST_CATEGORIES=PERFORMANCE only.

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 (see Full Processing of TriBITS Project Files).

XHOST <host0> <host1> ...

If specified, gives a list of hostnames (see HOST argument) on which 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 exclusion list overrides the HOST inclusion list.

HOSTTYPE <hosttype0> <hosttype1> ...

If specified, gives the names of the host system type (given by the built-in CMake cache variable CMAKE_HOST_SYSTEM_NAME which is printed in the TriBITS cmake configure output in the section Probing the environment) for which the test is allowed to be added. If HOSTTYPE is specified and CMAKE_HOST_SYSTEM_NAME is not equal to one of the values of <hosttypei>, then the test will **not** be added. Typical host system type names include Linux, Darwain, Windows, etc.

XHOSTTYPE <hosttype0> <hosttype1> ...

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

STANDARD_PASS_OUTPUT

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

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

If specified, then the test will be assumed to pass only if one of the regular expressions <regex0>, <regex1> etc. match the output send to stdout. Otherwise, the test will fail. This is set using the built-in CTest property PASS_REGULAR_EXPRESSION. Consult standard CMake documentation for full behavior. TIPS: Replace ';' with '[;]' or CMake will interpretet this as a array eleemnt boundary. To match '.', use '[.]'.

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 send to stdout. Otherwise, the test will pass. This is set using the built-in CTest property FAIL_REGULAR_EXPRESSION. Consult standard CMake documentation for full behavior (and see above tips for PASS_REGULAR_EXPRESSION).

WILL_FAIL

If passed in, then the pass/fail criteria will be inverted. This is set using the built-in CTest property WILL_FAIL. Consult standard CMake documentation for full behavior.

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

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

`TIMEOUT <maxSeconds>`

If passed in, gives maximum number of seconds the test will be allowed to run before being timed-out. This sets the CTest property `TIMEOUT`. The value `<maxSeconds>` will be scaled by the value of ${PROJECT_NAME}_SCALE_TEST_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 extremely expensive tests). Expensive tests are one of the worse forms of technical debt that a project can have!

`ADDED_TESTS_NAMES_OUT <testsNames>`

If specified, then on output the variable `<testsNames>` will be set with the name(S) of the tests passed to `ADD_TEST()`. If more than one test is added, then this will be a list of test names. Having this name allows the calling `CMakeLists.txt` file access and set additional test propeties (see Setting additional test properties (TRIBITS_ADD_TEST())).

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 function does not provide any fundamentally new features that are not already available 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 Executable or Command to Run (TRIBITS_ADD_TEST())

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

By default, the executable to run is determined by first getting the executable name which by default is assumed to be:

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

which is (by no coincidence) identical to how it is selected in TRIBITS_ADD_EXECUTABLE(). This name can be altered 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 executable is to be run (i.e. not build inside of the project), then pass in `NOEXEPREFIX` and `NOEXESUFFIX` and set `<exeRootName>` to the relative or absolute path of the executable to be run. If `<exeRootName>` is not an absolute path, then `${CMAKE_CURRENT_BINARY_DIR}/<exeRootName>` is set as the executable to run in this case.

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>` above.

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 relative to the package base directory is added to the name as well to help disambiguate the test name (see the above).

Let the test name determined as described above be <fullTestName>. If no arguments or only a single set of arguments are passed in through ARGS, then this is the test name actually passed in to ADD_TEST(). If multiple tests are defined, then this name becomes the base test name for each of the tests (see Adding Multiple Tests (TRIBITS_ADD_TEST())).

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 executable arguments and can even add multiple tests in one of two ways. One can either pass in one 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, one will 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:

```
POSTFIX_AND_ARGS_0 long_args --this-is-the-first-long-arg=very
  --this-is-the-second-long-arg=verylong
```

If one does not use quotes when using ARGS one will actually get more than one test. For example, if one passes in:

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

one actually gets 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 CTest properties PASS_REGULAR_EXPRESSION and FAIL_REGULAR_EXPRESSION which can only grep the test's 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 one or more output files to determine pass/fail. Raw CMake/CTest cannot do this. In this case, one should use TRIBITS_ADD_ADVANCED_TEST() instead to add the 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_TESTS_PROPERTIES(). Therefore, any tests properties that are not directly supported and passed through this wrapper function can be set in the outer CMakeLists.txt file after the call to TRIBITS_ADD_TEST().

If tests are added, then the names of those tests will be returned in the variable ADDED_TESTS_NAMES_OUT <testsNames>. This can be used, for example, to override the PROCESSORS property for the tests with:

```
TRIBITS_ADD_TEST( someTest ...
  ADDED_TESTS_NAMES_OUT  someTest_TEST_NAME )

IF (someTest_TEST_NAME)
  SET_TESTS_PROPERTIES( ${someTest_TEST_NAME}
    PROPERTIES ATTACHED_FILES someTest.log )
ENDIF()
```

where the test writes a log file someTest.log that we want to submit to CDash also.

This approach will work no matter what TriBITS names the individual test(s) or whether the test(s) are added or not (depending on other arguments like COMM, XHOST, etc.).

There are many other test properties that one may want to set also and this is the way it needs to be done.

Running multiple tests at the same time (TRIBITS_ADD_TEST())

By default, CTest will run as many tests defined with ADD_TEST() at same time as it can according to its parallel level (e.g. 'ctest -j<N>' or the CTest property CTEST_PARALLEL_LEVEL). For example, when raw 'ctest -j10' is run, CTest will run multiple tests at the same time to try to make usage of 10 processors/cores. If all of the defined tests only used one process (which is assumed by default except for MPI tests), then CTest will run 10 tests at the same time and will launch new tests as running tests finish. One can also define tests which use more than one process or use more cores than the number of MPI processes. When passing in NUM_MPI_PROCS <numMpiProcs> (see above), this TriBITS function will set the built-in CTest property PROCESSORS to <numMpiProcs> using:

```
SET_TESTS_PROPERTIES(<fullTestName> PROPERTIES PROCESSORS <numMpiProcs>)
```

This tells CTest that the defined test uses <numMpiProcs> processes and CTest will use that information to not exceed the requested parallel level. For example, if several NUM_MPI_PROCS 3 tests are defined and CTest is run with 'ctest -j12', then CTest would schedule and run 4 of these tests at a time (to make use of 12 processors/cores on the machine), starting new tests as running tests finish, until all of the tests have been run.

There are some situations where a test will use more processes/cores than specified by NUM_MPI_PROCS <numMpiProcs> such as when the underlying executable fires off more processes in parallel to do processing. Also, an MPI program may use threading and therefore use overall more cores than the number of MPI processes. For these cases, it is critical to set NUM_TOTAL_CORES_USED <numTotalCoresUsed> to tell TriBITS and CTest how many cores will be used by a threaded test. This is needed to exclude the test if there are too many processes/cores needed to run the test than are available. Also, if the test is added, then this is needed to set the built-in CTest PROCESSORS property so CTest can avoid overloading the machine. For example, for test where the MPI executable running on 4 processes uses 10 threads per process, one would set:

```
NUM_MPI_PROCS  4  NUM_TOTAL_CORES_USED  40
```

In this case, it sets the CTest PROCESSORS property as:

```
SET_TESTS_PROPERTIES(<fullTestName> PROPERTIES PROCESSORS <numTotalCoresUsed>)
```

When the number of processes a test uses does not cleanly divide into the requested CTest parallel level, it is not clear how CTest schedules the tests (hard to find documentation on this but one could always inspect the CTest source code to

find out for sure). However, one boundary case that is well observed is that CTest will run all defined tests regardless of the size of the PROCESSORS property or the value of CTEST_PARALLEL_LEVEL. For example, if there are tests where PROCESSORS is set to 20 but 'ctest -j10' is used, then CTest will still run those tests (using 20 processes) but will not schedule any other tests while the parallel level is exceeded. This can overload the machine obviously. Therefore, always set MPI_EXEC_MAX_NUMPROCS to the maximum number of cores/processes that can be comfortably run on a given machine. Also note that MPI tests are very fragile to overloaded machines

NOTE: **Never** manually override the PROCESSORS CTest property. Instead, always using NUM_TOTAL_CORES_USED <numTotalCoresUsed> to set this. This is important because TriBITS needs to know how many processes/cores are required for test so that it can disable the test if the test requires more cores/processes than a given machine can handle or to exceed an imposed budget of the number processes to be used. The latter is important when running multiple ctest -J<N> invocations on the same test machine.

Debugging and Examining Test Generation (TRIBITS_ADD_TEST())

In order to see what tests get added and if not then why, configure with ${PROJECT_NAME}_TRACE_ADD_TEST=ON. That will print one line per show that the test got added and if not then why the test was not added (i.e. due to COMM, NUM_MPI_PROCS, CATEGORIES, HOST, XHOST, HOSTTYPE, or XHOSTTYPE).

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, determine pass/fail and adjust other behavior using test properties. In this file, one can see the exact ADD_TEST() and SET_TESTS_PROPERTIES() commands. The is the ultimate way to debug exactly what tests are getting added by this function (or if the test is even being added at all).

Disabling Tests Externally (TRIBITS_ADD_TEST())

The test can be disabled externally by setting the CMake cache variable <fullTestName>_DISABLE=TRUE. This allows tests to be disabled on a case-by-case basis by the user (for whatever reason). Here, <fullTestName> must be the *exact* name that shows up in 'ctest -N' when running the test. If multiple tests are added in this function through multiple argument sets to ARGS or through multiple POSTFIX_AND_ARGS_<IDX> arguments, then <fullTestName>_DISABLE=TRUE must be set for each test individually. When a test is disabled in this way, TriBITS will always print a warning to the cmake stdout at configure time warning that the test is being disabled.

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 CMakeLists.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, all this macro does macro 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 future in order to modify behavior related to adding tests and examples in a uniform way.

TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES()

Allow listed packages to be missing and automatically excluded from the package dependency data-structures.

Usage:

```
TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES(<pkg0> <plg1> ...)
```

If the missing upstream SE package <pkgi> is optional, then the effect will be to simply ignore the missing package (i.e. it will never be added to package's list and not added to dependency data-structures) and remove it from the dependency lists for downstream SE packages that have an optional dependency on the missing upstream SE package <pkgi>. However, all downstream SE packages that have a required dependency on the missing upstream SE package

<pkgi> will be hard disabled, i.e. ${PROJECT_NAME}_ENABLE_{CURRENT_PACKAGE}=OFF and a note on the disable will be printed.

WARNING: This macro just sets the cache variable <pkgi>_ALLOW_MISSING_EXTERNAL_PACKAGE=TRUE for each SE package <pkgi>. Therefore, 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 (use cases mentioned below). Also note that missing packages are silently ignored by default. Therefore, when doing development involving these packages, it is usually a good idea to set:

```
-D<pkgi>_ALLOW_MISSING_EXTERNAL_PACKAGE=FALSE
```

so that it will catch errors in the misspelling of package names or source directories. However, notes on what missing packages are being ignored can printed by configuring with:

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

This macro is typically called in one of two different contexts:

- Called from <packageDir>/cmake/Dependencies.cmake when the upstream package <pkgi> is defined in an optional upstream TriBITS Repository. This allows the downstream repos and packages to still be enabled (assuming they don't have required dependencies on the missing packages) when one or more upstream repos are missing.
- Called from <repoDir>/PackagesList.cmake when the package <pkgi> might be defined in an optional non-TriBITS repo (see How to insert a package into an upstream repo).

For some meta-projects that composes packages from may different TriBITS repositories, one might need to also call this function from the file <projectDir>/cmake/ProjectDependenciesSetup.cmake.

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 substitution of CMake variables (see documentation for built-in CMake CONFIGURE_FILE() command for rules on how it performs substitutions). This command is typically used to configure the package's main <packageDir>/cmake/<packageName>_config.h.in file.

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

Deprecated Code Macros

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

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

to the <packageConfigFile>.in file and it will be expanded at configure time.

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 source directory to a destination directory at configure time, typically so that it can be used in one or more tests.

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 sets up all of the custom CMake commands and targets to ensure that the files in the destination directory are always up to date just by building the ALL target.

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>` prepended 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 omitted, this list will be the same as `DEST_FILES` with the argument `SOURCE_PREFIX <srcPrefix>` appended.

`SOURCE_DIR <sourceDir>`

Optional argument that gives the (absolute) base directory for all of the source files. If omitted, 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 omitted, 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 omitted, 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. This results in the copies to be performed when any of the targets `<targDepi>` are built.

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

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

`NOEXEPREFIX`

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

TRIBITS_CTEST_DRIVER()

Platform-independent package-by-package CTest/CDash driver (run by `ctest` **NOT** `cmake`).

Usage:

```
TRIBITS_CTEST_DRIVER()
```

This driver code that is platform independent. This script drives the testing process by doing a version control (VC) source update on all of the VC repos and then configuring and building the top-level TriBITS packages one at a time, in order. This function gets called from inside of a platform and build-specific `ctest -S` driver script.

To understand this script, one must understand that it gets run in several different modes:

Mode 1: Run where there are already existing source and binary directories (`CTEST_DASHBOARD_ROOT` is set empty before call). This is for when the ctest driver script is run on an existing source and binary tree. In this case, there is one project source tree and `CTEST_SOURCE_DIRECTORY` and `CTEST_BINARY_DIRECTORY` must be set by the user before calling this function. This is used to test a local build and post to CDash.

Mode 2: A new binary directory is created and new sources are cloned (or updated) in a driver directory (`CTEST_DASHBOARD_ROOT` is set before call). In this case, there are always two (partial) project source tree's, i) a "driver" skeleton source tree (typically embedded with TriBITS directory) that bootstraps the testing process, and ii) a true full "source" that is (optionally) cloned and/or updated.

There are a few different directory locations are significant for this script:

`TRIBITS_PROJECT_ROOT`

The root directory to an existing source tree where the project's <projectDir>/ProjectName.cmake (defining `PROJECT_NAME` variable) and `Version.cmake` file's can be found.

`${PROJECT_NAME}_TRIBITS_DIR`

The base directory for the TriBITS system's various CMake modules, python scripts, and other files. By default this is assumed to be in the source tree under `${TRIBITS_PROJECT_ROOT}` (see below) but it can be overridden to point to any location.

`CTEST_DASHBOARD_ROOT`

If set, this is the base directory where this script runs that clones the sources for the project. If this directory does not exist, it will be created. If empty, then has no effect on the script.

`CTEST_SOURCE_DIRECTORY`

Determines the location of the sources that are used to define packages, dependencies and configure and build the software. This is a variable that CTest directly reads and must therefore be set. This is used to set PROJECT_SOURCE_DIR which is used by the TriBITS system. If `CTEST_DASHBOARD_ROOT` is set, then this is hard-coded internally to `${CTEST_DASHBOARD_ROOT}/${CTEST_SOURCE_NAME}`.

`CTEST_BINARY_DIRECTORY`

Determines the location of the binary tree where output from CMake/CTest is put. This is used to set to PROJECT_BINARY_DIR which is used by the TriBITS system. If `CTEST_DASHBOARD_ROOT` is set, then this is hard-coded internally to `${CTEST_DASHBOARD_ROOT}/BUILD`.

ToDo: Document input variables that have defaults, to be set before, and can be overridden from the env.

ToDo: Finish Documentation!

TRIBITS_DETERMINE_IF_CURRENT_PACKAGE_NEEDS_REBUILT()

Determine at configure time if any of the upstream dependencies for a package require the current package to be rebuilt.

Usage:

```
TRIBITS_DETERMINE_IF_CURRENT_PACKAGE_NEEDS_REBUILT(
  [SHOW_MOST_RECENT_FILES]
  [SHOW_OVERALL_MOST_RECENT_FILES]
  CURRENT_PACKAGE_OUT_OF_DATE_OUT <currentPackageOutOfDate>
  )
```

Arguments:

`SHOW_MOST_RECENT_FILES`

If specified, then the most recently modified file for each individual base source and binary directory searched will be will be printed the STDOUT. Setting this implies `SHOW_OVERALL_MOST_RECENT_FILE`.

`SHOW_OVERALL_MOST_RECENT_FILE`

If specified, then only the most recent modified file over all of the individual directories for each category (i.e. one for upstream SE package source dirs, one for upstream SE package binary dirs, one for the package's source dir, and one for the package's own binary dir) is printed to STDOUT.

`CURRENT_PACKAGE_OUT_OF_DATE_OUT <currentPackageOutOfDate>`

On output, the local variable `<currentPackageOutOfDate>` will be set to `TRUE` if any of the upstream most modified files are more recent than the most modified file in the package's binary directory. Otherwise, this variable is set to `FALSE`.

Description:

This function is designed to help take an externally configured and built piece of software (that generates libraries) and wrap it as a TriBITS package or subpackage. This function uses the lower-level functions:

- TRIBITS_FIND_MOST_RECENT_SOURCE_FILE_TIMESTAMP()
- TRIBITS_FIND_MOST_RECENT_BINARY_FILE_TIMESTAMP()

to determine the most recent modified files in the upstream TriBITS SE packages' source and binary directories as well as the most recent source file for the current package. It then compares these timestamps to the most recent binary file timestamp in this package's binary directory. If any of these three files are more recent than this package's most recent binary file, then the output variable `<currentPackageOutOfDate>` is set to `TRUE`. Otherwise, it is set to `FALSE`.

NOTE: The source and binary directories for full packages are searched, not individual subpackage dirs. This is to reduce the number of dirs searched. This will, however, result in changes in non-dependent subpackages being considered as well.

See the demonstration of the usage of this function in the `WrapExternal` package in TribitsExampleProject.

TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS()

Disable a package automatically for a list of platforms.

Usage:

```
TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS( <packageName>
  <hosttype0> <hosttype1> ...)
```

If any of the host-type arguments `<hosttypei>` matches the `${PROJECT_NAME}_HOSTTYPE` variable for the current platform, then package `<packageName>` test group classification is changed to `EX`. Changing the package test group classification to `EX` results in the package being disabled by default (see EX SE packages disabled by default). However, an explicit enable can still enable the package.

TRIBITS_EXCLUDE_FILES()

Exclude package files/dirs from the source distribution by appending `CPACK_SOURCE_IGNORE_FILES`.

Usage:

```
TRIBITS_EXCLUDE_FILES(<file0> <file1> ...)
```

This is called in the package's top-level <packageDir>/CMakeLists.txt file and each file or directory name `<filei>` is actually interpreted by CMake/CPack as a regex that is prefixed by the project's and package's source directory names so as to not exclude files and directories of the same name and path from other packages. If `<filei>` is an absolute path it it not prefixed but is appended to `CPACK_SOURCE_IGNORE_FILES` unmodified.

In general, do **NOT** put in excludes for files and directories that are not under this package's source tree. If the given package is not enabled, then this command will never be called! For example, don't put in excludes for PackageB's files in PackageA's `CMakeLists.txt` file because if PackageB is enabled but PackageA is not, the excludes for PackageB will never get added to `CPACK_SOURCE_IGNORE_FILES`.

Also, be careful to note that the `<filei>` arguments are actually regexes and one must be very careful not understand how CPack will use these regexes to match files that get excluded from the tarball. For more details, see Creating Source Distributions.

TRIBITS_FIND_MOST_RECENT_BINARY_FILE_TIMESTAMP()

Find the most modified binary file in a set of base directories and return its timestamp.

Usage:

```
TRIBITS_FIND_MOST_RECENT_BINARY_FILE_TIMESTAMP(
  BINARY_BASE_DIRS <dir0> <dir1> ...
  [BINARY_BASE_BASE_DIR <dir>]
  [MOST_RECENT_TIMESTAMP_OUT  <mostRecentTimestamp>]
  [MOST_RECENT_FILEPATH_BASE_DIR_OUT <mostRecentFilepathBaseDir>]
  [MOST_RECENT_RELATIVE_FILEPATH_OUT <mostRecentRelativeFilePath>]
  [SHOW_MOST_RECENT_FILES]
  [SHOW_OVERALL_MOST_RECENT_FILE]
  )
```

This function just calls TRIBITS_FIND_MOST_RECENT_FILE_TIMESTAMP() passing in a set of basic exclude regexes like `CMakeFiles/`, `[.]cmake$`, and `/Makefile$`, etc. These types of files usually don't impact the build of downstream software in CMake projects.

TRIBITS_FIND_MOST_RECENT_FILE_TIMESTAMP()

Find the most modified file in a set of base directories and return its timestamp.

Usage:

```
TRIBITS_FIND_MOST_RECENT_FILE_TIMESTAMP(
  BASE_DIRS <dir0> <dir1> ...
  [BASE_BASE_DIR <dir>]
  [EXCLUDE_REGEXES "<re0>" "<re1>" ...
  [SHOW_MOST_RECENT_FILES]
  [SHOW_OVERALL_MOST_RECENT_FILE]
  [MOST_RECENT_TIMESTAMP_OUT  <mostRecentTimestamp>]
  [MOST_RECENT_FILEPATH_BASE_DIR_OUT <mostRecentFilepathBaseDir>]
  [MOST_RECENT_RELATIVE_FILEPATH_OUT <mostRecentRelativeFilePath>]
  )
```

Arguments:

```
BASE_DIRS <dir0> <dir1> ...
```

Gives the absolute base directory paths that will be searched for the most recently modified files, as described above.

`BASE_BASE_DIR <dir>`

Absolute path for which to print file paths relative to. This makes outputting less verbose and easier to read (optional).

`EXCLUDE_REGEXES "<re0>" "<re1>" ...`

Gives the regular expressions that are used to exclude files from consideration. Each "<rei>" regex is used with a *grep -v "<rei>"* filter to exclude files before sorting by time stamp.

`SHOW_MOST_RECENT_FILES`

If specified, then the most recently modified file for each individual directory `<dir0>`, `<dir1`, ... will be printed the STDOUT. Setting this implies `SHOW_OVERALL_MOST_RECENT_FILE`.

`SHOW_OVERALL_MOST_RECENT_FILE`

If specified, then only the most recent modified file over all of the individual directories is printed to STDOUT.

`MOST_RECENT_TIMESTAMP_OUT <mostRecentTimestamp>`

On output, the variable *<mostRecentTimestamp>* is set that gives the timestamp of the most recently modified file over all the directories. This number is given as the number of seconds since Jan. 1, 1970, 00:00 GMT.

`MOST_RECENT_FILEPATH_BASE_DIR_OUT <mostRecentFilepathBaseDir>`

On output, the variable *<mostRecentFilepathBaseDir>* gives absolute base directory of the file with the most recent timestamp over all directories.

`MOST_RECENT_RELATIVE_FILEPATH_OUT <mostRecentRelativeFilePath>`

On output, the variable *<mostRecentFilepathBaseDir>* gives the file name with relative path to the file with the most recent timestamp over all directories.

Description:

This function uses the Linux/Unix command:

```
$ find . -type f -printf '%T@ %p\n'
    | grep -v "<re0>" | grep -v "<re1>" | ... \
    | sort -n | tail -1
```

to return the most recent file in each listed directory <dir0>, <dir1>, etc. It then determines the most recently modified file over all of the directories and prints and returns in the variables *<mostRecentTimestamp>*, *<mostRecentFilepathBaseDir>*, and *<mostRecentRelativeFilePath>*.

TRIBITS_FIND_MOST_RECENT_SOURCE_FILE_TIMESTAMP()

Find the most modified source file in a set of base directories and return its timestamp.

Usage:

```
TRIBITS_FIND_MOST_RECENT_SOURCE_FILE_TIMESTAMP(
  SOURCE_BASE_DIRS <dir0> <dir1> ...
  [SOURCE_BASE_BASE_DIR <dir>]
  [SHOW_MOST_RECENT_FILES]
  [SHOW_OVERALL_MOST_RECENT_FILE]
  [MOST_RECENT_TIMESTAMP_OUT  <mostRecentTimestamp>]
  [MOST_RECENT_FILEPATH_BASE_DIR_OUT <mostRecentFilepathBaseDir>]
  [MOST_RECENT_RELATIVE_FILEPATH_OUT <mostRecentRelativeFilePath>]
  )
```

This function just calls TRIBITS_FIND_MOST_RECENT_FILE_TIMESTAMP() passing in a set of basic exclude regexes like `[.]git/`, `[.]svn/`, etc. These types of version control files can not possibly directly impact the source code.

TRIBITS_INSTALL_HEADERS()

Function used to (optionally) install header files using `INSTALL()` command.

Usage:

```
TRIBITS_INSTALL_HEADERS(
  HEADERS <h0> <h1> ...
  [INSTALL_SUBDIR <subdir>]
  [COMPONENT <component>]
  )
```

The formal arguments are:

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

List of header files to install. 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.

`INSTALL_SUBDIR <subdir>`

Optional subdirectory that the headers will be installed under the standard installation directory. If `<subdir>!=""`, then the headers will be installed under `${PROJECT_NAME}_INSTALL_INCLUDE_DIR}/<subdir>`. Otherwise, they will be installed under `${PROJECT_NAME}_INSTALL_INCLUDE_DIR}/`.

`COMPONENT <component>`

If specified, then `COMPONENT <component>` will be passed into `INSTALL()`. Otherwise, `COMPONENT ${PROJECT_NAME}` will get used.

If ${PROJECT_NAME}_INSTALL_LIBRARIES_AND_HEADERS is `FASLE`, then the headers will not get installed.

TRIBITS_INCLUDE_DIRECTORIES()

This function is to override the standard behavior of the built-in CMake `INCLUDE_DIRECTORIES()` command.

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 of `INCLUDE_DIRECTORIES()` for installation testing, to ensure that include directories will not be inadvertently added to the build lines for tests during installation testing (see Installation and Backward Compatibility 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 `REQUIRED_DURING_INSTALLATION_TESTING` must be passed in to ensure the include paths are added for installation testing.

TRIBITS_PACKAGE()

Macro called at the very beginning of a package's top-level <packageDir>/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 variables set) after calling this macro.

TRIBITS_PACKAGE_DECL()

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

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 must match the name of the package provided in the <repoDir>/PackagesList.cmake or an error is issued.

`ENABLE_SHADOWING_WARNINGS`

If specified, then shadowing warnings for the package's sources 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 for the package's sources 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 compiling the package's sources for all defined warnings.

`DISABLE_CIRCULAR_REF_DETECTION_FAILURE`

If specified, then the standard grep looking for RCPNode circular references in TRIBITS_ADD_TEST() and TRIBITS_ADD_ADVANCED_TEST() 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 variables `${PACKAGE_NAME}_LIB_TARGETS` (lists all of the package's targets) and `${PACKAGE_NAME}_ALL_TARGETS` (lists all of the package's libraries) and are initialized to empty.

- The local variables `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-scope (i.e., the package's 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.

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

TRIBITS_PACKAGE_DEF()

Macro called in <packageDir>/CMakeLists.txt after subpackages are processed in order to handle the libraries, tests, and examples of the parent 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 variable `PACKAGE_NAME` is set in the local scope for usage by the package's `CMakeLists.txt` files.
- The intra-package dependency variables (i.e. list of include directories, list of libraries, etc.) are initialized to empty.

TRIBITS_PACKAGE_DEFINE_DEPENDENCIES()

Define the dependencies for a given TriBITS SE Package (i.e. a top-level TriBITS Package or a TriBITS Subpackage) in the package's <packageDir>/cmake/Dependencies.cmake file.

Usage:

```
TRIBITS_PACKAGE_DEFINE_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 (that is, an SE package can have no upstream dependencies). The arguments that apply to all SE packages are:

`LIB_REQUIRED_PACKAGES`

> List of required upstream SE packages that must be enabled in order to build and use the libraries (or capabilities) in this SE package.

`LIB_OPTIONAL_PACKAGES`

List of additional optional upstream SE packages that can be used in this SE package if enabled. These upstream SE packages need not be enabled in order to enable this SE package but not enabling one or more of these optional upstream SE packages will result in diminished capabilities of this SE package.

`TEST_REQUIRED_PACKAGES`

List of additional upstream SE packages that must be enabled in order to build and/or run the tests and/or examples in this SE package. If any of these upstream SE packages are not enabled, then there will be no tests or examples defined or run for this SE package.

`TEST_OPTIONAL_PACKAGES`

List of additional optional upstream SE packages that can be used by the tests in this SE package. These upstream SE packages need not be enabled in order to run some basic tests or examples for this SE package. Typically, extra tests that depend on optional test SE packages involve integration testing of some type.

`LIB_REQUIRED_TPLS`

List of required upstream TPLs that must be enabled in order to build and use the libraries (or capabilities) in this SE package.

`LIB_OPTIONAL_TPLS`

List of additional optional upstream TPLs that can be used in this SE package if enabled. These upstream TPLs need not be enabled in order to use this SE package but not enabling one or more of these optional upstream TPLs will result in diminished capabilities of this SE 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 SE package. If any of these upstream TPLs are not enabled, then there will be no tests or examples defined or run for this SE package.

`TEST_OPTIONAL_TPLS`

List of additional optional upstream TPLs that can be used by the tests in this SE package. These upstream TPLs need not be enabled in order to run basic tests for this SE package. Typically, extra tests that depend on optional TPLs involve integration testing or some additional testing of some type.

Only upstream SE packages can be listed (as defined by the order the SE packages are listed in TRIBITS_REPOSITORY_DEFINE_PACKAGES() in the <repoDir>/PackagesList.cmake file). Otherwise an error will occur and processing will stop. Misspelled SE package names are caught as well.

Only direct SE package dependencies need to be listed. Indirect SE package dependencies are automatically handled. For example, if this SE package directly depends on SE package `PKG2` which depends on SE package `PKG1` (but this SE package does not directly depend on anything in `PKG1`) then this SE package only needs to list a dependency on `PKG2`, not `PKG1`. The dependency on `PKG1` will be taken care of automatically by the TriBITS dependency management system.

However, currently, all TPL dependencies must be listed, even the indirect ones. This is a requirement that will be dropped in a future version of TriBITS.

The SE packages listed in `LIB_REQUIRED_PACKAGES` are implicitly also dependencies in `TEST_REQUIRED_PACKAGES`. Likewise `LIB_OPTIONAL_PACKAGES` are implicitly also dependencies in `TEST_OPTIONAL_PACKAGES`. Same goes for TPL dependencies.

The upstream dependencies within a single list do not need to be listed in any order. For example if `PKG2` depends on `PKG1`, and this given SE package depends on both, then one can list:

```
LIB_REQUIRED_PACKAGES PKG2 PKG1
```

or:

```
"LIB_REQUIRED_PACKAGES PKG1 PKG2
```

Likewise the order that dependent TPLs are listed is not significant.

If some upstream SE packages are allowed to be missing, this can be specified by calling the macro TRIBITS_ALLOW_MISSING_EXTERNAL_PACKAGES().

A top-level TriBITS Package can also be broken down into TriBITS Subpackages. In this case, the following argument must be passed in:

`SUBPACKAGES_DIRS_CLASSIFICATIONS_OPTREQS`

2D array with rows listing the subpackages where each row has the columns:

- **SUBPACKAGE** (Column 0): 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 dependencies in other SE packages.
- **DIRS** (Column 1): 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 (see checkin-test.py).
- **CLASSIFICATIONS** (Column 2): The Test Test Category 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 same meaning as for full packages (see TRIBITS_REPOSITORY_DEFINE_PACKAGES()).
- **OPTREQ** (Column 3): 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 argument is missing, then the email list that CDash errors go to is determined by other means (see CDash regression email addresses).

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

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

which are then read by the TriBITS cmake code to build the SE package dependency graph. The advantage of using this macro instead of just directly setting the variables is that an SE package only needs to list dependencies that exist. Otherwise, the `Dependencies.cmake` file will need to set all of the above local variables, even those that are empty. This is a error checking property of the TriBITS system to avoid misspelling the names of these variables.

TRIBITS_PACKAGE_POSTPROCESS()

Macro called at the very end of a package's top-level <packageDir>/CMakeLists.txt file that performs some critical post-processing activities.

Usage:

```
TRIBITS_PACKAGE_POSTPROCESS()
```

NOTE: It is unfortunate that this macro must be called in a packages's top-level `CMakeLists.txt` file but limitations of the CMake language make it necessary to do so.

TRIBITS_PROCESS_SUBPACKAGES()

Macro that processes the TriBITS Subpackages for a parent TriBITS package for packages that are broken down into subpackages. This is called in the parent packages top-level <packageDir>/CMakeLists.txt file.

Usage:

```
TRIBITS_PROCESS_SUBPACKAGES()
```

This macro must be called after TRIBITS_PACKAGE_DECL() but before TRIBITS_PACKAGE_DEF().

TRIBITS_PROCESS_ENABLED_TPL()

Processs an enabled TPL's FindTPL${TPL_NAME}.cmake module.

TRIBITS_PROJECT()

Processes a TriBITS Project's files and configures its software which is called from the project's top-level <projectDir>/CMakeLists.txt file.

Usage:

```
TRIBITS_PROJECT()
```

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

This macro then adds all of the necessary paths to `CMAKE_MODULE_PATH` and then performs all processing of the TriBITS project files (see Full TriBITS Project Configuration).

TRIBITS_PROJECT_DEFINE_EXTRA_REPOSITORIES()

Declare a set of extra repositories for the TriBITS Project (i.e. in the project's <projectDir>/cmake/ExtraRepositoriesList.cmake file).

Usage:

```
TRIBITS_PROJECT_DEFINE_EXTRA_REPOSITORIES(
  <repo0_name> <repo0_dir> <repo0_vctype> <repo0_url> <repo0_packstat> <repo0_classif>
  <repo1_name> <repo1_dir> <repo1_vctype> <repo1_url> <rep10_packstat> <repo1_classif>
  ...
 )
```

This macro takes in a 2D array with 6 columns, where each row defines an extra repository. The 6 columns (ordered 0-5) are:

0. **REPO_NAME** (`<repoi_name>`): The name given to the repository `REPOSITORY_NAME`.

1. **REPO_DIR** (<repoi_dir>): The relative directory for the repository under the project directory ${PROJECT_SOURCE_DIR} (or <projectDir>). If this is set to empty quoted string "", then the relative directory name is assumed to be same as the repository name <repoi_name>. NOTE: If the repo is a TriBITS Repository (see REPO_PACKSTAT below) then, currently, the repo dir must be the same as the package name.
2. **REPO_VCTYPE** (<repoi_vctype>): The version control (VC) type of the repo. Value choices include GIT, SVN (i.e. Subversion) or empty "" (in which case <repoi_url> must be empty as well). WARNING: Only VC repos of type GIT can fully participate in the TriBITS development tool workflows. The other VC types are only supported for basic cloning and updating using TRIBITS_CTEST_DRIVER() scripts.
3. **REPO_URL** (<repoi_url>): The URL of the VC repo. This info is used to initially obtain the repo source code using the VC tool listed in <repoi_vctype>. If the repo does not need to be cloned for the needed use cases, then this can be the empty quoted string "". Also, this field must be the empty string "" if <repoi_vctype> is empty "".
4. **REPO_PACKSTAT** (<repoi_packstat>): Determines if the repository is a TriBITS Repository and contains any TriBITS packages (or if it just provides directories and files) and if its packages are listed before or after the project's own native packages. If this is a TriBITS Repository (and therefore contains TriBITS Repository Core Files) then this field must contain the keyword HASPACKAGES or left empty. If the listed repository is **not** a TriBITS repository, and just provides directories and files, then this field must contain the keyword NOPACKAGES. The default is assumed to be HASPACKAGES if neither of these keywords are provided. If the keyword PRE is provided, then the TriBITS packages in this repo come before the project's native packages. If the keyword POST is provided then the packages are listed after the project's native packages. The default is assumed to be POST if neither of these keywords are provided. The keywords must be separated by a comm with no spaces such as with "PRE,HASPACKAGES", "HASPACKAGES,POST", "POST,NOPACKAGES", etc. If no keywords are provided, then the empty string "" must be used (which defaults to "HASPACKAGES,POST").
5. **REPO_CLASSIFICATION** (<repoi_classif>): Gives the Repository Test Classification which also happens to be the CTest/CDash testing mode and the default dashboard track. Valid values are Continuous, Nightly, and Experimental. See Repository Test Classification for a detailed description.

This command is used to put together one or more VC and/or TriBITS repositories to construct a composite TriBITS Project. The option <Project>_EXTRAREPOS_FILE is used to point to files that call this macro.

Repositories with <repoi_packstat>=NOPACKAGES are **not** TriBITS Repositories and are technically not considered at all during the basic configuration of the a TriBITS project. They are only listed in this file so that they can be used in the version control logic for tools that perform version control with the repositories (such as getting git versions, cloning, updating, looking for changed files, etc.). For example, a non-TriBITS repo can be used to grab a set of directories and files that fill in the definition of a package in an upstream repository (see How to insert a package into an upstream repo). Also, non-TriBITS repos can be used to provide extra test data for a given package or a set of packages so that extra tests can be run.

Repositories with <repoi_repotype>='' are not VC repos. This can be used, for example, to represent the project's native repos or it can be used to point to a TriBITS repository that was cloned in an early listed VC repo.

NOTE: These repositories must be listed in the order of package dependencies. That is, all of the packages listed in repository i must have upstream TPL and SE package dependencies listed before this package in this repository or in upstream repositories i-1, i-2, etc.

NOTE: This module just sets the local variable:

```
${PROJECT_NAME}_EXTRAREPOS_DIR_VCTYPE_REPOURL_PACKSTAT_CATEGORY
```

in the current scope. The advantages of using this macro instead of directly setting this variable are that the macro:

- Asserts that the variable PROJECT_NAME is defined and set.
- Avoids misspelling the name of the variable ${PROJECT_NAME}_EXTRAREPOS_DIR_VCTYPE_REPOURL_PACKSTAT_CATEGORY. If one misspells the name of a macro, it is an immediate error in CMake. A misspelled set variable is just ignored.
- The variable name can change in the future as an implementation detail.

TRIBITS_PROJECT_ENABLE_ALL()

Process a project where you enable all of the packages by default.

Usage:

```
TRIBITS_PROJECT_ENABLE_ALL()
```

This macro just sets the global cache var ${PROJECT_NAME}_ENABLE_ALL_PACKAGES to `ON` by default then calls TRIBITS_PROJECT(). That is all. This macro is generally used for TriBITS projects that have just a single package or by default just want to enable all packages. This is especially useful when you have a TriBITS project with just a single package.

TRIBITS_REPOSITORY_DEFINE_PACKAGES()

Define the set of packages for a given TriBITS Repository. This macro is typically called from inside of a <repoDir>/PackagesList.cmake file for a given TriBITS repo.

Usage:

```
TRIBITS_REPOSITORY_DEFINE_PACKAGES(
   <pkg0>  <pkg0_dir>  <pkg0_classif>
   <pkg1>  <pkg1_dir>  <pkg1_classif>
   ...
   )
```

This macro sets up a 2D array of `NumPackages` by `NumColumns` listing out the packages for a TriBITS repository. Each row (with 3 column entries) specifies a package which contains the columns (ordered 0-2):

0. **PACKAGE** (`<pkgi>`): The name of the TriBITS package. This name must be unique across all other TriBITS packages in this or any other TriBITS repo that might be combined into a single TriBITS project meta-build (see Globally unique TriBITS package names). The name should be a valid identifier (e.g. matches the regex `[a-zA-Z_][a-zA-Z0-9_]*`). The package names tend to used mixed case (e.g. `'SomePackge` not `SOMEPACKGE`).
1. **DIR** (`<pkgi_dir>`): The relative directory for the package `<packageDir>`. This directory is relative to the TriBITS repository base directory `<repoDir>`. Under this directory will be a package-specific `cmake/` directory with the file <packageDir>/cmake/Dependencies.cmake and a base-level <packageDir>/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). For details, see checkin-test.py.
2. **CLASSIFICATION** (`<pkgi_classif>`): Gives the SE Package Test Group PT, ST, or EX and the maturity level `EP`, `RS`, `PG`, `PM`, `GRS`, `GPG`, `GPM`, `UM`. These are separated by a coma with no space in between such as `"RS,PT"` for a "Research Stable", "Primary Tested" package. No spaces are allowed so that CMake treats this a one field in the array. The maturity level can be left off in which case it is assumed to be `UM` for "Unspecified Maturity". This classification for individual packages can be changed to `EX` for specific platforms by calling TRIBITS_DISABLE_PACKAGE_ON_PLATFORMS().

IMPORTANT: The packages must be listed in increasing order of package dependencies. That is No circular dependencies of any kind are allowed (see the *ADP (Acyclic Dependencies Principle)* in Software Engineering Packaging Principles). Package `i` can only list dependencies (in <packageDir>/cmake/Dependencies.cmake) for packages listed before this package in this list (or in upstream TriBITS repositories). This avoids an expensive package sorting algorithm and makes it easy to flag packages with circular dependencies or misspelling of package names.

NOTE: For some rare use cases, the package directory `<pkgi_dir>` is allowed to be specified as an absolute directory but this absolute directory must be a subdirectory of the project source base directory given by PROJECT_SOURCE_DIR. If not, `MESSAGE(FATAL_ERROR ...)` is called and processing stops immediately.

NOTE: This macro just sets the variable:

```
${REPOSITORY_NAME}_PACKAGES_AND_DIRS_AND_CLASSIFICATIONS
```

in the current scope. The advantages of using this macro instead of directly setting this variable are that the macro:

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

TRIBITS_REPOSITORY_DEFINE_TPLS()

Define the list of TriBITS TPLs for a given TriBITS Repository which includes the TPL name, find module, and classification . This macro is typically called from inside of the repository's <repoDir>/TPLsList.cmake file.

Usage:

```
TRIBITS_REPOSITORY_DEFINE_TPLS(
  <tpl0_name>   <tpl0_findmod>  <tpl0_classif>
  <tpl1_name>   <tpl1_findmod>  <tpl1_classif>
  ...
  )
```

This macro sets up a 2D array of `NumTPLS` by `NumColumns` listing out the TriBITS TPLs for a TriBITS Repository. Each row (with 3 entries) specifies a TPL which contains the columns (ordered 0-2):

0. **TPL** (`<tpli_name>`): The name of the TriBITS TPL `<tplName>`. 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 (see Globally unique TriBITS TPL names). However, a TPL can be redefined from an upstream repo (see below). The name should be a valid identifier (e.g. matches the regex `[a-zA-Z_][a-zA-Z0-9_]*`). TPL names typically use mixed case (e.g. `SomeTpl` and not `SOMETPL`).
1. **FINDMOD** (`<tpli_findmod>`): The relative path for the find module, usually with the name `FindTPL<tplName>.cmake`. This path is relative to the repository base directory. If just the base path for the find module is given, ending with `"/"` (e.g. `"cmake/tpls/"`), then the find module will be assumed to be under that this directory with the standard name (e.g. `cmake/tpls/FindTPL<tplName>.cmake`). A standard way to write a `FindTPL<tplName>.cmake` module is to use the function TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES().
2. **CLASSIFICATION** (`<pkgi_classif>`): Gives the SE Package Test Group PT, ST, or EX and the maturity level `EP`, `RS`, `PG`, `PM`, `GRS`, `GPG`, `GPM`, `UM`. These are separated 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 in a downstream repo, which allows redefining the find module that is used to specify the TPL. This allows downstream repos to add additional requirements for a given TPL (i.e. add more libraries, headers, etc.). 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 variable:

```
${REPOSITORY_NAME}_TPLS_FINDMODS_CLASSIFICATIONS
```

in the current scope. The advantages of using this macro instead of directly setting this variable are that the macro:

- Asserts that the variable `REPOSITORY_NAME` is defined and set
- Avoids having to hard-code the assumed repository name `${REPOSITORY_NAME}`. This provides more flexibility for how other TriBITS projects choose to name a given TriBITS repo (i.e. the name of repo subdirs).
- Avoids misspelling the name of the variable `${REPOSITORY_NAME}_TPLS_FINDMODS_CLASSIFICATIONS`. If one misspells the name of a macro, it is an immediate error in CMake.

TRIBITS_SET_ST_FOR_DEV_MODE()

Function that allows packages to easily make a feature `ST` for development builds and `PT` for release builds by default.

Usage:

```
TRIBITS_SET_ST_FOR_DEV_MODE(<outputVar>)
```

This function is typically called in a package's top-level <packageDir>/CMakeLists.txt file before defining other options for the package. The output variable `${<outputVar>}` is set to `ON` or `OFF` based on the configure state. In development mode (i.e. `${PROJECT_NAME}_ENABLE_DEVELOPMENT_MODE==ON`), `${<outputVar>}` will be set to `ON` only if `ST` code is enabled (i.e. `${PROJECT_NAME}_ENABLE_SECONDARY_TESTED_CODE==ON`), otherwise it is set to `OFF`. In release mode (i.e. `${PROJECT_NAME}_ENABLE_DEVELOPMENT_MODE==OFF`), `${<outputVar>}` is always set to `ON`. This allows some parts of a TriBITS package to be considered `ST` for development mode (thereby reducing testing time by not enabling the dependent features/tests), while still having important functionality available to users by default in a release of the package.

TRIBITS_SUBPACKAGE()

Forward declare a TriBITS Subpackage called at the top of the subpackage's <packageDir>/<spkgDir>/CMakeLists.txt file.

Usage:

```
TRIBITS_SUBPACKAGE(<spkgName>)
```

Once called, the following local variables are in scope:

`PARENT_PACKAGE_NAME`

The name of the parent package.

`SUBPACKAGE_NAME`

The local name of the subpackage (does not contain the parent package name).

`SUBPACKAGE_FULLNAME`

The full project-level name of the subpackage (which includes the parent package name at the beginning, `${PARENT_PACKAGE_NAME}${SUBPACKAGE_NAME}`).

`PACKAGE_NAME`

Inside the subpackage, the same as `SUBPACKAGE_FULLNAME`.

TRIBITS_SUBPACKAGE_POSTPROCESS()

Macro that performs standard post-processing after defining a TriBITS Subpackage which is called at the bottom of a subpackage's <packageDir>/<spkgDir>/CMakeLists.txt file.

Usage:

```
TRIBITS_SUBPACKAGE_POSTPROCESS()
```

NOTE: It is unfortunate that a Subpackages's CMakeLists.txt file must call this macro but limitations of the CMake language make it necessary to do so.

TRIBITS_TPL_ALLOW_PRE_FIND_PACKAGE()

Function that determines if a TriBITS find module file `FindTPL<tplName>.cmake` is allowed to call `FIND_PACKAGE(<tplName> ...)` before calling TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES().

Usage:

```
TRIBITS_TPL_ALLOW_PRE_FIND_PACKAGE( <tplName>
  <allowPackagePrefindOut> )
```

The required arguments are:

`<tplName>` : The input name of the TriBITS TPL (e.g. `HDF5`).

`<allowPackagePrefindOut>` : Name of a variable which will be set to `TRUE` on output if `FIND_PACKAGE(<tplName> ...)` should be called to find the TPL `<tplName>` or `FALSE` if it should not be called.

This function will set `<allowPackagePrefindOut>` to `FALSE` if any of the variables `TPL_<tplName>_INCLUDE_DIRS`, `${TPL_<tplName>_LIBRARIES`, or `TPL_<tplName>_LIBRARY_DIRS` are set. This allows the user to override the search for the library components and just specify the absolute locations. The function will also set `<allowPackagePrefindOut>` to `FALSE` if `<tplName>_INCLUDE_DIRS`, `<tplName>_LIBRARY_NAMES`, or `<tplName>_LIBRARY_DIRS` is set and `<tplName>_FORCE_PRE_FIND_PACKAGE` is set to `FALSE`. Otherwise, if `<tplName>_FORCE_PRE_FIND_PACKAGE` is set to `TRUE`, the function will not return `FALSE` for `<allowPackagePrefindOut>` no matter what the values of `<tplName>_INCLUDE_DIRS`, `<tplName>_LIBRARY_NAMES`, or `<tplName>_LIBRARY_DIRS`.

The variable `<tplName>_FORCE_PRE_FIND_PACKAGE` is needed to allow users (or the `FindTPL<tplName>.cmake` module itself) to avoid name clashes with the variables `<tplName>_INCLUDE_DIRS` or `<tplName>_LIBRARY_DIRS` in the usage of `FIND_PACKAGE(<tplName> ...)` because a lot of default `Find<tplName>.cmake` modules also use these variables. This function sets `<tplName>_FORCE_PRE_FIND_PACKAGE` as a cache variable with default value `FALSE` to maintain backward compatibility with existing `FindTPL<tplName>.cmake` modules.

See How to use FIND_PACKAGE() for a TriBITS TPL for details in how to use this function to create a `FindTPL<tplName>.cmake` module file.

TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES()

Function that sets up cache variables for users to specify where to find a TriBITS TPL's headers and libraries. This function is typically called inside of a `FindTPL<tplName>.cmake` moulde file (see ${TPL_NAME}_FINDMOD).

Usage:

```
TRIBITS_TPL_FIND_INCLUDE_DIRS_AND_LIBRARIES(
  <tplName>
  [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 be called to specify/require header files and include directories and/or a list of libraries.

The input arguments to this function are:

`<tplName>`

Name of the TPL that is listed in a <repoDir>/TPLsList.cmake file.

`REQUIRED_HEADERS`

List of header files that are searched in order to find the TPL's include directories files 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_<tplName>_INCLUDE_DIRS` to be defined.

`REQUIRED_LIBS_NAMES`

List of libraries that are searched for when looking for the TPL's libraries using `FIND_LIBRARY()`. This list can be overridden by the user by setting `<tplName>_LIBRARY_DIRS` (see below).

`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

This function implements the TPL find behavior described in Enabling support for an optional Third-Party Library (TPL).

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

`<tplName>_INCLUDE_DIRS` (type `PATH`)

List of paths to search first for header files defined in `REQUIRED_HEADERS`.

`<tplName>_LIBRARY_DIRS` (type `PATH`)

The list of directories to search first for libraries defined in `REQUIRED_LIBS_NAMES`. If, for some reason, no libraries should be linked in for this particular configuration, then setting `<tplName>_LIBRARY_DIRS=OFF` will

`<tplName>_LIBRARY_NAMES` (type `STRING`)

List of library names to be looked for instead of what is specified in `REQUIRED_LIBS_NAMES`.

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

`TPL_<tplName>_INCLUDE_DIRS` (type `PATH`)

A list of common-separated full directory paths that contain the TPL's header files. If this variable 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_<tplName>_LIBRARIES` (type `FILEPATH`)

A list of commons-separated full library names (i.e. output from `FIND_LIBRARY()`) for all of the libraries found for the TPL. If this variable is set before calling this function, then no libraries are searched for and this variable will be assumed to have the correct list of libraries to link to.

`TPL_<tplName>_NOT_FOUND` (type `BOOL`)

Will be set to `ON` if all of the parts of the TPL could not be found.

ToDo: Document the behavior of this function for finding headers and libraries and when a find is successful and when it is not.

Note, if `TPL_TENTATIVE_ENABLE_<tplName>=ON`, then if all of the parts of the TPL can't be found, then `TPL_ENABLE_<tplName>` will be (forced) set to `OFF` in the cache. See TRIBITS_TPL_TENTATIVELY_ENABLE().

TRIBITS_TPL_TENTATIVELY_ENABLE()

Function that sets up for an optionally enabled TPL that is attempted to be enabled but will be disabled if all of the parts are not found.

Usage:

```
TRIBITS_TPL_TENTATIVELY_ENABLE(<tplName>)
```

This function can be called from any CMakeLists.txt file to put a TPL in tentative enable mode.

This should only be used for optional TPLs. It will not work correctly for required TPLs because any enabled packages that require this TPL will not be disabled and instead will fail to configure or fail to build.

All this function does is to force set `TPL_ENABLE_<tplName>=ON` if it has not already been set. and `TPL_TENTATIVE_ENABLE_<tplName>` in the cache.

NOTE: This function will only tentatively enable a TPL it its enable has not be explicitly set on input, i.e. if `TPL_ENABLE_<tplName>=""`.

TRIBITS_VERBOSE_PRINT_VAR()

print a variable giving its name then value if `${PROJECT_NAME}_VERBOSE_CONFIGURE=TRUE`.

Usage:

```
TRIBITS_VERBOSE_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.

TRIBITS_WRITE_FLEXIBLE_PACKAGE_CLIENT_EXPORT_FILES()

Utility function for writing `${PACKAGE_NAME}Config.cmake` and/or the `Makefile.export.${PACKAGE_NAME}` files for package `${PACKAGE_NAME}` with some greater flexibility than what is provided by the function `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_MAKEFILE <exportMakefileFileFullPath>]
  [WRITE_INSTALL_CMAKE_CONFIG_FILE]
  [WRITE_INSTALL_EXPORT_MAKEFILE]
  )
```

The arguments are:

`PACKAGE_NAME <pakageName>`

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

`EXPORT_FILE_VAR_PREFIX <exportFileVarPrefix>`

If specified, then all of the variables in the generated export files will be prefixed with `<exportFileVarPrefix>_` instead of `<pakageName>_`.

`WRITE_CMAKE_CONFIG_FILE <cmakeConfigFileFullPath>`

If specified, then the package's (`<packageName>`) cmake configure export file for use by external CMake client projects will be created as the file `<cmakeConfigFileFullPath>`. NOTE: the argument should be the full path!

`WRITE_EXPORT_MAKEFILE <exportMakefileFileFullPath>`

If specified, then the package's (`<packageName>`) export makefile for use by 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's (`<packageName>`) install cmake configured export file will be installed in to the install tree as well. The name and location of this file is hard-coded.

`WRITE_INSTALL_EXPORT_MAKEFILE`

If specified, then the package's (`<packageName>`) install export makefile to be installed into the install tree as well. 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 very specialized use cases such as when a TriBITS package needs to generate an export makefile for a given package but the name of the export makefile must be different and use different variable name prefixes. The particular 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 `INSTALL()` commands because CMake will not allow those to even be present in scripting mode that is used for unit testing this function. Instead, the files to be installed are only generated in the build tree and the install targets are added else where.

12.3 General Utility Macros and Functions

The following subsections give detailed documentation for some CMake macros and functions which are *not* a core part of the TriBITS system but are included in the TriBITS source tree, 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 different projects that use TriBITS.

These macros and functions are *not* prefixed with `TRIBITS_`. However, there is really not a large risk to defining and using these non-namespaces utility functions and macros. It turns out that CMake allows one to redefine any macro or function, even built-in ones, inside of one's project. Therefore, even if CMake did add new commands that clashed with these names, there would be no conflict. When overriding a built-in command, e.g. `some_builtin_command()`, one can always access the original built-in command as `_some_builtin_command()`.

ADD_SUBDIRECTORIES()

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

Usage:

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

instead of:

```
ADD_SUBDIRECTORY(<dir1>)
ADD_SUBDIRECTORY(<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 just calls the built-in CMake commands:

```
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 just calls the built-in commands:

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

APPEND_CMNDLINE_ARGS()

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

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. This just makes the formatting of command-line arguments easier.

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 variable (reduces boiler-plate code).

Usage:

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

This just calls:

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

There is better error reporting if one misspells `APPEND_SET` than if one misspells `APPEND`.

APPEND_STRING_VAR()

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

Usage:

```
APPEND_STRING_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 one wants to ignore the meaning of these special characters and are okay with just adding one string at a time, then 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_STRING_VAR_EXT(<stringVar> "<string>")
```

Simply sets `<stringVar> = "${<stringVar>}<string>"` and leaves in `';'` without creating new array elements.

APPEND_STRING_VAR_WITH_SEP()

Append strings to a given string variable, joining them using a separator string.

Usage:

```
APPEND_STRING_VAR_WITH_SEP(<stringVar> "<sepStr>" "<str0>" "<str1>" ...)
```

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

ASSERT_DEFINED()

Assert that a variable 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 dereferencing of undefined variables. For example, how does one know if one did not misspell the name of a variable in an if statement like:

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

?

If one misspelled the variable SOME_VARBLE (which is likely in this case), then the if statement will always be false! To avoid this problem when one always expects that a variable is explicitly set, instead do:

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

Now if one misspells this variable, then CMake will asset and stop processing. This is not a perfect solution since one can misspell the variable name in the following if statement but typically one would always just copy and paste between the two statements so these names are always the same. This is the best that can be done in CMake unfortunately to catch usage of misspelled undefined variables.

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 variables <depOpName0>, <depOptName1>, ... are all ON. However, if <combinedOptionName> is set to ON by the user and not all of the dependent option variables are also ON, then this results in a fatal error and all processing stops.

This is used by a CMake project to automatically turn on a feature that requires a set of other features (when they are all enabled) but allows a user to disable the feature if desired.

CONCAT_STRINGS()

Concatenate a set of string arguments.

Usage:

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

On output, <outputVar> is set to "<str0><str1>...". This makes it easier to format a long string over multiple CMake source code lines.

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 arguments 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 one calls `ADD_SUBDIRECTORY(<someDir>)` or enters a `FUNCTION` that CMake actually creates a copy of all of the regular non-cache variables 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 one moves to a subordinate `CMakeLists.txt` file or enters a `FUNCTION`, then a local copy of the variable is *not* created. If one sets the variable locally, it will shadow the global variable. However, if one sets the global cache 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 code).

Usage:

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

This just calls:

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

This avoid problems with misspelling `CACHE`.

GLOBAL_SET()

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

Usage:

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

This just calls:

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

This avoid misspelling `CACHE`.

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 <stringi> is quoted using an escaped quote char \". If FALSE then no escaped quote is used.

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

Zero or more string arguments to be joined.

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

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

If <quoteElements>=TRUE, then <outputStrVar> 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 interprets those as array boundaries.

MESSAGE_WRAPPER()

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

Usage:

```
MESSAGE_WRAPPER(...)
```

This function takes exactly the same arguments as built-in MESSAGE() function. However, when the variable MESSAGE_WRAPPER_UNIT_TEST_MODE is set to TRUE, then this function will not call MESSAGE(...) but instead will prepend set to the global variable MESSAGE_WRAPPER_INPUT the input argument that would have gon to MESSAGE(). To capture just this call's input, first call:

```
GLOBAL_NULL_SET(MESSAGE_WRAPPER_INPUT)
```

before calling this function (or the functions/macros that call this function).

This function allows one to unit test other user-defined CMake macros and functions that call this function to catch error conditions without 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 function to make it easier to set longer strings over multiple lines.

This function is exactly the same as CONCAT_STRINGS() and should not even exist :-(

PARSE_ARGUMENTS()

Parse a set of macro/function 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 are:

`<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 variable name `<argNamei>`, a local variable will be created in the current scope with the name `<prefix>_<argNamei>` which gives the list of variables parsed out of `<inputArgsList>`.

`<optionNamesList>`

Quoted array of list options (e.g. `"<optName0>;<optName1>;..."`) typically pass in as `${ARGN}` in the outer function/macro. For each variable name `<optNamei>`, a local variable will be created in the current scope with the name `<prefix>_<optNamei>` that is either set to `TRUE` or `FALSE` depending if `<optNamei>` appears in `<inputArgsList>` or not.

`<inputArgsList>`

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

What this macro does is very simple yet very useful. What it does is to allow one to create one's own user-defined keyword-based macros and functions like is used by some built-in CMake commands.

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 variables in the current scope:

```
MyVar_ARG0="a;b"
MyVar_ARG1=""
MyVar_ARG2="c"
MyVar_OPT0="FALSE"
MyVar_OPT1="TRUE"
```

This allows one to define user-defined macros and functions that have a mixture of positional arguments and keyword-based arguments like one can do in other languages. The keyword-based arguments can be passed in any order and those that are missing are empty (or false for options) by default.

Any initial arguments that are not recognized as `<argNamesList>` or `<optionNamesList>` keyword arguments will be put into the local variable `<prefix>_DEFAULT_ARGS`. If no arguments in `<inputArgsList>` (typically `${ARGN}`) match any in `<argNamesList>`, then all non-option arguments are put into `<prefix>_DEFAULT_ARGS`. For example, if one passes in:

```
PARSE_SPECIAL_VARS(MyVar ARG5 a b c)
```

you will get:

```
MyVar_DEFAULT_ARGS="ARG5;a;b;c"
MyVar_ARG0=""
MyVar_ARG1=""
MyVar_ARG2=""
MyVar_OPT0="FALSE"
MyVar_OPT1="FALSE"
```

Multiple occurrences of keyword arguments in `<inputArgsList>` is allowed but only the last one listed will be recorded. For example, if one calls:

```
PARSE_SPECIAL_VARS(MyVar ARG1 a b ARG1 c)
```

then this will set:

```
MyVar_ARG0=""
MyVar_ARG1="c"
MyVar_ARG2=""
MyVar_OPT0="FALSE"
MyVar_OPT1="FALSE"
```

This is actually consistent with the way that most argument list parsers behave with respect to multiple instances of the same argument so hopefully this will not be a surprise to anyone.

If one puts an option keyword in the middle of a keyword argument list, the option keyword will get pulled out of the list. For example, if one calls:

```
PARSE_SPECIAL_VARS(MyVar ARG0 a OPT0 c)
```

then this will set:

```
MyVar_ARG0="a;c"
MyVar_ARG1=""
MyVar_ARG2=""
MyVar_OPT0="TRUE"
MyVar_OPT1="FALSE"
```

This is confusing behavior so users would be smart not to mix option arguments inside of list arguments.

If `PARSE_ARGUMENTS_DUMP_OUTPUT_ENABLED` is set to `TRUE`, then a bunch of detailed debug info will be printed. This should only be used in the most desperate 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 sets of argument and option names and input argument list names.

PREPEND_CMNDLINE_ARGS()

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

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> ...)
```

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

PRINT_NONEMPTY_VAR()

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

Usage:

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

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

PRINT_NONEMPTY_VAR_WITH_SPACES()

Print a defined variable giving its name then value printed with spaces instead of `';'` but only if it is not empty.

Usage:

```
PRINT_NONEMPTY_VAR_WITH_SPACES(<varName>  <printedVarInOut>)
```

Prints the variable as:

```
<varName>: <ele0> <ele1> ...
```

If `$<printedVarInOut>` is `TRUE` on input, then the varible is not touched. If however, the varible `$<printedVarInOut>` is not `TRUE` on input, then it is set to `TRUE` on output.

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 (removes boiler-plate code and errors).

Usage:

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

This function is necessary in order to preserve the "global" nature of the variable. If one just calls `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! The variable `<globalVarName>` must be defined before this function is called. If `<globalVarName>` is actually not a global cache variable before this function is called it will be after it completes.

SET_AND_INC_DIRS()

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

Usage:

```
SET_AND_INC_DIRS(<dirVarName> <includeDir>)
```

On output, this sets `<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 drop-down box in the CMake cache manipulation GUIs.

SET_DEFAULT()

Give a local variable a default value 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 `<arg0> <arg1> ...`. 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 environment variable 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 the local variable `<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 variables to be set through the command-line like `cmake` always allowed.

SPLIT()

Split a string variable 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 (regex) so keep that in mind when considering special regex chars like `'*'`, `'.'`, etc!

TIMER_GET_RAW_SECONDS()

Return the raw time in seconds (nano-second accuracy) 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 with `EXECUTE_PROCESS()` to run the `date` command. Therefore, it only works on Unix/Linux and other systems that have a standard `date` command. Since this uses `EXECUTE_PROCESS()`, this function should only be used to time very course-grained operations (i.e. that take longer than a second). If the *date* command does not exist, then ${<rawSecondsVar>} will be empty on output!

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>` (returned 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 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: 0m5.235s
```

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_HAS_SUBSTR_CONST()

Check that a given string var contains the given substring and update overall test statistics

Usage:

```
UNITTEST_HAS_SUBSTR_CONST(<varName> <substr>)
```

If `${<varName>}` contains the substring `<substr>`, 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 statistics 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 test 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 as an extra precaution to make sure that important unit tests are not left out. CMake is a very loosely typed language and it pays to be a little paranoid.

13 FAQ

Q: Why does not TriBITS just use the standard CMake `Find<PACKAGE_NAME>.cmake` modules and the standard `FIND_PACKAGE()` function to find TPLs?

A: The different "standard" CMake `Find<PACKAGE_NAME>.cmake` modules do not have a standard set of outputs and therefore, can't be handled in a uniform way. For example,

14 Appendix

14.1 History of TriBITS

TriBITS started development in November 2007 as a set of helper macros to provide a CMake build system for a small subset of packages in Trilinos. The initial goal was to 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 and augmented 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 implementation. Then, stating in early 2008, a more detailed evaluation was performed to see if Trilinos should switch 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 scalable package-based architecture (PackageArch) for the Trilinos CMake project in late 2008. This Trilinos CMake PackageArch system evolved over the next few years with development in the system slowing into 2010. This Trilinos CMake build system was then adopted as the build infrastructure for the CASL VERA effort in 2011 where CASL VERA packages were treated as add-on Trilinos packages (see Section Multi-Repository Support). Over the next year, there was significant development of the system to support larger multi-repo projects in support of CASL VERA. That lead to the decision to formally generalize the Trilinos CMake PackageArch build system outside of Trilinos and the name TriBITS was formally adopted in November 2011. Work to refactor the Trilinos CMake system into a general reusable stand-alone CMake-based build system started in October 2011 and an initial implementation was complete in December 2011 when it was used for the CASL VERA build system. In early 2012, the ORNL CASL-related projects Denovo and SCALE (see [SCALE, 2011]) adopted TriBITS as their native development build systems. Shortly after, TriBITS was adopted as the native build system for 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 project (see [LiveV]) which was forked and extended independently.

14.2 Why a TriBITS Package is not a CMake Package

Note that a TriBITS Package is not the same thing as a "Package" in raw CMake terminology. In raw CMake, a "Package" is some externally provided bit of software or other utility for which the current CMake project has an optional or required dependency (see CMake: How to Find Libraries). Therefore, a raw CMake "Package" actually maps to a TriBITS TPL. A raw CMake "Package" (e.g. Boost, CUDA, etc.) can be found using a standard CMake find module `Find<rawPackageName>.cmake` using the built-in CMake command `FIND_PACKAGE(<rawPackageName>)`. It is unfortunate that the TriBITS and the raw CMake definitions of the term "Package" are not exactly the same. However, the term "Package" was coined by the Trilinos project long ago before CMake was adopted as the Trilinos build system and Trilinos' definition of "Package" (going back to 1998) pre-dates the development of CMake (see History of CMake) and therefore Trilinos dictated the terminology of TriBITS and the definition of the term "Package" in the TriBITS system. However, note that both meanings of the term "Package" are consistent with the more general software engineering definition of a "Package" according to Software Engineering Packaging Principles.

14.3 Design Considerations for TriBITS

Some of the basic requirements and design goals for TriBITS are outlined in the TriBITS Overview document.

As stated in TriBITS Dependency Handling Behaviors, No circular dependencies of any kind are allowed. That is, no TriBITS SE package (or its tests) can declare a dependency on a downstream SE package, period! To some, this might seem over constraining but adding support for circular dependencies to the TriBITS system would add significant complexity and space/time overhead and is a bad idea from a basic software engineering perspective (see the *ADP (Acyclic Dependencies Principle)* in Software Engineering Packaging Principles). From a versioning, building, and change-prorogation perspective, any packages involved in a circular dependency would need to be treated as a single software engineering package anyway so TriBITS forces development teams to glob all of this stuff together into a single TriBITS SE Package when cycles in software exist. There are numerous wonderful ways to break circular dependencies between packages that are proven and well established in the SE community (for example, see [Agile Software Development, 2003]).

14.4 clone_extra_repos.py --help

Below is a snapshot of the output from `clone_extra_repos.py --help`. For more details on the usage of `clone_extra_repos.py`, see Multi-Repository Support and Multi-Repository Development Workflow.

```
Usage: clone_extra_repos.py [options]

This script clones one more extra repos from a TriBITS
ExtraRepositoriesList.cmake file.  The standard usage is:

  $ cd base <projectDir>
  $ ./cmake/tribits/ci_support/clone-extra-repos.py

where <projectDir> is the base TriBITS project dir and base git repo.


By default, that will clone all the 'Nightly' extra repos that are listed in the
file:

  <projectDir>/cmake/ExtraRepositoriesList.cmake

(other repo types can be selected usng --extra-repos-type).

The list of which repos to clone can "white-list" selected with the option
--extra-repos (see options below for details).  Extra repos can in addition be
"back-listed" using the option --not-extra-repos.

To see the full list of repos that can be cloned, pass in just:

  --skip-clone --verbosity=more
```

```
That will print out a table like:

  ------------------------------------------------------------------------------
  | ID | Repo Name  | Repo Dir   | VC  | Repo URL                 | Category   |
  |----|------------|------------|-----|--------------------------|------------|
  |  1 | ExtraRepo1 | ExtraRepo1 | GIT | someurl.com.ExtraRepo1   | Continuous |
  |  2 | ExtraRepo3 | ExtraRepo3 | GIT | someurl3.com:/ExtraRepo3 | Continuous |
  ------------------------------------------------------------------------------

If the git repo server is using gitolite, one can set
--gitolite-root=<gitolite-root> and that will result in git repos being
selected only if the selcted repos are listed in 'ssh <gitolite-root> info'.
This allows one to automatically exclude repos from being cloned that the user
has no permissions to clone.  NOTE: See warning about --gitolite-root below!

TIP: After cloning the set of repos, a nice too to use on the repos is
'gitdist'.  If your project does not have a version controlled
.gitdist.default file, you can generate one using the
--create-gitdist-file=<gitdist-file> argument, for example with:

  --create-gitdist-file=.gitdist


Options:
  -h, --help            show this help message and exit
  --extra-repos=EXTRAREPOS
                        List of names of extra repos to be cloned <extra-
                        repos> (i.e. "repo0,repo1,,...").  When set to empty
                        '' (the default value) then all repos that match
                        <extra-repos-type> listed in <extra-repos-file> will
                        be selected.  But the repos listed in <extra-repos>
                        must always be a subset of the repos of type <extra-
                        repos-type> selected from <extra-repos-file>.
                        (Default '')
  --not-extra-repos=NOTEXTRAREPOS
                        List of names of extra repos *NOT* to clone (i.e.
                        "repo0,repo1,...").  (Default '')
  --extra-repos-file=EXTRAREPOSFILE
                        The file path <extra-repos-file> for the
                        ExtraRepositoriesList.cmake file.  This can be an
                        absolute or relative path.  (Default =
                        'cmake/ExtraRepositoriesList.cmake')
  --extra-repos-type=EXTRAREPOSTYPE
                        Type of extra repositories <extra-repos-type> to
                        select from <extra-repos-file>.  When --extra-repos is
                        set, then this arugment is ignored. Choices =
                        ('Continuous', 'Nightly', 'Experimental').  [default =
                        'Nightly']
  --gitolite-root=GITOLITEROOT
                        Gives the root for a gitolite repos <gitolite-root>
                        (e.g. git@<some-url>).  If specified, then any git
                        repos with the <gitolite-root> listed as their root
                        will only be selected if they are listed with 'R'
                        permissions returned from 'ssh <gitolite-root> info'.
                        WARNING: Make sure that you have your gitoliote SSH
                        registred correctly before using this option by typing
                        the command 'ssh <gitlite-root> info' and make sure
                        that it does *not* ask for a password! (Default = '')
  --with-cmake=WITHCMAKE
```

```
                        CMake executable to use with cmake -P scripts
                        internally (only set by unit testing code).  (Default
                        = 'cmake')
  --verbosity=VERBLEVEL
                        Verbosity of the script (levels are cumulative):  none
                        = no output at all (except for commands with --no-op).
                        minimal = print script args echo and clone commands.
                        more = print basic repo include/exclude logic and
                        print repo table.  most = print output from cmake
                        script called, the output from gitolite, and other
                        detailed info. Choices = ('none', 'minimal', 'more',
                        'most').  [default = 'more']
  --do-clone            Do the clone of the selected repos. [default]
  --skip-clone          Skip the clone of the repos and just show what would
                        be done.
  --do-op               Do the clone of the selected repos. [default]
  --no-op               Skip cloning the repos and just show the clone
                        commands.
  --create-gitdist-file=CREATEGITDISTFILE
                        If specified, the file <gitdist-file> will get
                        generated with the list of git repos (the same list
                        that is cloned with --do-clone).  (Default = '')
  --show-defaults       Show the default option values and do nothing at all.
```

14.5 gitdist --help

Below is a snapshot of the output from `gitdist --help`. For more details on the usage of `gitdist`, see Multi-Repository Support and Multi-Repository Development Workflow.

```
Usage: gitdist [gitdist arguments] [git arguments]
       gitdist [gitdist arguments] dist-repo-status

Run git recursively over extra repos multi-repository git projects.  Also
includes other tools like printing a repo status table and tracking versions
through RepoVersion.txt files.

Instead of typing:

  $ git [git arguments]

type:

  $ gitdist [gitdist options] [git arguments]

This will distribute git commands across all the listed git repos, including
the base git repo.  The options in [gitdist options] are prefixed with
'--dist-' and are pulled out before passing the remaining arguments in [git
arguments] to all the git repos.  See --help to see the [gitdist options].

For example, consider the following git repos with other git repos cloned
under it:

  BaseRepo/
    .git/
    ExtraRepo1/
      .git/
    ExtraRep2/
      .git/
```

The gitdist command is run in the base git repo 'BaseRepo' and the git commands are run on it as well as the git repos 'ExtraRepo1' and 'ExtraRepo2'.

The set of repos processed is determined by the argument --dist-extra-repos or the files .gitdist or .gitdist.default. If --dist-extra-repos="", then the list of extra repos will be read from the file '.gitdist' in the current working directory. If the file '.gitdist' does not exist, then the list of extra repos will be read from the file '.gitdist.default' in the current working directory. The format of this files '.gitdist' and '.gitdist.default' is to have one repo directory/name per line as in:

```
ExtraRepo1
ExtraRepo2
...
```

where each repo is the relative path to the repo under the base git repo (e.g. under 'BaseRepo/'). The file .gitdist.default is meant to be committed to the base git repo so that gitdist is ready to use right away after the extra repos are cloned (see the tool clone_extra_repos.py).

If an extra repository directory listed in the .gitdist file (or specified using --dist-extra-repos) does not exist, then it will be ignored by the script. Therefore, be careful to manually verify that the script recognizes the repositories that you list. The best way to do that is to run 'gitdist dist-repo-status' and see which repos are listed.

This script is self-contained and has no dependencies other than standard python 2.4 packages so it can be copied to anywhere and used.

TIPS:

- To see the status of all repos in a compact way, use the special 'dist-repo-status' command (see below).

- To process only repos that are changed w.r.t. their tracking branch, run 'gitdist --dist-mod-only [git arguments]'. For example, to see the status of only changed repos use 'gitdist --dist-mod-only status'.

- 'gitdist --help' will run gitdist help, not git help. If you want raw git help, run 'git --help'.

- By default, gitdist will use 'git' in the environment. If it can't find 'git' in the environment, it will require that the user specify the git command to run with --dist-use-git=<the command>.

- Pass in --dist-not-base-repo and/or --dist-not-extra-repos=RepoX,RepoZ,... to exclude the processing of either the base git repo and/or other git repos listed in .gitdist, respectively.

SUMMARY OF REPO STATUS:

This script supports the special command 'dist-repo-status' which prints a table showing the current status of all the repos. For the repos shown above:

```
$ gitdist dist-repo-status
```

prints a table like:

```
-------------------------------------------------------------------
| ID | Repo Dir        | Branch | Tracking Branch | C | M  | ? |
```

```
  |----|-----------------|--------|-----------------|---|----|---|
  |  0 | BaseRepo (Base) | dummy  |                 |   |    |   |
  |  1 | ExtraRepo1      | master | origin/master   | 1 |  2 |   |
  |  2 | ExtraRepo2      | HEAD   |                 |   | 25 | 4 |
  ----------------------------------------------------------------
```

If the option --dist-legend is passed in, it will print the legend:

```
Legend:
 ID: Repository ID, zero based (order git commands are run)
 Repo Dir: Relative to base repo (base repo shown first with '(Base)')
 Branch: Current branch (or detached HEAD)
 Tracking Branch: Tracking branch (or empty if no tracking branch exists)
 C: Number local commits w.r.t. tracking branch (empty if zero or no TB)
 M: Number of tracked modified (uncommitted) files (empty if zero)
 ?: Number of untracked, non-ignored files (empty if zero)
```

REPO VERSION FILES:

This script supports the options --dist-version-file=<versionFile> and --dist-version-file2=<versionFile2> which are used to provide different SHA1 versions for each repo. Each of these version files is expected to represent a compatible set of versions of the repos.

The format of these repo version files is shown in the following example:

```
-------------------------------------------------------
** Base Git Repo: BaseRepo
e102e27 [Mon Sep 23 11:34:59 2013 -0400] <author1@someurl.com>
First summary message
** Git Repo: ExtraRepo1
b894b9c [Fri Aug 30 09:55:07 2013 -0400] <author2@someurl.com>
Second summary message
** Git Repo: ExtraRepo2
97cf1ac [Thu Dec 1 23:34:06 2011 -0500] <author3@someurl.com>
Third summary message
-------------------------------------------------------
```

(the lines '---------' are *not* included in the file.)

Each repository entry can have a summary message or not (i.e. use two or three lines per repo in the file). A compatible repo version file can be generated with this script using, for example, using:

```
  $ gitdist --dist-no-color log -1 --pretty=format:"%h [%ad] <%ae>%n%s" \
    | grep -v "^$" &> RepoVersion.txt
```

using three lines per repo, or just:

```
  $ gitdist --dist-no-color log -1 --pretty=format:"%h [%ad] <%ae>" \
    | grep -v "^$" &> RepoVersion.txt
```

using two lines per repo in the output file.

This allows checking out consistent versions of the repos, diffing two consistent versions of the repos, etc.

To checkout an older set of consistent versions of the set of repos represented by the set of versions given in the file RepoVersion.<date>.txt, use:

```
  $ gitdist fetch origin
  $ gitdist --dist-version-file=RepoVersion.<date>.txt checkout _VERSION_

The '_VERSION_' string will be replaced with the SHA1 for each of the repos.

To tag and branch the set of repos using a consistent set of versions, use:

  $ gitdist --dist-version-file=RepoVersion.<date>.txt \
      tag -a <some_tag> _VERSION_

To diff two sets of versions of the repos, for example, use:

  $ gitdist --dist-version-file=RepoVersion.<new-date>.txt \
      --dist-version-file2=RepoVersion.<old-date>.txt \
      diff _VERSION_ ^_VERSION2_

Here, _VERSION_ is replaced by the SHA1s listed in RepoVersion.<new-date>.txt
and _VERSION2_ is replaced by the SHA1s listed in RepoVersion.<old-date>.txt.

One can construct any git commit taking one or two different repo version
arguments (SHA1s) using this approach.

Note that the set of repos listed in the RepoVersion.txt file must be a
super-set of those processed by this script or an error will occur and the
script will stop.

USEFUL ALIASES:

Two very useful Linux/Unix aliases are:

  $ alias gitdist-status="gitdist dist-repo-status"
  $ alias gitdist-mod="gitdist --dist-mod-only"

This avoids lot of typing as these arguments are used a lot when working with
many git repos.  For example, to see the status of all your repos, do:

  $ gitdist-status | less

To see the changes in the repos use:

  $ gitdist-mod log --name-status HEAD ^origin/master

or

  $ gitdist-mod local-stat

(where 'local-stat' is a useful git alias defined in the script
'git-config-alias.sh').

SCRIPT DEPENDENCIES:

This Python script only depends on the Python 2.4+ standard modules 'sys',
'os', 'subprocess', and 're'. Also, of course, it requires some compatible
version of 'git' in your path.



Options:
  -h, --help            show this help message and exit
```

```
  --dist-use-git=USEGIT
                        The (path) to the git executable to use for each git
                        repo command (default='git')
  --dist-extra-repos=EXTRAREPOS
                        Comma-separated list of extra repos to forward git
                        commands to.  If the list is empty, it will look for a
                        file called .gitdist to get the list of extra repos
                        separated by newlines.
  --dist-not-extra-repos=NOTEXTRAREPOS
                        Comma separated list of extra repos to *not* forward
                        git commands to.  This removes any repos from being
                        processed that would otherwise be.
  --dist-not-base-repo  If set, don't pass the git command on to the base git
                        repo.
  --dist-version-file=VERSIONFILE
                        Path to a file contains a list of extra repo
                        directories and git versions (replaces _VERSION_).
  --dist-version-file2=VERSIONFILE2
                        Path to a second file contains a list of extra repo
                        directories and git versions (replaces _VERSION2_).
  --dist-no-color       If set, don't use color in the output for gitdist
                        (better for output to a file).
  --dist-debug          If set, then debugging info is printed.
  --dist-no-opt         If set, then no git commands will be run but instead
                        will just be printed.
  --dist-mod-only       If set, then the listed git command will be only be
                        run and output for the repo will only be produced if
                        the command 'git diff --name-only ^<tracking-branch>'
                        returns non-empty output where <tracking-branch> is
                        returned from 'rev-parse --abbrev-ref --symbolic-full-
                        name @{u}'.  In order words, if a git repo is
                        unchanged w.r.t. its tracking branch, then the git
                        command is skipped for that repo.  If a repo does not
                        have a tracking branch, then the repo will be skipped
                        as well.  Therefore, be careful to first run with
                        dist-local-stat to see the status of each local repo
                        to know which repos don't have tracking branches.
  --dist-legend         If set, then a legend will be printed below the repo
                        summary table for the special dist-repo-status
                        command.  Only applicable with dist-repo-status.
```

14.6 snapshot-dir.py --help

Below is a snapshot of the output from `snapshot-dir.py --help`. For more details on the usage of `snapshot-dir.py`, specifically for snapshotting the <projectDir>/cmake/tribits/ directory, see TriBITS directory snapshotting.

```
Usage: snapshot-dir.py [OPTIONS]

This tool snapshots the contents of an origin directory ('orig-dir') to
destination directory ('dest-dir') and creates linkages between the two git
repos in the commit message in the 'dest-dir' git branch.  The command 'git'
must be in the path for this script to be used.

To demonstrate how this script is used, consider the desire to snapshot the
directory tree:

  <some-orig-base-dir>/orig-dir/
```

and duplicate it in the directory tree

```
  <some-dest-base-dir>/dest-dir/
```

Here, the directories can be any two directories from local git repos with any names as long as they are given a final '/' at the end. Otherwise, if you are missing the final '/', then rsync will copy the contents from 'orig-dir' into a subdir of 'dest-dir' which is usually not what you want.

A typical case is to have snapshot-dir.py soft linked into orig-dir/ to allow a simple sync process. This is the case, for example, with the 'tribits' source tree. The linked-in location of snapshot-dir.py gives the default 'orig-dir' directory automatically (but can be overridden with --orig-dir option).

When snapshot-dir.py is soft-linked into the 'orig-dir' directory base, the way to run this script would be:

```
  $ cd <some-dest-base-dir>/dest-dir/
  $ <some-orig-base-dir>/orig-dir/snapshot-dir.py
```

By default, this assumes that git repos are used for both the 'orig-dir' and 'dest-dir' locations. The info about the origin of the snapshot from 'orig-dir' is recorded in the commit message of the 'dest-dir' git repo to provide tractability for the versions (see below).

To sync between any two arbitrary directories invoking this script from any directory location, one can do:

```
  $ <some-base-dir>/snapshot-dir.py \
    --orig-dir=<some-orig-dir>/ \
    --dest-dir=<some-dest-dir>/
```

Note the trailing '/' is critical for the correct functioning of rsync.

By default, this script does the following:

1) Asserts that the git repo for 'orig-dir' is clean (i.e. no uncommitted files, no unknown files, etc.). This can be disabled by passing in --allow-dirty-orig-dir.

2) Asserts that the git repo for <some-dest-dir>/ is clean (see above). This can be disabled by passing in --allow-dirty-dest-dir.

3) Run 'rsync -cav --delete' to copy the contents from 'orig-dir' to 'dest-dir', excluding the '.git/' directory if it exists in either git repo dir. After this runs, <some-dest-dir>/ should be an exact duplicate of <some-orig-dir>/ (except for otherwise noted excluded files). This rsync will delete any files in 'dest-dir' that are not in 'orig-dir'. Note that if there are ignored untracked files, then the copied .gitignore files should avoid showing them as tracked or unknown files in the 'dest-dir' git repo as well.

4) Run 'git add .' in <some-dest-dir>/ to stage any new files. Note that git will automatically stage deletes for any files removed by the 'rsync -cav --delete' command!

5) Get the git remote URL from the orig-dir git repo, and the git log for the last commit for the directory from orig-dir. This information is used to define perfect tracing of the version info when doing the snapshot.

```
6) Commit the updated dest-dir directory using a commit message with the
   orig-dir repo URL and log info.  This will only commit files in 'dest-dir'
   and not in other directories in the destination git repo!

NOTES:

 This script allows the syncing between base git repos or subdirs within git
  repos.  This is allowed because the rsync command is told to ignore the
  .git/ directory when syncing.

 Snapshotting with this script will create an exact duplicate of 'orig-dir'
  in 'dest-dir' and therefore if there are any local changes to the files or
  chagnes after the last snapshot, they will get wiped out.  To avoid this, do
  the snapshot on a branch in the 'dest-dir' git repo, then merge that branch
  into the master branch in 'dest-dir' repo that has the local changes.  As
  long as there are no merge conflicts, this will preserve local changes for
  the mirrored directories and files.  This works amazingly well in most
  cases.


Options:
  -h, --help            show this help message and exit
  --show-defaults       Show the default option values and do nothing at all.
  --orig-dir=ORIGDIR    Original directory that is the source for the
                        snapshotted directory.  Note that it is important to
                        add a final /' to the directory name.  The default is
                        the directory where this script lives (or is soft-
                        linked).  [default: '<orig-dir>']
  --dest-dir=DESTDIR    Destination directory that is the target for the
                        snapshoted directory.  Note that a final '/' just be
                        added or the origin will be added as subdir.  The
                        default dest-dir is current working directory.
                        [default: '<dest-dir>']
  --assert-clean-orig-dir
                        Check that orig-dir is committed and clean. [default]
  --allow-dirty-orig-dir
                        Skip clean check of orig-dir.
  --assert-clean-dest-dir
                        Check that dest-dir is committed and clean. [default]
  --allow-dirty-dest-dir
                        Skip clean check of dest-dir.
  --do-rsync            Actually do the rsync. [default]
  --skip-rsync          Skip the rsync (testing only?).
  --do-commit           Actually do the commit. [default]
  --skip-commit         Skip the commit.
```

14.7 checkin-test.py --help

Below is a snapshot of the output from `checkin-test.py --help`. This `--help` output contains a lot of information about the recommended development workflow (mostly related to pushing commits) and outlines a number of different use cases for using the script.

```
Usage: checkin-test.py [OPTIONS]

This tool does testing of a TriBITS-based project using CTest and this script
can actually do the push itself using eg/git in a safe way.  In fact, it is
recommended that one uses this script to push since it will amend the last
commit message with a (minimal) summary of the builds and tests run with
```

results and/or send out a summary email about the builds/tests performed.

Quickstart:

In order to do a safe push, perform the following recommended workflow
(different variations on this workflow are described below):

1) Commit changes in the local repo:

```
  # 1.a) See what files are changed, newly added, etc. that need to be committed
  # or stashed.
  $ eg status

  # 1.b) Stage the files you want to commit (optional)
  $ eg stage <files you want to commit>

  # 1.c) Create your local commits
  $ eg commit -- SOMETHING
  $ eg commit -- SOMETHING_ELSE
  ...

  # 1.d) Stash whatever changes are left you don't want to test/push (optional)
  $ eg stash
```

  NOTE: You can group your commits any way that you would like (see the basic
  eg/git documentation).

  NOTE: If not installed on your system, the eg script can be found at
  tribits/common_tools/git/eg. Just add it to your path.

  NOTE: When multiple repos are involved, use egdist instead. It is provided
  at tribits/common_tools/git/egdist. See egdist --help for details.

2) Review the changes that you have made to make sure it is safe to push:

```
  $ cd $PROJECT_HOME
  $ eg local-stat                  # Look at the full status of local repo
  $ eg diff --name-status origin   # [Optional] Look at the files that have changed
```

  NOTE: The command 'local-stat' is a git alias that can be installed with the
  script tribits/common_tools/git/git-config-alias.sh. It is highly
  recommended over just a raw 'eg status' or 'eg log' to review commits before
  attempting to test/push commits.

  NOTE: If you see any files/directories that are listed as 'unknown' returned
  from 'eg local-stat', then you will need to do an 'eg add' to track them or
  add them to an ignore list *before* you run the checkin-test.py script.
  The eg script will not allow you to push if there are new 'unknown' files or
  uncommitted changes to tracked files.

3) Set up the checkin base build directory (first time only):

```
  $ cd $PROJECT_HOME
  $ echo CHECKIN >> .git/info/exclude
  $ mkdir CHECKIN
  $ cd CHECKIN
```

  NOTE: You may need to set up some configuration files if CMake can not find

```
  the right compilers, MPI, and TPLs by default (see detailed documentation
  below).

  NOTE: You might want to set up a simple shell driver script.

  NOTE: You can set up a CHECKIN directory of any name in any location
  you want.  If you create one outside of the main source dir, then
  you will not have to add the git exclude shown above.

4) Do the pull, configure, build, test, and push:

  $ cd $PROJECT_HOME
  $ cd CHECKIN
  $ ../checkin-test.py -j4 --do-all --push

  NOTE: The above will: a) pull updates from the global repo(s), b)
  automatically enable the correct packages, c) configure and build the right
  packages, d) run the tests, e) send you emails about what happened, f) do a
  final pull from the global repo, g) optionally amend the last local commit
  with the test results, and h) finally push local commits to the global
  repo(s) if everything passes.

  NOTE: You must have installed the official versions of eg/git with the
  install-git.py script in order to run this script.  If you don't, the script
  will die right away with an error message telling you what the problem is.

  NOTE: The current branch will be used to pull and push to.  A raw 'eg pull'
  is performed which will get all of the branches from 'origin'.  This means
  that your current branch must be a tracking branch so that it will get
  updated correctly.  The branch 'master' is the most common branch but
  release tracking branches are also common.

  NOTE: You must not have any uncommitted changes or the 'eg pull && eg rebase
  --against origin' command will fail on the final pull/rebase before the push
  and therefore the whole script will fail.  To run the script, you will may
  need to first use 'eg stash' to stash away your unstaged/uncommitted changes
  *before* running this script.

  NOTE: You need to have SSH public/private keys set up to the remote repo
  machines for the git commands invoked in the script to work without you
  having to type a password.

  NOTE: You can do the final push in a second invocation of the script with a
  follow-up run with --push and removing --do-all (it will remember the
  results from the build/test cases just ran).  For more details, see detailed
  documentation below.

  NOTE: Once you start running the checkin-test.py script, you can go off and
  do something else and just check your email to see if all the builds and
  tests passed and if the push happened or not.

  NOTE: The commands 'cmake', 'ctest', and 'make' must be in your default path
  before running this script.

For more details on using this script, see the detailed documentation below.


Detailed Documentation:
-----------------------
```

```
The following approximate steps are performed by this script:

--------------------------------------------------------------------------------

1) Check to see if the local repo(s) are clean:

  $ eg status

  NOTE: If any modified or any unknown files are shown, the process will be
  aborted.  The local repo(s) working directory must be clean and ready to
  push *everything* that is not stashed away.

2) Do a 'eg pull' to update the code (done if --pull or --do-all is set):

  NOTE: If not doing a pull, use --allow-no-pull or --local-do-all.

3) Select the list of packages to enable forward/downstream based on the
package directories where there are changed files (or from a list of packages
passed in by the user).

  NOTE: The automatic enable behavior can be overridden or modified using the
  options --enable-all-packages, --enable-packages, --disable-packages, and/or
  --no-enable-fwd-packages.

4) For each build/test case <BUILD_NAME> (e.g. MPI_DEBUG, SERIAL_RELEASE,
extra builds specified with --extra-builds):

  4.a) Configure a build directory <BUILD_NAME> in a standard way for all of
  the packages that have changed and all of the packages that depend on these
  packages forward/downstream. You can manually select which packages get
  enabled (see the enable options above).  (done if --configure, --do-all, or
  --local-do-all is set.)

  4.b) Build all configured code with 'make' (e.g. with -jN set through
  -j or --make-options).  (done if --build, --do-all, or --local-do-all is set.)

  4.c) Run all BASIC tests for enabled packages.  (done if --test, --do-all,
  or --local-do-all is set.)

  4.d) Analyze the results of the update, configure, build, and tests and send
  email about results.  (emails only sent out if --send-emails-to != "")

5) Do final pull and rebase, append test results to last commit message, and
push (done if --push is set)

  5.a) Do a final 'eg pull' (done if --pull or --do-all is set)

  5.b) Do 'eg rebase --against origin/<current_branch>' (done if --pull or
  --do-all is set and --rebase is set)

    NOTE: The final 'eg rebase --against origin/<current_branch>' is
    required to avoid trivial merge commits that the global get repo
    will reject on the push.

  5.c) Amend commit message of the most recent commit with the summary of the
  testing performed.  (done if --append-test-results is set.)

  5.d) Push the local commits to the global repo (done if --push is set)
```

```
-------------------------------------------------------------------------------


The recommended way to use this script is to create a new base CHECKIN test
directory apart from your standard build directories such as with:

  $ $PROJECT_HOME
  $ mkdir CHECKIN
  $ echo CHECKIN >> .git/info/exclude

The most basic way to do pre-push testing is with:

  $ cd CHECKIN
  $ ../checkin-test.py --do-all [other options]

If your MPI installation, other compilers, and standard TPLs (i.e. BLAS and
LAPACK) can be found automatically, then this is all you will need to do.
However, if the setup cannot be determined automatically, then you can add a
set of CMake variables that will get read in the files:

  COMMON.config
  MPI_DEBUG.config
  SERIAL_RELEASE.config

(or whatever your standard --default-builds are).

Actually, for built-in build/test cases, skeletons of these files will
automatically be written out with typical CMake cache variables (commented
out) that you would need to set out.  Any CMake cache variables listed in
these files will be read into and passed on the configure line to 'cmake'.

WARNING: Please do not add any extra CMake cache variables than what are
needed to get the Primary Tested (PT) --default-builds builds to work.  Adding
other enables/disables will make the builds non-standard and can break these
PT builds.  The goal of these configuration files is to allow you to specify
the minimum environment to find MPI, your compilers, and the required TPLs
(e.g. BLAS, LAPACK, etc.).  If you need to fudge what packages are enabled,
please use the script arguments --enable-packages, --disable-packages,
--no-enable-fwd-packages, and/or --enable-all-packages to control this, not
the *.config files!

WARNING: Please do not add any CMake cache variables in the *.config files
that will alter what packages or TPLs are enabled or what tests are run.
Actually, the script will not allow you to change TPL enables in these
standard *.config files because to do so deviates from a consistent build
configuration for Primary Tested (PT) Code.

NOTE: All tentatively-enabled TPLs (e.g. Pthreads and BinUtils) are hard
disabled in order to avoid different behaviors between machines where they
would be enabled and machines where they would be disabled.

NOTE: If you want to add extra build/test cases that do not conform to
the standard build/test configurations described above, then you need
to create extra builds with the --extra-builds and/or
--st-extra-builds options (see below).

NOTE: Before running this script, you should first do an 'eg status' and 'eg
diff --name-status origin..' and examine what files are changed to make sure
```

```
you want to push what you have in your local working directory.  Also, please
look out for unknown files that you may need to add to the git repository with
'eg add' or add to your ignores list.  There cannot be any uncommitted changes
in the local repo before running this script.

NOTE: You don't need to run this script if you have not changed any files that
affect the build or the tests.  For example, if all you have changed are
documentation files, then you don't need to run this script before pushing
manually.

NOTE: To see detailed debug-level information, set
TRIBITS_CHECKIN_TEST_DEBUG_DUMP=ON in the env before running this script.


Common Use Cases (examples):
----------------------------

(*) Basic full testing with integrating with global repo(s) without push:

  ../checkin-test.py --do-all

  NOTE: This will result in a set of emails getting sent to your email address
  for the different configurations and an overall push readiness status email.

  NOTE: If everything passed, you can follow this up with a --push (see
  below).

(*) Basic full testing with integrating with local repo and push:

  ../checkin-test.py --do-all --push

  NOTE: By default this will rebase your local commits and ammend the last
  commit with a short summary of test results.  This is appropriate for
  pushing commits that only exist in your local repo and are not shared with
  any remote repo.

(*) Push to global repo after a completed set of tests have finished:

  ../checkin-test.py [other options] --push

  NOTE: This will pick up the results for the last completed test runs with
  [other options] and append the results of those tests to the log of the most
  recent commit.

  NOTE: Take the action options for the prior run and replace --do-all with
  --push but keep all of the rest of the options the same.  For example, if
  you did:

    ../checkin-test.py --enable-packages=Blah --default-builds=MPI_DEBUG --do-all

  then follow that up with:

    ../checkin-test.py --enable-packages=Blah --default-builds=MPI_DEBUG --push

  NOTE: This is a common use case when some tests are failing which aborted
  the initial push but you determine it is okay to push anyway and do so with
  --force-push.

(*) Test only the packages modified and not the forward dependent packages:
```

```
../checkin-test.py --do-all --no-enable-fwd-packages
```

NOTE: This is a safe thing to do when only tests in the modified packages are changed and not library code. This can speed up the testing process and is to be preferred over not running this script at all. It would be very hard to make this script automatically determine if only test code has changed because every package does not follow a set pattern for tests and test code.

(*) Run the most important default (e.g. MPI_DEBUG) build/test only:

```
../checkin-test.py --do-all --default-builds=MPI_DEBUG
```

(*) The minimum acceptable testing when code has been changed:

```
../checkin-test.py \
  --do-all --enable-all-packages=off --no-enable-fwd-packages \
  --default-builds=MPI_DEBUG
```

NOTE: This will do only an MPI DEBUG build and will only build and run the tests for the packages that have directly been changed and not any forward packages. Replace "MPI_DEBUG" with whatever your most important default build is.

(*) Test only a specific set of packages and no others:

```
../checkin-test.py \
  --enable-packages=<P0>,<P1>,<P2> --no-enable-fwd-packages \
  --do-all
```

NOTE: This will override all logic in the script about which packages will be enabled based on file changes and only the given packages will be enabled. When there are tens of thousands of changed files and hundreds of defined packages, this auto-detection algorithm can be very expensive!

NOTE: You might also want to pass in --enable-all-packages=off in case the script wants to enable all the packages (see the output in the checkin-test.py log file for details) and you think it is not necessary to do so.

NOTE: Using these options is greatly preferred to not running this script at all and should not be any more expensive than the testing you would already do manually before a push.

(*) Test changes locally without pulling updates:

```
../checkin-test.py --local-do-all
```

NOTE: This will just configure, build, test, and send an email notification without updating or changing the status of the local git repo in any way and without any communication with the global repo. Hence, you can have uncommitted changes and still run configure, build, test without having to commit or having to stash changes.

NOTE: This is not a sufficient level of testing in order to push the changes to the global repo because you have not fully integrated your changes yet with other developers. However, this would be a sufficient level of testing in order to do a commit on the local machine and then pull to a remote machine for further testing and a push (see below).

(*) Adding extra build/test cases:

Often you will be working on Secondary Tested (ST) Code or Experimental (EX) Code and want to include the testing of this in your pre-push testing process along with the standard --default-builds build/test cases which can only include Primary Tested (PT) Code. In this case you can run with:

```
../checkin-test.py --extra-builds=<BUILD1>,<BUILD2>,... [other options]
```

For example, if you have a build that enables the TPL CUDA you would do:

```
echo "
-DTPL_ENABLE_MPI:BOOL=ON
-DTPL_ENABLE_CUDA:BOOL=ON
" > MPI_DEBUG_CUDA.config
```

and then run with:

```
../checkin-test.py --extra-builds=MPI_DEBUG_CUDA --do-all
```

This will do the standard --default-builds (e.g. MPI_DEBUG and SERIAL_RELEASE) build/test cases along with your non-standard MPI_DEBUG_CUDA build/test case.

NOTE: You can disable the default build/test cases with --default-builds="". However, please only do this when you are not going to push because you need at least one default build/test case (the most important default PT case, e.g. MPI_DEBUG) to do a safe push.

(*) Including extra repos and extra packages:

You can also use the checkin-test.py script to continuously integrate multiple git repos containing add-on packages. To do so, just run:

```
../checkin-test.py --extra-repos=<REPO1>,<REPO2>,... [options]
```

NOTE: You have to create local commits in all of the extra repos where there are changes or the script will abort.

NOTE: Extra repos can be specified with more flexibility using the --extra-repos-file and --extra-repos-type arguments (also see --ignore-missing-extra-repos).

NOTE: Each of the last local commits in each of the changed repos will get amended with the appended summary of what was enabled in the build/test (if --append-test-results is set).

(*) Avoid changing any of the local commit SHA1s:

If you are pushing commits from a shared branch, it is critical that you do not change any of the SHA1s of the commits. Changing the SHA1s for any of the commits will mess up various multi-repo, multi-branch workflows. To avoid changing any of the SHA1s of the local commits, one must run with:

```
../checkin-test.py --no-rebase --no-append-test-results [options]
```

(*) Performing a remote test/push:

If you develop on a slow machine like your laptop, doing an appropriate level of testing can take a long time. In this case, you can pull the

changes to another faster remote workstation and do a more complete set of tests and push from there. If you are knowledgeable with git, this will be easy and natural to do, without any help from this script. However, this script can still help and automate the steps and can do so in one command invocation on the part of the developer.

On your slow local development machine 'mymachine', do the limited testing with:

```
../checkin-test.py --do-all --no-enable-fwd-packages
```

On your fast remote test machine, do a full test and push with:

```
../checkin-test.py \
  --extra-pull-from=<remote-name>:master \
  --do-all --push
```

where <remote-name> is a git repo pointing to mymachine:/some/dir/to/your/src:master (see 'git help remote').

NOTE: You can of course adjust the packages and/or build/test cases that get enabled on the different machines.

NOTE: Once you invoke the checkin-test.py script on the remote test machine and it has pulled the commits from mymachine, then you can start changing files again on your local development machine and just check your email to see what happens on the remote test machine.

NOTE: If something goes wrong on the remote test machine, you can either work on fixing the problem there or you can fix the problem on your local development machine and then do the process over again.

NOTE: If you alter the commits on the remote machine (such as squashing commits), you will have trouble merging back on our local machine. Therefore, if you have to to fix problems, make new commits and don't alter the ones you pulled from your local machine (but rebasing them should be okay as long as the local commits on mymachine are not pushed to other repos).

NOTE: Git will resolve the duplicated commits when you pull the commits pushed from the remote machine. Git knows that the commits are the same and will do the right thing when rebasing (or just merging).

(*) Check push readiness status:

```
../checkin-test.py
```

NOTE: This will examine results for the last testing process and send out an email stating if the a push is ready to perform or not.

(*) See the default option values without doing anything:

```
../checkin-test.py --show-defaults
```

NOTE: This is the easiest way to figure out what all of the default options are.

Hopefully the above documentation, the example use cases, the documentation of the command-line arguments below, and some experimentation will be enough to get you going using this script for all of your pre-push testing and pushes.

If that is not sufficient, send email to your development support team to ask for help.

Handling of PT, ST, and EX Code in built-in and extra builds:

This script will only process PT (Primary Tested) packages in the --default-builds (e.g. MPI_DEBUG and SERIAL_RELEASE) builds. This is to avoid problems of side-effects of turning on ST packages that would impact PT packages (e.g. an ST package getting enabled that enables an ST TPL which turns on support for that TPL in a PT package producing different code which might work but the pure PT build without the extra TPL may actually be broken and not know it). Therefore, any non-PT packages that are enabled (either implicitly through changed files or explicitly by listing in --enable-packages) will be turned off in the --default-builds builds. If none of the enabled packages are PT, then they will all be disabled and the --default-builds builds will be skipped.

In order to better support the development of ST and EX packages, this script allows you to define some extra builds that will be invoked and used to determine overall pass/fail before a potential push. The option --st-extra-builds is used to specify extra builds that will test ST packages (and also PT packages if any are enabled). If only PT packages are enabled then the builds specified in --st-extra-builds will still be run. The reasoning is that PT packages may contain extra ST features and therefore if the goal is to test these ST builds it is desirable to also run these builds because they also my impact downstream ST packages.

Finally, the option --extra-builds will test all enabled packages, including EX packages, regardless of their test group. Therefore, when using --extra-builds, be careful that you watch what packages are enabled. If you change an EX package, it will be enabled in --extra-builds builds.

A few use cases might help better demonstrate the behavior. Consider the following input arguments specifying extra builds

```
  --st-extra-builds=MPI_DEBUG_ST --extra-builds=INTEL_DEBUG
```

with the packages Teuchos, Phalanx, and Meros where Teuchos is PT, Phalanx is ST, and Meros is EX.

Here is what packages would be enabled in each of the builds:

```
  --default-builds=MPI_DEBUG,SERIAL_RELEASE \
  --st-extra-builds=MPI_DEBUG_ST \
  --extra-builds=INTEL_DEBUG
```

and which packages would be excluded:

```
A) --enable-packages=Teuchos:
   MPI_DEBUG:       [Teuchos]
   SERIAL_RELEASE:  [Teuchos]
   MPI_DEBUG_ST:    [Teuchos]
   INTEL_DEBUG:     [Teuchos]     Always enabled!

B) --enable-packages=Phalanx:
   MPI_DEBUG:       []            Skipped, no PT packages!
   SERIAL_RELEASE:  []            Skipped, no PT packages!
   MPI_DEBUG_ST:    [Phalanx]
```

```
   INTEL_DEBUG:     [Phalanx]

C) --enable-packages=Meros:
   MPI_DEBUG:       []            Skipped, no PT packages!
   SERIAL_RELEASE:  []            Skipped, no PT packages!
   MPI_DEBUG_ST:    []            Skipped, no PT or ST packages!
   INTEL_DEBUG:     [Meros]

D) --enable-packages=Teuchos,Phalanx:
   MPI_DEBUG:       [Teuchos]
   SERIAL_RELEASE:  [Teuchos]
   MPI_DEBUG_ST:    [Teuchos,Phalanx]
   INTEL_DEBUG:     [Teuchos,Phalanx]

E) --enable-packages=Teuchos,Phalanx,Meros:
   MPI_DEBUG:       [Teuchos]
   SERIAL_RELEASE:  [Teuchos]
   MPI_DEBUG_ST:    [Teuchos,Phalanx]
   INTEL_DEBUG:     [Teuchos,Phalanx,Meros]

The --extra-builds=INTEL_DEBUG build is always performed with all of the
enabled packages.  This logic given above must be understood in order to
understand the output given in the script.


Conventions for Command-Line Arguments:
---------------------------------------

The command-line arguments are segregated into three broad categories: a)
action commands, b) aggregate action commands, and c) others.

a) The action commands are those such as --build, --test, etc. and are shown
with [ACTION] in their documentation.  These action commands have no off
complement.  If the action command appears, then the action will be performed.

b) Aggregate action commands such as --do-all and --local-do-all turn on sets
of other action commands and are shown with [AGGR ACTION] in their
documentation.  The sub-actions that these aggregate action commands turn on
cannot be disabled with other arguments.

c) Other arguments are those that are not marked with [ACTION] or [AGGR
ACTION] tend to either pass in data and turn control flags on or off.


Exit Code:
---------

This script returns 0 if the actions requested are successful.  This does not
necessarily imply that it is okay to do a push.  For example, if only --pull
is passed in and is successful, then 0 will be returned but that does *not*
mean that it is okay to do a push.  A 0 return value is a necessary but not
sufficient condition for readiness to push.



Options:
  -h, --help            show this help message and exit
  --project-configuration=PROJECTCONFIGURATION
                        Custom file to provide configuration defaults for the
                        project.  By default, the file project-checkin-test-
```

```
                      config.py is looked for in <checkin-test-path>/../..
                      (assuming default <projectDir>/cmake/tribits/
                      directory structure and second is looked for in
                      <checkin-test-path>/ (which is common practice to
                      symlink the checkin-test.py script into the project's
                      base directory).  If this file is set to a location
                      that is not in the project's base directory, then
                      --src-dir must be set to point to the project's base
                      directory.
--show-defaults       Show the default option values and do nothing at all.
--project-name=PROJECTNAME
                      Set the project's name. This is used to locate various
                      files.
--eg-git-version-check
                      Enable automatic check for the right versions of eg
                      and git. [default]
--no-eg-git-version-check
                      Do not check the versions of eg and git, just trust
                      they are okay.
--src-dir=SRCDIR      The source base directory for code to be tested.  The
                      default is determined by the location of the found
                      project-checkin-test-config.py file.
--default-builds=DEFAULTBUILDS
                      Comma separated list of builds that should always be
                      run by default.
--extra-repos-file=EXTRAREPOSFILE
                      File path to an extra repositories list file.  If set
                      to 'project', then
                      <project_dir>/cmake/ExtraRepositoriesList.cmake is
                      read.  See the argument --extra-repos for details on
                      how this list is used (default empty '')
--extra-repos-type=EXTRAREPOSTYPE
                      The test type of repos to read from
                      <extra_repos_file>. Choices = ('', 'Continuous',
                      'Nightly', 'Experimental').  [default = '']
--extra-repos=EXTRAREPOS
                      List of comma separated extra repositories containing
                      extra  packages that can be enabled.  The order these
                      repos is listed in not important.  This option
                      overrides --extra-repos-file.
--ignore-missing-extra-repos
                      If set, then extra repos read in from
                      <extra_repos_file> will be ignored and removed from
                      list.  This option is not applicable if
                      <extra_repos_file>==''  or <extra_repos_type>==''.
--require-extra-repos-exist
                      If set, then all listed extra repos must exist or the
                      script will exit. [default]
--with-cmake=WITHCMAKE
                      CMake executable to use with cmake -P scripts
                      internally (only set by unit testing code).
--skip-deps-update    If set, skip the update of the dependency XML file.
                      If the package structure has not changed since the
                      last invocation, then it is safe to use this option.
--enable-packages=ENABLEPACKAGES
                      List of comma separated packages to test changes for
                      (example, 'Teuchos,Epetra').  If this list of packages
                      is empty, then the list of packages to enable will be
                      determined automatically by examining the set of
                      modified files from the version control update log.
```

```
                      Note that this will skip the auto-detection of changed
                      packages based on changed files.
--disable-packages=DISABLEPACKAGES
                      List of comma separated packages to explicitly disable
                      (example, 'Tpetra,NOX').  This list of disables will
                      be appended after all of the listed enables no mater
                      how they are determined (see --enable-packages
                      option).  NOTE: Only use this option to remove
                      packages that will not build for some reason.  You can
                      disable tests that run by using the CTest option -E
                      passed through the --ctest-options argument in this
                      script.
--enable-all-packages=ENABLEALLPACKAGES
                      Determine if all packages are enabled 'on', or 'off',
                      or 'auto' (let other logic decide).  Setting to 'off'
                      is appropriate when the logic in this script
                      determines that a global build file has changed but
                      you know that you don't need to rebuild and test every
                      package for a reasonable test.  Setting --enable-
                      packages effectively disables this option.  Setting
                      this to 'off' does *not* stop the forward enabling of
                      downstream packages for packages that are modified or
                      set by --enable-packages. Setting this to 'on' will
                      skip the automatic detection of changed packages based
                      on changed files.  It can be helpful to stop the auto-
                      detection changed packages when there are thousands of
                      changed files and hundreds of defined packages.
                      Choices = ('auto', 'on', 'off').  [default = 'auto']
--enable-fwd-packages
                      Enable forward packages. [default]
--no-enable-fwd-packages
                      Do not enable forward packages.
--continue-if-no-updates
                      If set, then the script will continue if no updates
                      are pulled from any repo. [default]
--abort-gracefully-if-no-updates
                      If set, then the script will abort gracefully if no
                      updates are pulled from any repo.
--continue-if-no-changes-to-push
                      If set, then the script will continue if no changes to
                      push from any repo. [default]
--abort-gracefully-if-no-changes-to-push
                      If set, then the script will abort gracefully if no
                      changes to push from any repo.
--continue-if-no-enables
                      If set, then the script will continue if no packages
                      are enabled. [default]
--abort-gracefully-if-no-enables
                      If set, then the script will abort gracefully if no
                      packages are enabled.
--extra-cmake-options=EXTRACMAKEOPTIONS
                      Extra options to pass to 'cmake' after all other
                      options. This should be used only as a last resort.
                      To disable packages, instead use --disable-packages.
                      To change test categories, use --test-categories.
--test-categories=TESTCATEGORIES
                      . Change the test categories.  Can be 'BASIC',
                      'CONTINUOUS',  'NIGHTLY', or 'WEEKLY' (default
                      'BASIC').
-j OVERALLNUMPROCS    The options to pass to make and ctest (e.g. -j4).
```

```
--make-options=MAKEOPTIONS
                      The options to pass to make (e.g. -j4).
--ctest-options=CTESTOPTIONS
                      Extra options to pass to 'ctest' (e.g. -j2).
--ctest-timeout=CTESTTIMEOUT
                      timeout (in seconds) for each single 'ctest' test
                      (e.g. 180 for three minutes).
--show-all-tests      Show all of the tests in the summary email and in the
                      commit message summary (see --append-test-results).
--no-show-all-tests   Don't show all of the test results in the summary
                      email. [default]
--without-default-builds
                      Skip the default builds (same as --default-builds='').
                      You would use option along with --extra-
                      builds=BUILD1,BUILD2,... to run your own local custom
                      builds.
--st-extra-builds=STEXTRABUILDS
                      List of comma-separated ST extra build names.  For
                      each of the build names in --st-extra-
                      builds=<BUILD1>,<BUILD2>,..., there must be a file
                      <BUILDN>.config in the local directory along side the
                      COMMON.config file that defines the special build
                      options for the extra build.
--ss-extra-builds=SSEXTRABUILDS
                      DEPRECATED!  Use --st-extra-builds instead!.
--extra-builds=EXTRABUILDS
                      List of comma-separated extra build names.  For each
                      of the build names in --extra-
                      builds=<BUILD1>,<BUILD2>,..., there must be a file
                      <BUILDN>.config in the local directory along side the
                      COMMON.config file that defines the special build
                      options for the extra build.
--send-email-to=SENDEMAILTO
                      List of comma-separated email addresses to send email
                      notification to after every build/test case finishes
                      and at the end for an overall summary and push status.
                      By default, this is the email address you set for git
                      returned by `git config --get user.email`.  In order
                      to turn off email notification, just set --send-email-
                      to='' and no email will be sent.
--skip-case-send-email
                      If set then if a build/test case is skipped for some
                      reason (i.e. because no packages are enabled) then an
                      email will go out for that case. [default]
--skip-case-no-email  If set then if a build/test case is skipped for some
                      reason (i.e. because no packages are enabled) then no
                      email will go out for that case. [default]
--send-email-for-all  If set, then emails will get sent out for all
                      operations. [default]
--send-email-only-on-failure
                      If set, then emails will only get sent out for
                      failures.
--send-email-to-on-push=SENDEMAILTOONPUSH
                      List of comma-separated email addresses to send email
                      notification to on a successful push.  This is used to
                      log pushes to a central list.  In order to turn off
                      this email notification, just set --send-email-to-on-
                      push='' and no email will be sent to these email
                      lists.
--force-push          Force the local push even if there are build/test
```

```
                      errors. WARNING: Only do this when you are 100%
                      certain that the errors are not caused by your code
                      changes.  This only applies when --push is specified
                      and this script.
--no-force-push       Do not force a push if there are failures. [default]
--do-push-readiness-check
                      Check the push readiness status at the end and send
                      email if not actually pushing. [default]
--skip-push-readiness-check
                      Skip push status check.
--rebase              Rebase the local commits on top of origin/master
                      before amending the last commit and pushing.  Rebasing
                      keeps a nice linear commit history like with CVS or
                      SVN and will work perfectly for the basic workflow of
                      adding commits to the 'master' branch and then syncing
                      up with origin/master before the final push. [default]
--no-rebase           Do not rebase the local commits on top of
                      origin/master before amending the final commit and
                      pushing.  This allows for some more  complex workflows
                      involving local branches with multiple merges.
                      However, this will result in non-linear history and
                      will allow for trivial merge commits with
                      origin/master to get pushed.  This mode should only be
                      used in cases where the rebase mode will not work or
                      when it is desired to use a merge commit to integrate
                      changes on a branch that you wish be able to easily
                      back out.  For sophisticated users of git, this may in
                      fact be the preferred mode.
--append-test-results
                      Before the final push, amend the most recent local
                      commit by appending a summary of the test results.
                      This provides a record of what builds and tests were
                      performed in order to test the local changes.  This is
                      only  performed if --push is also set.  NOTE: If the
                      same local commit is amended more than once, the prior
                      test summary sections will be overwritten with the
                      most recent test results from the current run.
                      [default]
--no-append-test-results
                      Do not amend the last local commit with test results.
                      NOTE: If you have uncommitted local changes that you
                      do not want this script to commit then you must select
                      this option to avoid this last amending commit.  Also,
                      if you are pushing commits from a shared branch and
                      don't want to change any of the SHA1s for the commits,
                      then you must set this option!
--extra-pull-from=EXTRAPULLFROM
                      Optional extra git pull '<repository>:<branch>' to
                      merge in changes from after pulling in changes from
                      'origin'.  This option uses a colon with no spaces in
                      between <repository>:<branch>' to avoid issues with
                      passing arguments with spaces.  For example --extra-
                      pull-from=machine:/base/dir/repo:master.  This extra
                      pull is only done if --pull is also specified.  NOTE:
                      when using --extra-repo=REPO1,REPO2,... the
                      <repository> must be a named repository that is
                      present in all of the git repos or it will be an
                      error.
--allow-no-pull       Allowing for there to be no pull performed and still
                      doing the other actions.  This option is useful for
```

```
                        testing against local changes without having to get
                        the updates from the global repo.  However, if you
                        don't pull, you can't push your changes to the global
                        repo.  WARNING: This does *not* stop a pull attempt
                        from being performed by --pull or --do-all!
  --wipe-clean          [ACTION] Blow existing build directories and
                        build/test results.  The action can be performed on
                        its own or with other actions in which case the wipe
                        clean will be performed before any other actions.
                        NOTE: This will only wipe clean the builds that are
                        specified and will not touch those being ignored (e.g.
                        SERIAL_RELEASE will not be removed if --default-
                        builds=MPI_DEBUG is specified).
  --pull                [ACTION] Do the pull from the default (origin)
                        repository and optionally also merge in changes from
                        the repo pointed to by --extra-pull-from.
  --configure           [ACTION] Do the configure step.
  --build               [ACTION] Do the build step.
  --test                [ACTION] Do the running of the enabled tests.
  --local-do-all        [AGGR ACTION] Do configure, build, and test with no
                        pull (same as setting --allow-no-pull ---configure
                        --build --test).  This is the same as --do-all except
                        it does not do --pull and also allows for no pull.
  --do-all              [AGGR ACTION] Do update, configure, build, and test
                        (same as --pull --configure --build --test).  NOTE:
                        This will do a --pull regardless if --allow-no-pull is
                        set or not.  To avoid the pull, use --local-do-all.
  --push                [ACTION] Push the committed changes in the local repo
                        into to global repo 'origin' for the current branch.
                        Note: If you have uncommitted changes this command
                        will fail.  Note: You must have SSH public/private
                        keys set up with the origin machine (e.g.
                        software.sandia.gov) for the push to happen without
                        having to type your password.
  --execute-on-ready-to-push=EXECUTEONREADYTOPUSH
                        [ACTION] A command to execute on successful execution
                        and 'READY TO PUSH' status from this script.  This can
                        be used to do a remote SSH invocation to a remote
                        machine to do a remote pull/test/push after this
                        machine finishes.
```

14.8 install_devtools.py --help

Below is a snapshot of the output from `install_devtools.py --help`.

```
Usage: install-devtools.py [OPTIONS]

This script drives the installation of a number of tools needed by many
TriBITS-based projects.  The most typical usage is to first create a scratch
directory with::

  mkdir scratch
  cd scratch

and then run:

  install-devtools.py --install-dir=<dev_env_base> \
   --parallel=<num-procs> --do-all
```

```
By default, this installs the following tools in the dev env install
directory:

  <dev_env_base>/
    common_tools/
      autoconf-<autoconf-version>/
      cmake-<cmake-version>/
      git-<git-version>/
      gitdist
    gcc-<gcc-version>/
      load_dev_env.[sh,csh]
      toolset/
        gcc-<gcc-version>/
        mpich-<mpich-version>/

The default versions of the tools installed are:

 autoconf-2.69
 cmake-2.8.11
 git-1.7.0.4
 gcc-4.8.3
 mpich-3.1.3

The tools installed under common_tools/ only need to be installed once
independent of any compilers that may be used to build TriBITS-based projects.

The tools installed under gcc-<gcc-version>/ are specific to a GCC compiler
and MPICH configuration and build.

The download and install of each of these tools is drive by its own
install-<toolname>.py script in the same directory as install-devtools.py.

Before running this script, some version of a C and C++ compiler must already
be installed on the system.

At a high level, this script performs the following actions.

1) Create the base directories (if they don't already exist) and install
   load_dev_env.sh (csh).  (if --initial-setup is passed in.)

2) Download the sources for all of the requested common tools and compiler
   toolset.  (if --download is passed in.)

3) Configure, build, and install the requested common tools under
   common_tools/. (if --install is passed in.)

4) Configure, build, and install the downloaded GCC and MPICH tools.  First
   install GCC then MPICH using the installed GCC and install under
   gcc-<gcc-version>/.  (if --install is passed in.)

The informational arguments to this function are:

  --install-dir=<dev_env_base>

    The base directory that will be used for the install.  There is not
    default.  If this is not specified then it will abort.

  --source-git-url-base=<url_base>

    Gives the base URL for to get the tool sources from.  The default is:
```

```
      https://github.com/tribitsdevtools/

    This is used to build the full git URL as:

      <url_base><tool_name>-<tool_version>-base

    This can also accomidate gitolite repos and other directory structures,
    for example, with:

      git@<host-name>:prerequisites/

  --common-tools=all

    Specifies the tools to download and install under common_tools/.  One can
    pick specific tools with:

      --common-tools=git,cmake,...

    This will download and install the default versions of these tools.  To
    select specific versions, use:

      --common-tools=git:1.7.0.4,cmake:2.8.11,...

    The default is 'all'.  To install none of these, pass in empty:

      --common-tools=''

  --compiler-toolset=all

    Specifies GCC and MPICH (and other compiler-specific tools) to download
    and install under gcc-<gcc-version>/toolset/.  One can pick specific
    componets with:

      --compiler-toolset=gcc,mpich

    or specific versions with:

      --compiler-toolset=gcc:4.8.3,mpich:3.1.3

    Of course if one is only installing GCC with an existing installed MPICH,
    one will need to also reinstall MPICH as well.

    The default is 'all'.  To install none of these, pass in empty:

      --compiler-toolset=''

The action argumnets are:

  --initial-setup: Create <dev_env_base>/ directories and install
    load_dev_env.sh

  --download: Download all of the requested tools

  --install: Configure, build, and install all of the requested tools

  --do-all: Do everything.  Implies --initial-setup --downlaod --install

To change modify the permissions of the installed files, see the options
--install-owner, --install-group, and --install-for-all.
```

```
Note that the user can see what operations and command would be run without
actually running them by passing in --no-op.  This can be used to show how to
run each of the individual install command so that the user can run it for
him/her-self and customize it as needed.

If the user needs more customization, then they can just run with --do-all
--no-op and see what commands are run to install things and then they can run
the commands themselves manually and make whatever modifications they need.


Options:
  -h, --help            show this help message and exit
  --install-dir=INSTALLDIR
                        The base directory <dev_env_base> that will be used
                        for the install.  There is not default.  If this is
                        not specified then will abort.
  --install-owner=INSTALLOWNER
                        If set, then 'chown -R <install-owner> <install-dir>'
                        will be run after install.  Note that you can only
                        change the owner when running this script as sudo.
  --install-group=INSTALLGROUP
                        If set, then 'chgrp -R <install-group> <install-dir>'
                        and 'chmod -R g+rX <install-dir> will be run after
                        install.  Note that you can only change a to a group
                        that the owner is a member of.
  --install-for-all     If set, then 'chmod -R a+rX <install-dir>' will be run
                        after install.
  --no-install-for-all  If set, then <install-dir> is not opened up to
                        everyone.
  --source-git-url-base=SOURCEGITURLBASE
                        Gives the base URL <url_base> for the git repos to
                        object the source from.
  --load-dev-env-file-base-name=LOADDEVENVFILEBASENAME
                        Base name of the load dev env script that will be
                        installed.  (Default = 'load_dev_env')
  --common-tools=COMMONTOOLS
                        Specifies the common tools to download and install
                        under common_tools/.  Can be 'all', or empty '', or
                        any combination of 'gitdist,autoconf,cmake,git'
                        (separated by commas, no spaces).
  --compiler-toolset=COMPILERTOOLSET
                        Specifies GCC and MPICH and other compiler-specific
                        tools to download and install under gcc-<gcc-
                        version>/toolset/.  Can be 'all', or empty '', or any
                        combination of 'gcc,mpich' (separated by commas, no
                        spaces).
  --parallel=PARALLELLEVEL
                        Number of parallel processes to use in the build.  The
                        default is just '1'.  Use something like '8' to get
                        faster parallel builds.
  --do-op               Do all of the requested actions [default].
  --no-op               Skip all of the requested actions and just print what
                        would be done.
  --show-defaults       [ACTION] Show the defaults and exit.
  --initial-setup       [ACTION] Create base directories under <dev_env_base>/
                        and install load_dev_env.[sh,csh].
  --download            [ACTION] Download all of the tools specified by
                        --common-tools and --compiler-toolset.  WARNING:  If
                        the source for a tool has already been downloaded, it
```

```
                        will be deleted (along with the build directory) and
                        downloaded from scratch!
--install               [ACTION] Configure, build, and install all of the
                        tools specified by --common-tools and --compiler-
                        toolset.
--show-final-instructions
                        [ACTION] Show final instructions for using the
                        installed dev env.
--do-all                [AGGR ACTION] Do everything.  Implies --initial-setup
                        --downlaod --install --show-final-instructions
```