The Cxx Dual Library

Edward Diener

Table of Contents

Introduction

The Cxx dual library, or CXXD for short, is a macro library which chooses between using a Boost library or its C++ standard equivalent library for a number of different C++ implementations, while using the same code to program either choice. An 'implementation' is a Boost library which has a C++ standard library equivalent whose public interfaces are nearly the same in both cases. An 'implementation' is called a 'dual library' in this documentation, or 'CXXD-mod' for short.

The library does this by defining object-like macros for including the appropriate header file(s) and namespace for using either the Boost library version or the C++ standard library version of a particular dual library. Alternatively, for those who prefer not to use macros, the library provides an implementation header file for including the appropriate header file(s) and for setting a namespace alias for using either the Boost library version or the C++ standard library version of a particular dual library.

CXXD currently supports twenty eight different dual libraries, where the Boost version and the C++ standard version is nearly interchangeable. CXXD also provides a macro-based solution for distinguishing between the Boost version and the C++ standard version of a dual library so that specific code for a particular dual library choice may be written in those cases where the public interfaces diverge.

The default algorithm for choosing between the Boost library or the C++ standard library for any of the different dual libraries is that if the C++ standard library is available, it is chosen else the Boost library is chosen. The default algorithm can be overridden by the end-user. CXXD uses the Boost.Config library for determining whether ot not the C++ standard library is available for a dual library. The choice is made at compile time.

Who the library is for

The CXXD library is for:

1. Programmers writing code not using C++11 on up syntax who still want to target some C++11 on up libraries if the code is compiled in C++11 on up mode.

2. Programmers writing code using C++11 on up syntax who still want the option of targeting some Boost libraries if the equivalent C++11 library does not exist for a given implementation.

The problem

Particular functionality of a number of Boost libraries is duplicated to a great degree by C++ standard libraries, most of which were originally created from the equivalent Boost libraries. In designing software the choice of a particular library to use is often predicated on what is available when the end-user compiles the sources for a module.

Given a Boost library, call it BBB, which has an equivalent C++ standard library, call it SSS, how does the designer of software choose whether to use BBB or SSS in his code ?

The normal choice may be presented as:

1. Use BBB since it is always available for a given Boost distribution. This is by far the most popular and widely used decision for Boost library developers as well as end-users of Boost.

2. Use SSS and fail to compile if the C++ standard library is not available given the compiler options and the C++ compiler implementation. This is the most popular choice if the designer of the software insists on a certain level of C++ standard support, such as that the software is compiled so that the C++11 standard or higher is supported.

3. Support both BBB and SSS in the interface, with the proviso that if Boost is not being used BBB in the interface is not used and if the implementation and compiler options are not adequate then SSS in the interface is not used. Being "used" or not is done through preprocessing defines and preprocessing #if statements. This is the most time consuming solution and will result in much duplication of functionality to support both BBB and SSS.

As mentioned above the first choice is easily the most popular since Boost library developers, and those involved using any one of the approximately current 130+ Boost libraries, assume that a Boost distribution is always available for the programmer to use. If you see this choice as always the single most viable one you need not continue reading about this library and can move on to what you have always used.

What are the possible downsides to the first choice ?

- Some programmers, programming groups, businesses and large corporation employing programmers, do not like the idea of having to rely on the Boost distribution as a whole while feeling it is fine to rely on certain individual Boost libraries.
- Many programmers would like to use the C++ standard libraries when available with their compiler implementation rather than have a dependency in their code on the equivalent Boost library.
- Programmers may be already using C++ standard library SSS in their code and do not want to have to therefore use the Boost library equivalent BBB for a particular library interface.

The second choice, always using the C++ standard equivalent library, will occur less often because of its most obvious downside, which is that if the C++ standard library is not available for a particular compiler implementation and C++ standard compiler level the code will fail to compile. If however you write a library for a particular level of the C++ standard, such as C++11, and assume a strong implementation of that standard is needed by certain compilers which can compile your code, this is often your most viable choice and you need not continue reading about this library.

The third choice, supporting both the Boost version of a library and the equivalent C++ standard version of that library, is obviously programmable but entails a much greater amount of work. Each usage of BBB and SSS will entail writing code that supports both libraries and this will require a great deal of extra code. Furthermore your code will be filled with uses of preprocessor #if statements to delineate which usage of BBB or SSS would be available at any given time. However if you are happy to do all the extra work to support functionality in both BBB and SSS in your code you need not continue reading about this library.

The solution

Any of the three choices presented above can be used and represent a solution to the problem presented. This particular library represents a fourth choice, which is more flexible than any of the three choices listed above, but also has some downsides just like each of the three choices previously discussed.

The solution presented by the CXXD library is a means of supporting either BBB or SSS with a single set of code, without having to worry which is being used at compile time. This is the most important factor when using CXXD. The single set of code will support both the Boost and C++ standard equivalents of a particular dual library, with the particular choice of which implementation for a dual library being made at compile time.

The solution is predicated on the fact that for BBB and SSS the exact same functionality is present for a very large amount of the implementation of both libraries, with the only difference being the namespace being used and the header file(s) needing to be included. The solution also offers the means to distinguish between BBB and SSS for the very small amount of functionality which may be different between the Boost and C++ standard version of a library.

The solution is not problem free but will work painlessly most of the time, and when it does not work will inform the end-user why during compilation with preprocessing errors. This documentation will also discuss issues with this solution when it does not work as desired.

The solution involves the use of macros, all starting with this library's common mnemonic of CXXD_, which allow a programmer to use the functionality of equivalent BBB and SSS libraries with a single set of code. Alternatively implementation header files and a namespace aliases can be used instead of macros.

How it works

There are two different modes of using CXXD. These are called the 'macro' mode and the 'alias' mode. The 'alias' mode internally uses the 'macro' mode but provides an alternative syntax for those who prefer to use macros as little as possible in their written code.

In both modes, for any given library called 'XXX', which has a Boost version BBB and an equivalent C++ standard library SSS, the user of the CXXD library includes a header file called '<boost/cxx_dual/XXX.hpp>'. In macro mode this is specifically done while in alias mode this is done internally.

In this documentation the library 'XXX" is called, for short, a 'CXXD-mod', its choices between either a Boost library version or an equivalent C++ standard library version is called a 'dual library', and the header to be included in order to use the CXXD-mod is called the 'CXXD-mod header'. The action of including a CXXD-header in the programmer's code is called 'including a CXXD-mod'.

Once the CXXD-mod's CXXD header is included, it will have three mandatory macro definitions and one optional macro definition, depending on the particular CXXD-mod. These definitions, for a particular 'XXX' CXXD-mod, are:

- CXXD_XXX_HEADER, the name of XXX's header file as used by the preprocessor '#include' statement. This is called the "CXXD-mod macro header" to distinguish it from the 'CXXD-mod header', the latter referring to to the initial CXXD-mod header file which must be included to generate the correct macros. Occasionally this CXXD-mod macro header is an internal CXXD header which includes more than one dual library header file.
- CXXD_XXX_NS, the name of XXX CXXD-mod's namespace. This is called the "CXXD-mod namespace".
- CXXD_HAS_STD_XXX, 1 if XXX CXXD-mod's C++ standard library version is being used or 0 if XXX CXXD-mod's Boost library version is being used.
- Optionally CXXD_XXX_MACRO(macro), transforms any XXX CXXD-mod 'macro' to its actual name.

The CXXD-mod header has all the logic in it for choosing the dual library but has no dependency itself on either dual library.

Using macro mode

In macro mode, for any given library called 'XXX', which has a Boost version BBB and an equivalent C++ standard library SSS, the user of the CXXD library starts by specifically including the CXXD-mod header file '<boost/cxx_dual/XXX.hpp>'.

The subsequent usage im macro mode of these CXXD_ macros in code for any given CXXD-mod, once the appropriate CXXD-mod header has been included, is very simple:

1. Include the correct CXXD-mod macro header file using '#include CXXD_XXX_HEADER'. This brings in whatever header fles are needed by the implementation.
2. Access any functionality in the CXXD-mod namespace using 'CXXD_XXX_NS::some_functionality etc...' or, if there are no name clashes with another namespace, through 'using namespace CXXD_XXX_NS; some_functionality etc...'.

Using alias mode

In alias mode, for any given library called 'XXX', which has a Boost version BBB and an equivalent C++ standard library SSS, the user of the CXXD library starts by including a header file called '<boost/cxx_dual/impl/XXX.hpp>'. This header file is called the CXXD-mod implementation header.

The implementation header file for 'XXX' internally includes the CXXD-mod header for 'XXX". After including the CXXD-mod header the implementation file automatically includes whatever header files are needed by the implementation and creates a namespace alias to the correct implementation namespace.

The subsequent usage in alias mode of this CXXD_ namespace alias in code for any given CXXD-mod, once the appropriate CXXD-mod implementation header has been included, is very simple:

1. Access any functionality in the CXXD-mod namespace using 'cxxd_xxx_ns::some_functionality etc...' or, if there are no name clashes with another namespace, through 'using namespace cxxd_xxx_ns; some_functionality etc...'. The mnemonic 'cxxd_xxx_ns' is a namespace alias for CXXD-mod XXX for either the Boost or rhe C++ standard library namespace of the dual library chosen.

In alias mode the end-user can first specifically include the CXXD-mod header for 'XXX' before including the CXXD-mod implementation header for 'XXX', but it is not necessary. In alias mode the end-user can still use the namespace macro of macro mode instead of the namespace alias whenever he so chooses, since they both refer to the same correct namespace.

Using either mode

Further usage of the CXXD_ macros in code for any given CXXD-mod, once the appropriate CXXD-mod header has been included, is:

1. Use any macro MMM, which is part of the CXXD-mod implementation, by specifying 'CXXD_XXX_MACRO(MMM)' as the macro name.
2. Optionally it is possible to write specific code for the Boost version of the CXXD-mod or the C++ standard version of the particular CXXD-mod by using the CXXD_HAS_STD_XXX macro with a preprocessor #if statement.

This further usage of CXXD in alias mode does involve the use of macros in end-user's code but it will occur much less, if not at all, compared to using the namespace alias which has been setup for the end-user.

Both the macro mode and the alias mode can be used interchangeably for a particular CXXD-mod in the same TU. The header file(s) included in macro mode by including the CXXD-mod macro header is the exact same header file(s) include when the CXXD-mod implementation header is included. The namespace alias in alias mode refers to the exact same namespace as the namespace macro in macro mode. The end-user can always include the CXXD-implementation header for a particular CXXD-mod, and use the namespace alias, even if previously only the CXXD-mod header had been included for that particular CXXD-mod and the namespace macro had been used.

Including and using a CXXD-mod

A programmer who, after including a CXXD-mod header for 'XXX' CXXD-mod, whether specifically in macro mode or internally in alias mode, proceeds to either include the CXXD-mod macro header in macro mode, or who merely includes the CXXD-mod implementation header in alias mode, is said to 'use a CXXD-mod'. Please note the distinction between including a CXXD-mod and using a CXXD-mod. Including a CXXD-mod just defines a set of macros and is not dependent on any other CXXD-mod Boost library or outside implementation. Using a CXXD-mod establishes dependency on either a Boost library or its equivalent C++ standard library.

In alias mode including the CXXD-mod implementation header immediately establishes a dependency on its dual library and uses a CXXD-mod. In macro mode the dependency on its dual library only occurs when including the CXXD-mod macro header.

Generalized example

Somewhere in your code you want to use the XXX CXXD-mod, without worrying whether you are using the Boost version or the C++ standard library version of XXX. Your code in general may look like:

Using macros

```
// Creates the macros for CXXD-mod XXX by including the appropriate CXXD-mod header
#include <boost/cxx_dual/XXX.hpp>
// Includes the appropriate CXXD-mod macro header file
#include CXXD_XXX_HEADER

void SomeFunction()
    {
   // Creates an object of some XXX type using the CXXD-mod namespace
   CXXD_XXX_NS::some_XXX_class a_xxx_object;
    // Use the a_xxx_object in your code...
    #if CXXD_HAS_STD_XXX
    // Some code specific to the C++ standard implementation of XXX
    #else
    // Otherwise some code specific to the Boost implementation of XXX
    #endif
    // Use XXX macro 'macro' in some context
    CXXD_XXX_MACRO(macro)
    }
```

Using aliases

```
// Includes the appropriate CXXD-mod implementation header file.
// This also includes the appropriate CXXD-mod header file first.
#include <boost/cxx_dual/impl/XXX.hpp>

void SomeFunction()
    {
   // Creates an object of some XXX type using the CXXD-mod namespace
   cxxd_xxx_ns::some_XXX_class a_xxx_object;
    // Use the a_xxx_object in your code...
    #if CXXD_HAS_STD_XXX
   // Some code specific to the C++ standard implementation of XXX
    #else
    // Otherwise some code specific to the Boost implementation of XXX
    #endif
    // Use XXX macro 'macro' in some context
    CXXD_XXX_MACRO(macro)
    }
```

By far the major use of these CXXD macros, which including a particular CXXD-mod header defines, will be in the use of the CXXD-mod macro header CXXD_XXX_HEADER, to include the correct header file(s) for XXX, and in the CXXD-mod namespace CXXD_XXX_NS, to use the functionality of XXX.

Alternatively the major use of CXXD functionality will be in the use of the cxxd_xxx_ns namespace alias for accessing the correct dual library namespace, once the correct XXX implementation header file has been included.

On the other hand CXXD_HAS_STD_XXX may rarely if ever be used and CXXD_XXX_MACRO(macro) may not even exist for a given XXX CXXD-mod.

Terminology

The CXXD library uses various terms which will subsequently be explained in full. This section of the documentation is just a reference for the terminology, which will be encountered later in the documentation, rather than a full explanation:

- Dual library = Chooses a particular Boost or C++ standard implementation
- CXXD-mod = CXXD dual library
- CXXD-mod header = header file for a CXXD-mod
- CXXD-mod macro header = header file macro for a CXXD-mod
- CXXD-mod implementation header = implementation header file for a CXXD-mod
- macro mode = using the CXXD-mod macro header and CXXD-mod namespace macro
- alias mode = using the CXXD-mod namespace alias
- header file = normal C++ header file
- mod-ID = preprocessing identifier for a CXXD-mod
- module = executable or shared/static library
- TU = translation unit
- variant = a non-header only library generated with a different name but the same CXXD code.

Modules

A dual library which CXXD supports with its system of macros is called a "CXXD-mod" in this documentation, to distinguish the name from the general term of "library". A CXXD-mod is a Boost library whose functionality largely exists as a C++ standard library, where the differences between the Boost library and its C++ standard equivalent are usually very minimal. An example of such a library would be the Boost regex library, whose equivalent is the C++ standard regex library, both offering the use of regular expressions in C++ with very largely the exact same syntax.

I want to make it clear that the choice of using, for example, Boost regex or C++ standard regex, has little to do with the technical quality of the library. Both versions are of an extremely high quality and this is true for every Boost library which has an equivalent C++ standard library version.

The CXXD-mods supported by CXXD are simply those Boost libraries which have a C++ standard equivalent where the syntactical use of either is very largely the same except for the namespace involved.

The following table lists the CXXD-mods supported by CXXD, in alphabetical order, and the appropriate information for each one. All header files are based off of the <boost/cxx_dual/> directory;

Table 1. CXXD-Mods

CXXD-Mod	Header file	Implementation header file	Include macro	Namespace macro	Namespace alias	Choice macro
array	array.hpp	impl/array.hpp	CXXD_ARRAY_HEADER	CXXD_ARRAY_NS	cxxd_array_ns	CXXD_HAS_STD_ARRAY
atomic	atomic.hpp	impl/atomic.hpp	CXXD_ATOMIC_HEADER	CXXD_ATOMIC_NS	cxxd_atomic_ns	CXXD_HAS_STD_ATOMIC
bind	bind.hpp	impl/bind.hpp	CXXD_BIND_HEADER	CXXD_BIND_NS	cxxd_bind_ns	CXXD_HAS_STD_BIND
chrono	chrono.hpp	impl/chrono.hpp	CXXD_CHRONO_HEADER	CXXD_CHRONO_NS	cxxd_chrono_ns	CXXD_HAS_STD_CHRONO
condition_variable	condition_variable.hpp	impl/condition_variable.hpp	CXXD_CONDITION_VARIABLE_HEADER	CXXD_CONDITION_VARIABLE_NS	cxxd_condition_variable_ns	CXXD_HAS_STD_CONDITION_VARIABLE
enable_shared_from_this	enable_shared_from_this.hpp	impl/enable_shared_from_this.hpp	CXXD_ENABLE_SHARED_FROM_THIS_HEADER	CXXD_ENABLE_SHARED_FROM_THIS_NS	cxxd_enable_shared_from_this_ns	CXXD_HAS_STD_ENABLE_SHARED_FROM_THIS
function	function.hpp	impl/function.hpp	CXXD_FUNCTION_HEADER	CXXD_FUNCTION_NS	cxxd_function_ns	CXXD_HAS_STD_FUNCTION
hash	hash.hpp	impl/hash.hpp	CXXD_HASH_HEADER	CXXD_HASH_NS	cxxd_hash_ns	CXXD_HAS_STD_HASH
make_shared	make_shared.hpp	impl/make_shared.hpp	CXXD_MAKE_SHARED_HEADER	CXXD_MAKE_SHARED_NS	cxxd_make_shared_ns	CXXD_HAS_STD_MAKE_SHARED
mem_fn	mem_fn.hpp	impl/mem_fn.hpp	CXXD_MEM_FN_HEADER	CXXD_MEM_FN_NS	cxxd_mem_fn_ns	CXXD_HAS_STD_MEM_FN
move	move.hpp	impl/move.hpp	CXXD_MOVE_HEADER	CXXD_MOVE_NS	cxxd_move_ns	CXXD_HAS_STD_MOVE
mutex	mutex.hpp	impl/mutex.hpp	CXXD_MUTEX_HEADER	CXXD_MUTEX_NS	cxxd_mutex_ns	CXXD_HAS_STD_MUTEX
random	random.hpp	impl/random.hpp	CXXD_RANDOM_HEADER	CXXD_RANDOM_NS	cxxd_random_ns	CXXD_HAS_STD_RANDOM
ratio	ratio.hpp	impl/ratio.hpp	CXXD_RATIO_HEADER	CXXD_RATIO_NS	cxxd_ratio_ns	CXXD_HAS_STD_RATIO
ref	ref.hpp	impl/ref.hpp	CXXD_REF_HEADER	CXXD_REF_NS	cxxd_ref_ns	CXXD_HAS_STD_REF

CXXD-Mod	Header file	Implementation header file	Include macro	Namespace macro	Namespace alias	Choice macro
regex	regex.hpp	impl/regex.hpp	CXXD_REGEX_HEADER	CXXD_REGEX_NS	cxxd_regex_ns	CXXD_HAS_STD_REGEX
shared_mutex	shared_mutex.hpp	impl/shared_mutex.hpp	CXXD_SHARED_MUTEX_HEADER	CXXD_SHARED_MUTEX_NS	cxxd_shared_mutex_ns	CXXD_HAS_STD_SHARED_MUTEX
shared_ptr	shared_ptr.hpp	• impl/shared_ptr.hpp • impl/shared_ptr_only.hpp	• CXXD_SHARED_PTR_HEADER - all shared_ptr related header files • CXXD_SHARED_PTR_ONLY_HEADER - only the shared_ptr header file	CXXD_SHARED_PTR_NS	cxxd_shared_ptr_ns	CXXD_HAS_STD_SHARED_PTR
system_error	shared_ptr.hpp	impl/system_error.hpp	CXXD_SYSTEM_ERROR_HEADER	CXXD_SYSTEM_ERROR_NS	cxxd_system_error_ns	CXXD_HAS_STD_SYSTEM_ERROR
thread	thread.hpp	impl/thread.hpp	CXXD_THREAD_HEADER	CXXD_THREAD_NS	cxxd_thread_ns	CXXD_HAS_STD_THREAD
tuple	tuple.hpp	impl/tuple.hpp	CXXD_TUPLE_HEADER	CXXD_TUPLE_NS	cxxd_tuple_ns	CXXD_HAS_STD_TUPLE
type_index	type_index.hpp	impl/type_index.hpp	CXXD_TYPE_INDEX_HEADER	CXXD_TYPE_INDEX_NS	cxxd_type_index_ns	CXXD_HAS_STD_TYPE_INDEX
type_traits	type_traits.hpp	impl/type_traits.hpp	CXXD_TYPE_TRAITS_HEADER	CXXD_TYPE_TRAITS_NS	cxxd_type_traits_ns	CXXD_HAS_STD_TYPE_TRAITS
unordered_map	unordered_map.hpp	impl/unordered_map.hpp	CXXD_UNORDERED_MAP_HEADER	CXXD_UNORDERED_MAP_NS	cxxd_unordered_map_ns	CXXD_HAS_STD_UNORDERED_MAP
unordered_multimap	unordered_multimap.hpp	impl/unordered_multimap.hpp	CXXD_UNORDERED_MULTIMAP_HEADER	CXXD_UNORDERED_MULTIMAP_NS	cxxd_unordered_multimap_ns	CXXD_HAS_STD_UNORDERED_MULTIMAP
unordered_multiset	unordered_multiset.hpp	impl/unordered_multiset.hpp	CXXD_UNORDERED_MULTISET_HEADER	CXXD_UNORDERED_MULTISET_NS	cxxd_unordered_multiset_ns	CXXD_HAS_STD_UNORDERED_MULTISET
unordered_set	unordered_set.hpp	impl/unordered_set.hpp	CXXD_UNORDERED_SET_HEADER	CXXD_UNORDERED_SET_NS	cxxd_unordered_set_ns	CXXD_HAS_STD_UNORDERED_SET
weak_ptr	weak_ptr.hpp	impl/weak_ptr.hpp	CXXD_WEAK_PTR_HEADER	CXXD_WEAK_PTR_NS	cxxd_weak_ptr_ns	CXXD_HAS_STD_WEAK_PTR

The naming scheme for each CXXD-mod follows common conventions:

- Each header file is the CXXD-mod name preceded by 'boost/cxx_dual/' and followed by the '.hpp' extension.
- Each implementation header file is the CXXD-mod name preceded by 'boost/cxx_dual/impl/' and followed by the '.hpp' extension.
- Each include macro is named starting with the mnemonic CXXD_, followed by the CXXD-mod name in uppercase, followed by the mnemonic _HEADER.
- Each namespace macro is named starting with the mnemonic CXXD_, followed by the CXXD-mod name in uppercase, followed by the mnemonic _NS.
- Each namespace alias is named starting with the mnemonic cxxd_, followed by the CXXD-mod name, followed by the mnemonic _ns.
- Each choice macro is named starting with the mnemonic CXXD_HAS_STD_, followed by the CXXD-mod name in uppercase.

Not listed above in the table is a column for the macro of the form CXXD_XXX_MACRO, which is only defined for a CXXD-mod XXX which has equivalent macro names between the C++ standard library version and the Boost library version. This macro is used in the form of CXXD_XXX_MACRO(MACRO_NAME) to produce the correct macro name for the equivalent macro no matter which implementation is being used. Currently, among the CXXD-mods listed above, the only one which uses this form is the 'atomic' CXXD-mod. It's name therefore is CXXD_ATOMIC_MACRO and it can be used in the form of CXXD_ATOMIC_MACRO(SOME_ATOMIC_MACRO) to produce the correct equivalent macro name for the 'atomic' CXXD-mod.

For the 'shared_ptr' dual library the normal CXXD-mod macro header, CXXD_SHARED_PTR_HEADER, when included brings in header files for shared_ptr, weak_ptr, make_shared, and enable_shared_from_this for either the Boost library or the C++ standard library. The CXXD-mod macro header, CXXD_SHARED_PTR_ONLY_HEADER, when included brings in only the header file for shared_ptr for either the Boost library or the C++ standard library; the end-user can then use the separate functionality for weak_ptr, make_shared, and enable_shared_from_this in their respective CXXD-mods. In both cases the namespace macro 'CXXD_SHARED_PTR_NS' is set to the correct namespace for shared_ptr for either the Boost library or the C++ standard library. This namespace macro can also be used for weak_ptr, make_shared, and enable_shared_from_this namespaces when the CXXD_SHARED_PTR_HEADER is included.

Alternatively for the 'shared_ptr' dual library the normal implementation header file, <boost/cxx_dual/impl/shared_ptr.hpp>, when included brings in the header files for shared_ptr, weak_ptr, make_shared, and enable_shared_from_this for either the Boost library or the C++ standard library. The implementation header, <boost/cxx_dual/impl/shared_ptr_only.hpp>, when included brings in only the header file for shared_ptr for either the Boost library or the C++ standard library; the end-user can then use the separate CXXD functionality for weak_ptr, make_shared, and enable_shared_from_this in their respective CXXD-mods. In both cases the namespace alias 'cxxd_shared_ptr_ns' is set to the correct namespace for shared_ptr for either the Boost library or the C++ standard library. This namespace alias can also be used for weak_ptr, make_shared, and enable_shared_from_this namespaces when the <boost/cxx_dual/impl/shared_ptr.hpp> is included.

Using the CXXD-mods

The form of using the CXXD-mod information in a translation unit will now be given, choosing the regex library as an example.

Using macros	**Using aliases**

```
#include <boost/cxx_dual/regex.hpp>
#include CXXD_REGEX_HDR

void SomeFunction()
    {
   CXXD_REGEX_NS::regex re("A regular expres↲
sion etc.");
    bool res↲
ult(CXXD_REGEX_NS::regex_match("Some ↲
string...",re));
    // etc.
    }
```
	```
	#include <boost/cxx_dual/impl/regex.hpp>

	void SomeFunction()
	    {
	   cxxd_regex_ns::regex re("A regular expres↲
	sion etc.");
	    bool res↲
	ult(cxxd_regex_ns::regex_match("Some ↲
	string...",re));
	    // etc.
	    }
	```

In the example the code will work whether we are using the C++ standard regex library or the Boost regex library.

Using the CXXD_HAS_STD_... macro

The 'CXXD_HAS_STD_...' macro for any given CXXD-mod tests whether or not the C++ standard version of the library is being used, returning '1' if it is used or '0' if it is not used. Conversely the form '!CXXD_HAS_STD_...' for any given CXXD-mod tests whether or not the Boost version of the library is being used, returning '1' if the Boost version of the library is being used or '0' if it is not being used.

The CXXD_HAS_STD_... macro for any given CXXD-mod is a more understandable form of macro than Boost.Config already has for determining whether the compiler supports certain C++ libraries. Most of these macros are taken from whether or not a given BOOST_NO_CXX11_HDR_ is defined.

You may decide, for a particular code path which will occur very rarely, you need the C++ standard version of a particular library, rather than the Boost version, or else you do not want your code to compile even though you are using a particular CXXD-mod. As an example let's say that you want to create a compile error if the compiler does not support the C++ standard type_traits library, even though the Boost type_traits library could also normally be used.

```
#include <boost/cxx_dual/type_traits.hpp>
// Some_code
#if !CXXD_HAS_STD_TYPE_TRAITS
#error C++ standard type_traits library needed and not present.
#endif
// Further code
```

Another use for the CXXD_HAS_STD_... macro is to include particular header files rather than a main header file, for some given library functionality. This is more prevalent with Boost than with the C++ standard library, since the latter almost always has a single header file which includes library functionality for all parts of a library.

```
#include <boost/cxx_dual/type_traits.hpp>
#if CXXD_HAS_STD_TYPE_TRAITS
#include CXXD_TYPE_TRAITS_HDR
#else
#include <boost/type_traits/add_const.hpp>
#endif
// Further code using CXXD_TYPE_TRAITS_NS::add_const
```

You can also use the CXXD_HAS_STD_... macro to provide specific functionality depending on whether or not a particular CXXD-mod is using the C++ standard or Boost version. Of course you hope to minimize these situations but occasionally they happen:

Using macros	Using aliases

```
#include <boost/cxx_dual/thread.hpp>
#include CXXD_THREAD_HDR
// Code...

#if CXXD_HAS_STD_THREAD

// Functionality available if the C++ stand↲
ard version of the thread library is being ↲
used

#else

// Functionality available if the Boost ver↲
sion of the thread library is being used

#endif
```
	```
	#include <boost/cxx_dual/impl/thread.hpp>
	// Code...

	#if CXXD_HAS_STD_THREAD

	// Functionality available if the C++ stand↲
	ard version of the thread library is being ↲
	used

	#else

	// Functionality available if the Boost ver↲
	sion of the thread library is being used

	#endif
	```

Macro or alias

The main functionality in using any CXXD-mod, after including a CXXD-mod header, is to include the correct header file(s) and use the appropriate namespace for the dual library chosen.

The macro-based solution for accomplishing this is to use the include macro and the namespace macro for any given CXXD-mod, both defined in the CXXD-mod header file. This works perfectly well in all cases and is the simplest solution when using CXXD.

However many programmers do not like to see macros in their source code, or even in their header files where macros are more common. Therefore CXXD offers a solution to including the correct header file(s) and using the appropriate namespace for the dual library chosen which does not involve using macros at the end-user level. This solution involves including a CXXD implementation header file and subsequently using a C++ namespace alias defined in that implementation header file. Using a namespace alias instead of a namespace macro may be seen as more C++-like since namespace aliases are part of the C++ language. This solution reduces the use of CXXD macros in source files and may be seen as clearer syntax in C++ code.

Furthermore the solution of including the CXXD-mod implementation header, since it always includes the CXXD-mod header first, may be seen as an effective substitute for including the CXXD-mod header while providing header includes and a namespace alias all at the same time. Even when including the CXXD-mod implementation header the macros in the CXXD-mod header can still be used. It should be understood that when the CXXD-mod implementation header is included an immediate dependency is created on the dual library implementation chosen. This is not true when just including the CXXD-mod header.

When using the namespace alias for a CXXD-mod the namespace macro is still available for use. This is important because there are a few situations where using the namespace alias to designate the correct namespace does not work whereas using the namespace macro does work.

One of these situations is when the name of the dual library namespace is needed in a string as data or for output in a message. In this case the programmer can use a macro such as BOOST_STRINGIZE to convert the namespace macro to string representation, or can simply use the preprocessor stringize operator ('#').

Another of these situations is when you need to reopen the namespace of the dual library chosen for template specialization. You can do this, for some XXX CXXD-mod using code like:

```
namespace CXXD_XXX_NS
  {
  ... some template specialization code for a class template construct in 'XXX'
  }
```

where using the namespace alias to do so is not valid C++.

Both uses given above, of using the namespace macro where the namespace alias will not work, can be seen in the 'test_hash.cpp' test in the 'test' subdirectory of CXXD.

CXXD-mod identifiers

Each CXXD-mod also has an identifier, called a "mod-ID", by which that CXXD-mod is identified. The mod-ID is a VMD identifier. This means that it is registered and pre-detected so that CXXD can identify it in macro code.

The mod-IDs are not part of the individual CXXD-mod header files and do not take part in the individual CXXD-mod processing. Instead the mod-IDs are used in optional CXXD helper macros, which will be described later in the documentation, to allow the user of the particular macro to identify a particular CXXD-mod. Each mod-ID is simply the name of the CXXD-mod in upper case with 'CXXD_' prepended:

Table 2. Identifiers

CXXD-mod	mod-ID
array	CXXD_ARRAY
atomic	CXXD_ATOMIC
bind	CXXD_BIND
chrono	CXXD_CHRONO
condition_variable	CXXD_CONDITION_VARIABLE
enable_shared_from_this	CXXD_ENABLE_SHARED_FROM_THIS
function	CXXD_FUNCTION
hash	CXXD_HASH
make_shared	CXXD_MAKE_SHARED
mem_fn	CXXD_MEM_FN
move	CXXD_MOVE
mutex	CXXD_MUTEX
random	CXXD_RANDOM
ratio	CXXD_RATIO
ref	CXXD_REF
regex	CXXD_REGEX
shared_mutex	CXXD_SHARED_MUTEX
shared_ptr	CXXD_SHARED_PTR
system_error	CXXD_SYSTEM_ERROR
thread	CXXD_THREAD
tuple	CXXD_TUPLE
type_index	CXXD_TYPE_INDEX
type_traits	CXXD_TYPE_TRAITS
unordered_map	CXXD_UNORDERED_MAP
unordered_multimap	CXXD_UNORDERED_MULTIMAP
unordered_multiset	CXXD_UNORDERED_MULTISET
unordered_set	CXXD_UNORDERED_SET

CXXD-mod	**mod-ID**
weak_ptr	CXXD_WEAK_PTR

The mod-IDs have their own header file:

```
#include <boost/cxx_dual/mod_ids.hpp>
```

We will see a use for these CXXD-mod identifiers when I discuss optional helper macros which CXXD offers for the end-user later in the documentation. An end-user, who wishes to design his own macros to be used with CXXD, can use the mod-IDs as a means of specifying a particular CXXD-mod by including the header file above in his own code.

Mod header file dependency

When the programmer includes the appropriate CXXD-mod header file for a particular dual library from within the cxx_dual directory there is no dependency being established on the CXXD-mod itself, whether the CXXD-mod represents a Boost library or a C++ standard library. Any one of the CXXD-mod header files merely defines macros which the programmer may choose to use or not.

Therefore the CXXD library itself does not depend on any of the CXXD-mods it supports.

It is only when using a particular CXXD-mod macro header, along with a particular CXXD-mod namespace, or when including a particular CXXD-mod implementation header that a dependency is established.

Because there is no dependency on the CXXD-mod itself when a particular CXXD-mod header is included, CXXD allows inclusion of all CXDD-mod headers with a single include:

```
#include <boost/cxx_dual/cxx_mods.hpp>
```

This is called the "general mod header".

The general mod header includes macros, starting with CXXD_, for each CXXD-mod. As long as the prefix CXXD_ does not conflict with macros from any other software library in the translation unit there should be no problems even including this general mod header. Furthermore, since including any individual CXXD-mod header imposes no dependency on either the Boost version or the C++ standard version of the dual library itself, including the general mod header above introduces no dependencies on any of the CXXD-mods in that header file.

Choosing the dual library

This area of the documentation explains the mechanisms by which the CXXD library chooses whether to use the Boost version or the C++ standard version of a particular CXXD-mod.

The default algorithm

The default algorithm employed by CXXD to choose whether to use a Boost library or its C++ standard equivalent for any CXXD-mod is very simple. If the C++ standard library is available during compilation it is chosen and if it is not available during compilation its equivalent Boost library is chosen.

The determination of availability for any given CXXD-mod is done by examining Boost.Config macros. Boost.Config has quite a number of macros which will specify which C++ standard library is available during compilation. The logic of determining which library is available is determined within Boost.Config, and CXXD just uses the results of that logic to configure the macros for a particular CXXD-mod.

The logic of setting the macros for any particular CXXD-mod occurs when the CXXD-mod header for that CXXD-mod is included in the code. The logic is completely preprocessor macro based, and is specific to each CXXD-mod header, although the logic for each CXXD-mod header is generally the same.

As an example, using the regex CXXD-mod:

```
#include <boost/cxx_dual/regex.hpp>
... code
```

If Boost.Config determines that the C++ standard regex library is available CXXD_HAS_STD_REGEX is defined as '1', CXXD_REGEX_NS is defined as 'std', and CXXD_REGEX_HEADER is defined as '<regex>'. If Boost.Config determines that the C++ standard regex library is not available CXXD_HAS_STD_REGEX is defined as '0', CXXD_REGEX_NS is defined as 'boost', and CXXD_REGEX_HEADER is defined as '<boost/regex.hpp>'.

Overriding the default algorithm

Although CXXD chooses automatically whether the Boost library or the C++ standard library is to be used for any given CXXD-mod the programmer may decide to override this choice. The method of overriding the choice is to define a particular macro before including a particular CXXD-mod header.

The programmer may override the automatic choice of CXXD by specifying that the Boost library be used or by specifying that the C++ standard library be used. If the programmer specifies that the C++ standard library be used and that library is not available for use, a preprocessor error will normally be generated.

Specific overriding

For any given CXXD-mod 'xxx' defining a macro called 'CXXD_XXX_USE_BOOST', where XXX is the uppercase name of the CXXD-mod, specifies that the Boost library will be used for 'xxx'. If used CXXD_XXX_USE_BOOST must always be defined to nothing, as in:

```
#define CXXD_'XXX'_USE_BOOST
```

where XXX is the uppercase name of the CXXD-mod.

For any given CXXD-mod 'xxx' defining a macro called 'CXXD_XXX_USE_STD', where XXX is the uppercase name of the CXXD-mod, specifies that the C++ standard library will be used for 'xxx'. If used CXXD_XXX_USE_STD must always be defined to nothing, as in:

```
#define CXXD_'XXX'_USE_STD
```

where XXX is the uppercase name of the CXXD-mod.

The specific override macros for each CXXD-mod are:

Table 3. Specific override macros

CXXD-mod	Boost override macro	C++ standard override macro
array	CXXD_ARRAY_USE_BOOST	CXXD_ARRAY_USE_STD
atomic	CXXD_ATOMIC_USE_BOOST	CXXD_ATOMIC_USE_STD
bind	CXXD_BIND_USE_BOOST	CXXD_BIND_USE_STD
chrono	CXXD_CHRONO_USE_BOOST	CXXD_CHRONO_USE_STD
condition_variable	CXXD_CONDITION_VARIABLE_USE_BOOST	CXXD_CONDITION_VARIABLE_USE_STD
enable_shared_from_this	CXXD_ENABLE_SHARED_FROM_THIS_USE_BOOST	CXXD_ENABLE_SHARED_FROM_THIS_USE_STD
function	CXXD_FUNCTION_USE_BOOST	CXXD_FUNCTION_USE_STD
hash	CXXD_HASH_USE_BOOST	CXXD_HASH_USE_STD
make_shared	CXXD_MAKE_SHARED_USE_BOOST	CXXD_MAKE_SHARED_USE_STD
mem_fn	CXXD_MEM_FN_USE_BOOST	CXXD_MEM_FN_USE_STD
move	CXXD_MOVE_USE_BOOST	CXXD_MOVE_USE_STD
mutex	CXXD_MUTEX_USE_BOOST	CXXD_MUTEX_USE_STD
random	CXXD_RANDOM_USE_BOOST	CXXD_RANDOM_USE_STD
ratio	CXXD_RATIO_USE_BOOST	CXXD_RATIO_USE_STD
ref	CXXD_REF_USE_BOOST	CXXD_REF_USE_STD
regex	CXXD_REGEX_USE_BOOST	CXXD_REGEX_USE_STD
shared_mutex	CXXD_SHARED_MUTEX_USE_BOOST	CXXD_SHARED_MUTEX_USE_STD
shared_ptr	CXXD_SHARED_PTR_USE_BOOST	CXXD_SHARED_PTR_USE_STD
system_error	CXXD_SYSTEM_ERROR_USE_BOOST	CXXD_SYSTEM_ERROR_USE_STD
thread	CXXD_THREAD_USE_BOOST	CXXD_THREAD_USE_STD
tuple	CXXD_TUPLE_USE_BOOST	CXXD_TUPLE_USE_STD
type_index	CXXD_TYPE_INDEX_USE_BOOST	CXXD_TYPE_INDEX_USE_STD
type_traits	CXXD_TYPE_TRAITS_USE_BOOST	CXXD_TYPE_TRAITS_USE_STD
unordered_map	CXXD_UNORDERED_MAP_USE_BOOST	CXXD_UNORDERED_MAP_USE_STD

CXXD-mod	Boost override macro	C++ standard override macro
unordered_multimap	CXXD_UNORDERED_MULTIMAP_USE_BOOST	CXXD_UNORDERED_MULTIMAP_USE_STD
unordered_multiset	CXXD_UNORDERED_MULTISET_USE_BOOST	CXXD_UNORDERED_MULTISET_USE_STD
unordered_set	CXXD_UNORDERED_SET_USE_BOOST	CXXD_UNORDERED_SET_USE_STD
weak_ptr	CXXD_WEAK_PTR_USE_BOOST	CXXD_WEAK_PTR_USE_STD

If for any given CXXD-mod 'xxx' both CXXD_XXX_USE_BOOST and CXXD_XXX_USE_STD is defined a preprocessing error will occur when the CXXD header for 'xxx' is included.

As examples, using the regex CXXD-mod:

```
#define CXXD_REGEX_USE_BOOST
#include <boost/cxx_dual/regex.hpp>
... code
```

CXXD_HAS_STD_REGEX is defined as '0', CXXD_REGEX_NS is defined as 'boost', and CXXD_REGEX_HEADER is defined as '<boost/regex.hpp>'.

```
#define CXXD_REGEX_USE_STD
#include <boost/cxx_dual/regex.hpp>
... code
```

CXXD_HAS_STD_REGEX is defined as '1', CXXD_REGEX_NS is defined as 'std', and CXXD_REGEX_HEADER is defined as '<regex>' as long as the C++ standard regex library is available for use, else a preprocessing error is generated.

General overriding

Along with the specific macros of the form CXXD_XXX_USE_BOOST and CXXD_XXX_USE_STD which override default processing for the specific 'xxx' CXXD-mod the programmer can specify either of two generalized macros which overrides default processing for any CXXD-mod.

Defining a macro called CXXD_USE_BOOST specifies that the Boost library will be used for any CXXD-mod. If used CXXD_USE_BOOST must always be defined to nothing, as in:

```
#define CXXD_USE_BOOST
```

Defining a macro called CXXD_USE_STD specifies that the C++ standard library will be used for any CXXD-mod. If used CXXD_USE_STD must always be defined to nothing, as in:

```
#define CXXD_USE_STD
```

If both CXXD_USE_BOOST and CXXD_USE_STD is defined a preprocessing error will occur when any CXXD-mod header is included.

If for any given CXXD-mod 'xxx' both CXXD_'XXX'_USE_BOOST and CXXD_USE_STD is defined a preprocessing error will occur when the CXXD-mod header file for 'xxx' is included.

If for any given CXXD-mod 'xxx' both CXXD_'XXX'_USE_STD and CXXD_USE_BOOST is defined a preprocessing error will occur when the CXXD-mod header file for 'xxx' is included.

As examples, using the regex CXXD-mod:

```
#define CXXD_USE_BOOST
#include <boost/cxx_dual/regex.hpp>
... code
```

CXXD_HAS_STD_REGEX is defined as '0', CXXD_REGEX_NS is defined as 'boost', and CXXD_REGEX_HEADER is defined as '<boost/regex.hpp>'.

```
#define CXXD_USE_STD
#include <boost/cxx_dual/regex.hpp>
... code
```

CXXD_HAS_STD_REGEX is defined as '1', CXXD_REGEX_NS is defined as 'std', and CXXD_REGEX_HEADER is defined as '<regex>' as long as the C++ standard regex library is available for use, else a preprocessing error is generated.

```
#define CXXD_REGEX_USE_BOOST
#define CXXD_USE_STD
#include <boost/cxx_dual/regex.hpp>
... code
```

A preprocessing error is generated.

```
#define CXXD_REGEX_USE_STD
#define CXXD_USE_BOOST
#include <boost/cxx_dual/regex.hpp>
... code
```

A preprocessing error is generated.

Using any CXXD header:

```
#define CXXD_USE_BOOST
#define CXXD_USE_STD
#include <boost/cxx_dual/'any_cxxd_header'
... code
```

A preprocessing error is generated.

Purpose of overriding

The purpose of using overriding macros, for any CXXD-mod with the specific overriding macros or for CXXD as a whole with the general overriding macros, is to override the default algorithm which CXXD uses to choose a dual library. However using overriding macros for one's own direct use(s) of CXXD is generally foolish. This is because the easiest implementation, rather than using an overriding macro, is to just drop CXXD in favor of using either the Boost library or C++ standard library directly.

In other words, instead of:

Using macros	Using aliases

`#define CXXD_USE_BOOST`
`#include <boost/cxx_dual/regex.hpp>`
`#include CXXD_REGEX_HDR`
`CXXD_REGEX_NS::regex my_regex; // etc.`
`... code`
	`#define CXXD_USE_BOOST`
	`#include <boost/cxx_dual/impl/regex.hpp>`
	`cxxd_regex_ns::regex my_regex; // etc.`
	`... code`

you can simply code:

```
#include <boost/regex.hpp>
boost::regex my_regex; // etc.
... code
```

and instead of:

Using macros	Using aliases

`#define CXXD_USE_STD`
`#include <boost/cxx_dual/regex.hpp>`
`#include CXXD_REGEX_HDR`
`CXXD_REGEX_NS::regex my_regex; // etc.`
`... code`
	`#define CXXD_USE_STD`
	`#include <boost/cxx_dual/impl/regex.hpp>`
	`cxxd_regex_ns::regex my_regex; // etc.`
	`... code`

you can simply code:

```
#include <regex>
std::regex my_regex; // etc.
... code
```

In this sense the examples given above of using specific or general overriding macros are purely artificial in order to merely show what the meaning of these overriding macros entail. In actual code directly overriding a particular CXXD-mod should almost never be done in favor of directly using either the Boost or C++ standard implementations of a dual library.

So what is the actual purpose of the overriding macros ? The purpose is to override the use of CXXD by another implementation whose source files can not or should not be modified. A third-party library may choose to use CXXD to provide a dual library for a particular CXXD-mod. Your own code, which uses the third-party library but otherwise uses either the Boost version of the dual library or the C++ standard version of the dual library in all other situations, may enforce its own usage on the dual library interface of the third-party library by using an overriding macro. As an example a third-party library might have as a public interface:

Using macros	Using aliases

`// Header file ThirdPartyHeader.hpp`
`#include <boost/cxx_dual/regex.hpp>`
`#include CXXD_REGEX_HEADER`
`class ThirdPartyClass`
`    {`
`    public:`
`   void SomeFunction(CXXD_REGEX_NS::regex &);`
`    ... // other functionality`
`    };`
	`// Header file ThirdPartyHeader.hpp`
	`#include <boost/cxx_dual/impl/regex.hpp>`
	`class ThirdPartyClass`
	`    {`
	`    public:`
	`   void SomeFunction(cxxd_regex_ns::regex &);`
	`    ... // other functionality`
	`    };`

We will assume this is a header-only library for the time being (a discussion of using CXXD with a non-header-only library will follow later in the documentation). In one's own code you may desire to use the ThirdPartyClass::SomeFunction interface with specifically the Boost regex library only. Your own code might then look like:

```
#define CXXD_REGEX_USE_BOOST
#include <ThirdPartyHeader.hpp>
boost::regex my_regex("SomeRegularExpression etc.");
ThirdPartyClass tpclass;
tpclass.SomeFunction(my_regex);
```

Conversely you may decide that your use of the ThirdPartyClass::SomeFunction interface should be with the C++ standard regex library only, so your code might then look like:

```
#define CXXD_REGEX_USE_STD
#include <ThirdPartyHeader.hpp>
std::regex my_regex("SomeRegularExpression etc.");
ThirdPartyClass tpclass;
tpclass.SomeFunction(my_regex);
```

Instead if you did accept the dual nature of the ThirdPartyHeader.hpp header, your code might then like:

Using macros	Using aliases

```
#include <ThirdPartyHeader.hpp>
CXXD_REGEX_NS::regex my_regex("SomeRegularEx↲
pression etc.");
ThirdPartyClass tpclass;
tpclass.SomeFunction(my_regex);
```
	```
	#include <ThirdPartyHeader.hpp>
	cxxd_regex_ns::regex my_regex("SomeRegularEx↲
	pression etc.");
	ThirdPartyClass tpclass;
	tpclass.SomeFunction(my_regex);
	```

In other words using the overriding macros of CXXD serves the purpose of using some already established interface involving CXXD without changing that interface's own code, at the same time forcing the choice of a particular dual library to be used.

Overriding one's own interface

There is a situation where the programmer may choose to override his own use of a particular CXXD-mod. This is when he programs his own interface using a particular CXXD-mod but decides for a particular combination of compiler and/or OS that he wants to override that choice. In other words the programmer wishes to use a dual library for the benefits that CXXD has to offer but knows that for some combination of compiler and/or operating system that the particular dual library choice must be overridden to prevent a faulty implementation from being used.

Let's imagine that for a particular compiler version 'CCC' version 'NNN' on a particular OS 'OOO' we know that the standard library version of CXXD-mod 'XXX' is buggy. Also let's imagine that we can test for the buggy combination through using the preprocessor defines, as we can do in Boost through either Boost.Config or even more easily through Boost.Predef. Our pseudocode for overriding our own use of the dual library in a header file might look like:

Using macros	Using aliases

```
// Header file OwnOverrideHeader.hpp
#if CCCNNN && OOO
#define CXXD_XXX_USE_BOOST
#endif
#include <boost/cxxd/XXX.hpp>
#include CXXD_XXX_HEADER
class OwnOverrideClass
    {
    public:
    void SomeOwnOverrideFunc↲
tion(CXXD_XXX_NS::xxx_object &);
    ... // other functionality
    };
```
	```
	// Header file OwnOverrideHeader.hpp
	#if CCCNNN && OOO
	#define CXXD_XXX_USE_BOOST
	#endif
	#include <boost/cxxd/impl/XXX.hpp>
	class OwnOverrideClass
	    {
	    public:
	    void SomeOwnOverrideFunc↲
	tion(cxxd_xxx_ns::xxx_object &);
	    ... // other functionality
	    };
	```

So essentially where the purpose of the override macros is to change another interface's dual library choice for a particular CXXD-mod to a single implementation, you can use the override macros with one's own interface in order to provide one-off situations where you want to specifically control the implementation for a particular compiling environment.

Header file inclusion

CXXD-mod header content is processed each time a particular CXXD-mod header is included in a translation unit. This is different from most headers in C++, which use inclusion guards or compiler dependent pragmas so that the header file content only gets processed the first time.

Note

CXXD-mod implementation headers, after always including the appropriate CXXD-mod header, are subsequently only processed once in a TU.

The reason for processing CXXD-mod header content each time a CXXD-mod header is included is because overriding macros could be defined (or undefined) at any time within a translation unit. CXXD always makes sure that overriding macros do not conflict with each other, as discussed previously, and that the choice of either Boost or its equivalent standard library remains consistent for any particular CXXD-mod within a TU.

In actual usage the programmer himself will usually not include the same CXXD header more than once in a TU.

```
#include <boost/cxx_dual/regex.hpp>
... other #includes
#include <boost/cxx_dual/regex.hpp>
... code
```

The above is possible but will rarely happen.

What far more likely happens is that the programmer includes a non-CXXD header, often from another library, and that header will include a particular CXXD-mod header.

```
// Header another_library.hpp
#include <boost/cxx_dual/regex.hpp>
... other #includes
... header code
```

A TU with:

```
#include <boost/cxx_dual/regex.hpp>
#include <another_library.hpp>
... code
```

It is this latter case which will often cause a CXXD-mod header to be included more than once in a TU.

Default algorithm redux

Recall that the default algorithm is used to choose between the Boost library or its C++ standard equivalent for a CXXD-mod when no overriding macros are relevant for that CXXD-mod. Because CXXD-mod header content is processed each time the header is included the default algorithm is slightly different the second and subsequent times a particular CXXD-mod header is included. In that particular case the default algorithm simply accepts which of the dual libraries of the CXXD-mod has been previously chosen. This saves preprocessing time and also makes sure that the choice for a given CXXD-mod is consistent throughout the TU.

Let's look at how this works in practice with the default algorithm.

```
#include <boost/cxx_dual/regex.hpp>
... code
```

This is our normal case where the default algorithm will choose the C++ standard regex library if it is available, otherwise the Boost regex library.

```
// Header another_library.hpp
#include <boost/cxx_dual/regex.hpp>
... other #includes
... header code

// Header my_header.hpp
#include <boost/cxx_dual/regex.hpp>
#include <another_library.hpp>
... code
```

In this situation when the regex CXXD-mod is included a second time in my_header.hpp, by including another_library.hpp, it simply accepts the choice made the first time it was directly included.

```
// Header another_library.hpp
#include <boost/cxx_dual/regex.hpp>
... other #includes
... header code

// Header my_header.hpp
#define CXXD_REGEX_USE_BOOST
#include <boost/cxx_dual/regex.hpp>
#undef CXXD_REGEX_USE_BOOST
#include <another_library.hpp>
... code
```

In this case the first time that the regex CXXD-mod is included in my_header.hpp the default algorithm is not in effect since we have overridden the choice by specifying that the Boost regex library will be used. Although in practical experience it would be very unusual to undefine the overridden macro, we do it here to illustrate the fact that the second time that the regex CXXD-mod is included in my_header.hpp the default algorithm is in effect but it simply accepts the choice made the first time, which is to use the Boost regex library. This is done even if the C++ standard regex library is available.

Dual library consistency

By default whenever a CXXD-mod header for a particular CXXD-mod is included a second or subsequent time in a TU the choice of the dual library cannot change from the original inclusion of that CXXD-mod header. This consistency is enforced because it would normally cause problems in user's code if, for a given CXXD-mod, one part of the TU were using the C++ standard library and another part of the TU were using the Boost library equivalent. This consistency, of the same syntax meaning the same thing within a TU, is something which the CXXD library enforces. CXXD enforces this consistency by creating a preprocessing error if the dual library choice would change when a particular CXXD header is included more than once in a TU.

However this consistency means that the order of header file inclusion potentially changes the way that CXXD works for a given end-user's module. This is the major downside of CXXD so we will take a look at it. To illustrate this we will use as an example a header file which has an overriding macro:

```
// Header a_library.hpp
#include <boost/cxx_dual/regex.hpp>
... other #includes
... header code using the regex implementation

// Header another_library_with_override.hpp
#define CXXD_REGEX_USE_BOOST
#include <a_library.hpp>
... other #includes
... header code
```

In the another_library_with_override.hpp header file we override the default processing for the regex CXXD-mod so that the Boost regex library is always used. Let's also assume for our example that the C++ standard regex library is available when compiling our own TU. Now if we include the another_library_with_override.hpp header first in that TU followed by the CXXD regex header, as in:

```
#include <another_library_with_override.hpp>
...other header files
#include <boost/cxx_dual/regex.hpp>
```

everything works fine. This is because we have the overridden macro in effect each time to determine that the Boost regex library will be used.

But if we reverse the order of includes:

```
#include <boost/cxx_dual/regex.hpp>
...other header files
#include <another_library_with_override.hpp>
```

we are essentially changing the dual library choice in our TU, between the first time the regex CXXD-mod header is included and the second time it is included. The first time the regex CXXD-mod header is included no overriding macro is in effect so that the default algorithm chooses the C++ standard library because it is available. The second time the regex CXXD-mod header is included an overriding macro changes the regex CXXD-mod to use the Boost regex library. Thus consistency between which dual library is chosen is broken and CXXD creates a preprocessing error.

This is one of the weaknesses of a macro based system such as CXXD. Normally the order of inclusion of header files should not affect the way that code compiles. But in CXXD it does affect the compilation since, by default, CXXD does not allow the dual library choice for a particular CXXD-mod to change within a TU.

Turning off consistency

The end user can override the dual library consistency for all CXXD-mods by defining a macro, called CXXD_NO_CONSISTENCY. If used CXXD_NO_CONSISTENCY must always be defined to nothing, as in:

```
#define CXXD_NO_CONSISTENCY
```

Overriding dual library consistency for a TU is a serious matter and should not be done lightly. CXXD does not recommend doing this but in the spirit of flexibility allows it. Overriding consistency allows the dual library for a CXXD-mod to change when that CXXD-mod's CXXD header file is included more than once. In actuality, unless the end-user goes about undefining already defined overriding macros, the only case where overriding the dual library consistency by defining CXXD_NO_CONSISTENCY will normally cause the dual library for a CXXD-mod to change within a TU is in the second example given above, ie. when the default algorithm chooses a dual library the first time a CXXD-mod header is included and then an overriding macro changes the dual library with the opposite choice the second time the CXXD-mod header is included. In all other cases merely adding further override macros before further inclusion of the same CXXD-mod header will either cause overriding macro conflicts, which will always produce preprocessing errors from within CXXD, or not change the dual library chosen in any further way.

Using the CXXD_NO_CONSISTENCY macro will not work for header file includes and namespace aliases if a CXXD-mod implementation header file is used for a particular CXXD-mod, and the dual library choice for a CXXD-mod changes in a TU. This is because allowing the choice of a particular CXXD-mod's dual library to change in a TU means that the namespace alias for that CXXD-mod will change in that TU, and C++ does not allow that as part of the rules of the language. This is a weakness of the more C++-like namespaces alias compared to the namespace macro. Where the namespace macro can be legally undefined and redefined when a particular dual library choice changes in a TU, there is no way in C++ to "undefine" and "redefine" a namespace alias. Because a namespace alias cannot be redefined to some other namespace in C++, specific CXXD-mod implementation header files are processed only once in a TU by using header file guards. So while including a CXXD-mod implementation header file more than once will not produce a C++ compile error, the namespace alias in the implementation header file, as well as the header files included, will always reflect the first time the implementation header file was included. Therefore if a particular TU defines the CXXD_NO_CONSISTENCY macro, it should always itself use CXXD-mod macros to include the appropriate header files and specify the appropriate namespace, and not a CXXD-mod implementation header file.

It is highly recommended that CXXD_NO_CONSISTENCY not be used unless the programmer knows exactly what the code is doing in a particular TU including all intermediate header file code. When dual library consistency is turned off using CXXD_NO_CONSISTENCY the chance that CXXD for a particular CXXD-mod will access the Boost library in some part of a

TU and the C++ standard library in another part of a TU, and the problems this may cause, are generally not worth the problems that could occur in code confusion. Nonetheless if the programmer feels he knows what he is doing in using this macro he can do so.

An alternative to using the dangerous CXXD_NO_CONSISTENCY macro is to define the appropriate overriding macro at the very beginning of a TU, either in the TU itself or by some compiler command line parameter which allows a macro definition to be made. This maintains TU consistency but should really only be done if using CXXD produces a preprocessor error based on consistency for a TU being broken. As an extension to our example, if the code in our TU was:

```
#define CXXD_REGEX_USE_BOOST
#include <boost/cxx_dual/regex.hpp>
...other header files
#include <another_library_with_override.hpp>
```

we could avoid the dangerous CXXD_NO_CONSISTENCY macro while providing the override which another_library_with_override.hpp provides without incurring any preprocessing error.

Using in a library

Using CXXD in a library presents a number of situations which do not occur when using CXXD in an executable. These situations will be discussed here.

Specifying the interface

When using a particular Boost library within another library, let's call it MyLibrary, a class interface in a MyLibrary header file might look like:

```
// Header file MyHeader.hpp
#include <boost/regex.hpp>
class MyClass
    {
    public:
    void MyFunction(boost::regex &);
    ... // other functionality
    };
```

Documentation for MyClass::MyFunction would specify that it takes a single parameter which is a reference to a boost::regex object.

A user of MyLibrary's MyClass functionality might then look like:

```
#include <MyHeader.hpp>
MyClass an_object;
boost::regex a_regular_expression(...some regular expression etc.);
an_object.MyFunction(a_regular_expression);
// more code etc.
```

Similarly when using a C++ standard library equivalent to a particular Boost library within another library, let's call it again MyLibrary, a class interface in a MyLibrary header file might look like:

```
// Header file MyHeader.hpp
#include <regex>
class MyClass
    {
    public:
    void MyFunction(std::regex &);
    ... // other functionality
    };
```

Documentation for MyClass::MyFunction would specify that it takes a single parameter which is a reference to a std::regex object.

A user of MyLibrary's MyClass functionality might then look like:

```
#include <MyHeader.hpp>
MyClass an_object;
std::regex a_regular_expression(...some regular expression etc.);
an_object.MyFunction(a_regular_expression);
// more code etc.
```

Given these examples of using a Boost library or its C++ standard equivalent in another library we can then see how this works for creating an interface using CXXD in an example 'MyLibrary'. Our CXXD example would look like:

Using macros

```
// Header file MyHeader.hpp
#include <boost/cxx_dual/regex.hpp>
#include CXXD_REGEX_HEADER
class MyClass
    {
    public:
    void MyFunction(CXXD_REGEX_NS::regex &);
    ... // other functionality
    };
```

Using aliases

```
// Header file MyHeader.hpp
#include <boost/cxx_dual/impl/regex.hpp>
class MyClass
    {
    public:
    void MyFunction(cxxd_regex_ns::regex &);
    ... // other functionality
    };
```

Documentation for MyClass::MyFunction would specify that it takes a single parameter which is a reference to a CXXD_REGEX_NS::regex object, where 'CXXD_REGEX_NS' represents the namespace being used.

Alternatively documentation for MyClass::MyFunction would specify that it takes a single parameter which is a reference to a cxxd_regex_ns::regex object, where 'cxxd_regex_ns' represents the namespace being used.

A user of MyLibrary's MyClass functionality might look like:

Using macros

```
#include <MyHeader.hpp>
MyClass an_object;
CXXD_REGEX_NS::regex a_regular_expres↲
sion(...some regular expression etc.);
an_object.MyFunction(a_regular_expression);
// more code etc.
```

Using aliases

```
#include <MyHeader.hpp>
MyClass an_object;
cxxd_regex_ns::regex a_regular_expres↲
sion(...some regular expression etc.);
an_object.MyFunction(a_regular_expression);
// more code etc.
```

As in all these similar situations, once the user understands that 'CXXD_REGEX_NS' (or 'cxxd_regex_ns') represents the name of a namespace, functionality using MyLibrary's MyClass functionality could also be coded as:

Using macros

```
#include <MyHeader.hpp>
using namespace CXXD_REGEX_NS;
MyClass an_object;
regex a_regular_expression(...some regular ex↲
pression etc.);
an_object.MyFunction(a_regular_expression);
// more code etc.
```

Using aliases

```
#include <MyHeader.hpp>
using namespace cxxd_regex_ns;
MyClass an_object;
regex a_regular_expression(...some regular ex↲
pression etc.);
an_object.MyFunction(a_regular_expression);
// more code etc.
```

The important thing here is that the user of MyFunction understands that 'CXXD_REGEX_NS' (or 'cxxd_regex_ns') should always be used to designate the namespace without assuming that either Boost or the C++ standard equivalent of the regex library is being used. Obviously the same goes for any other CXXD-mod, where the CXXD namespace for any particular CXXD-mod should be consistently used as opposed to making any assumptions about whether the Boost version or the C++ standard of a CXXD-mod is being chosen. As long as the documentation for an interface using CXXD specifies this usage the end-user of such an interface should be able to use it correctly.

Overriding a CXXD-mod

As explained when generally discussing the purpose of the overriding macros it is usually foolish for code to override its own interface(s) which use CXXD, rather than simply dropping a CXXD dual library in order to directly use either a Boost library or its standard C++ equivalent library in the interface.

Overriding macros for a CXXD-mod in a library should normally occur when that CXXD-mod is being used in a second library and the second library doing the overriding consistently uses the Boost or C++ standard dual library for that CXXD-mod otherwise.

Given in OtherLibrary:

Using macros	Using aliases

`// Header file OtherHeader.hpp`
`#include <boost/cxx_dual/regex.hpp>`
`#include CXXD_REGEX_HEADER`
`class OtherClass`
`    {`
`    public:`
`    void OtherFunc↲`
`tion(CXXD_REGEX_NS::regex &);`
`    ... // other functionality`
`    };`
	`// Header file OtherHeader.hpp`
	`#include <boost/cxx_dual/impl/regex.hpp>`
	`class OtherClass`
	`    {`
	`    public:`
	`    void OtherFunc↲`
	`tion(cxxd_regex_ns::regex &);`
	`    ... // other functionality`
	`    };`

let us suppose that MyLibrary consistently uses the Boost regex library in other places. Now MyLibrary wants to use OtherLibrary's OtherClass::OtherFunction functionality with a Boost regex rather than let CXXD choose the default library for the regex CXXD-mod. The code for using OtherLibrary's OtherClass::OhterFunction functionality with a Boost regex in MyLibrary's own header file might look like:

```
// Header file MyHeader.hpp
#define CXXD_REGEX_USE_BOOST
#include <Otherheader.hpp>
class MyClass
    {
    public:
    void MyFunction(boost::regex &my_regex)
        {
        OtherClass oc;
        oc.OtherFunction(my_regex);
        ... // other functionality
        }
    ... // other functionality
    };
```

Subsequently in this case if MyLibrary's MyClass functionality is meant to be used by another library or executable the documentation should state that the regex CXXD-mod has been overridden to use boost::regex, and therefore MyHeader.hpp should be included before any other header file which might include the CXXD regex header. The reason for this was explained when discussing dual library consistency, where including a CXXD header more than once could lead to a preprocessing error if the CXXD header is subsequently overridden opposite to its initial default choice.

What is also important in a library is that the overriding macro be defined within the public header file of the library. In our example just above the

```
#define CXXD_REGEX_USE_BOOST
```

overriding macro must be in the header file itself so that any other code which uses the library, and includes the MyHeader.hpp file, picks up the override. This is different from an executable where an overriding macro could be passed on the command line when a particular TU is being compiled.

Use in a non-header only library

A non-header only library is a library which gets built into a shared library or a static library. In a traditional non-header only library all the source code of the library is built into the shared library or static library. A non-header only library also could contain some code which is header only along with the code which consists of the built portion of the library. Because the built portion of a non-header library is code whose functionality is fixed a non-header only library presents a more difficult problem when CXXD is used.

The problem

The reason why a non-header only library presents a problem if CXXD is used is because CXXD does its work at compile time. This means that once the built portions of a non-header only library get compiled and linked the choice for any of the CXXD dual libraries has already been made and essentially encoded into the resulting shared or static library. This choice is based on the compiler and its command line switches when the non-header only library is built. The ability of CXXD to choose, for the end-user of the built portion of the library, at compile time the particular dual library being used is eliminated in such a case.

The solution

But there is a solution. It consists of generating more than one variant for the non-header only library based on different CXXD dual possibilities. Each variant would have:

- The same code, with more or less the same functionality, based on which dual library(s) are chosen.
- A slightly different base library name based on which dual library(s) are chosen.

For a particular non-header only library different variants of the library, depending on the CXXD dual library possibilities, are built. Then depending on the end-user's compiler settings when he uses the library the particular correct variant of the library is 'linked' to his code as a shared library or as a static library.

The solution in general

As a simple generalized example without immediately going into all the details, which I will do shortly, let's suppose that a shared or static library, whose current base name is called MyLib, is changed to use the CXXD regex interface in a built portion of the library. The process would be that when MyLib is built it will consist of two variants; one variant when MyLib is built when using the Boost regex library and one variant when MyLib is built when using the C++ standard regex library. In order for this to work I need two separate 'base' names for MyLib, depending on whether Boost regex or C++ standard regex is the dual library for regex when the library is built. The choice of these 'base' names will be encoded into the build process for MyLib. When an end-user uses my library, by including its appropriate header file(s), he links to the correct name depending on whether or not CXXD chooses the Boost regex or C++ standard regex library as the dual library when the regex CXXD-mod header file gets included.

The solution in detail

Now let's look in detail, using a simple example, how this would be implemented in MyLib's code.

First we have MyLib's functionality which uses the CXXD regex interface. Since MyLib is a non-header only library our functionality will consist of a header file, which the end-user will include, and a source file, which contains the code which will be compiled and linked in order to create MyLib's shared or static library.

When we used a header-only library our example was:

Using macros	**Using aliases**

`// Header file MyHeader.hpp`
`#include <boost/cxx_dual/regex.hpp>`
`#include CXXD_REGEX_HEADER`
`class MyClass`
`    {`
`    public:`
`    void MyFunc↵`
`tion(CXXD_REGEX_NS::regex &) { /* Some func↵`
`tionality */ }`
`    };`
	`// Header file MyHeader.hpp`
	`#include <boost/cxx_dual/impl/regex.hpp>`
	`class MyClass`
	`    {`
	`    public:`
	`    void MyFunc↵`
	`tion(cxxd_regex_ns::regex &) { /* Some func↵`
	`tionality */ }`
	`    };`

For our non-header only library as the header file for our example, we specify instead:

Using macros	**Using aliases**

`// Header file MyHeader.hpp`
`#include <boost/cxx_dual/regex.hpp>`
`#include CXXD_REGEX_HEADER`
`class MyClass`
`    {`
`    public:`
`    void MyFunction(CXXD_REGEX_NS::regex &);`
`    };`
	`// Header file MyHeader.hpp`
	`#include <boost/cxx_dual/impl/regex.hpp>`
	`class MyClass`
	`    {`
	`    public:`
	`    void MyFunction(cxxd_regex_ns::regex &);`
	`    };`

This is very little different from what we presented before with a header-only library. Our difference is that for our non-header only library we are going to compile the implementation of MyClass into a shared or static library. So we now have both our header file and a separate source code file MyHeader.cpp:

Using macros	**Using aliases**

`// Source file MyHeader.cpp`
`#include "MyHeader.hpp"`
`void MyClass::MyFunc↵`
`tion(CXXD_REGEX_NS::regex & rx)`
`    {`
`    /* Some Functionality using 'rx' */`
`    }`
	`// Source file MyHeader.cpp`
	`#include "MyHeader.hpp"`
	`void MyClass::MyFunc↵`
	`tion(cxxd_regex_ns::regex & rx)`
	`    {`
	`    /* Some Functionality using 'rx' */`
	`    }`

At this point we have two problems to solve:

- We want to be able to build MyLib as two variants with different base names, one for compiling when the CXXD regex mod uses the Boost regex library and one for compiling when the CXXD regex mod uses the C++ standard regex library.
- We want the end-user of MyLib to link to the correct MyLib variant depending on whether the end-user's compilation will choose the Boost regex library or the C++ standard regex library.

Building the library

The first of our two problems is the easiest to solve. As long as we have a system for building our library where we can specify the name of the library we want along with specifying macros we define for our build on the command line, we always have a solution for our first problem.

In specific terms we are going to specify two builds for our library under two different names, and we are going to specify appropriate CXXD override macros on the command line for each build. For the purposes of our example I will choose the library name 'MyLib' for the build where the Boost regex library will be used and the library name 'MyLib_std' for the build where the C++ standard version of the library will be used. When we build 'MyLib' we will pass the regex override macro CXXD_REGEX_USE_BOOST on the command line of the build process. When we build 'MyLib_std' we will pass the regex override macro CXXD_REGEX_USE_STD on the command line of the build process. In Boost bjam terms this would mean, at minimum:

```
lib MyLib : MyHeader.cpp : <define>CXXD_REGEX_USE_BOOST ;
lib MyLib_std : MyHeader.cpp : <define>CXXD_REGEX_USE_STD ;
```

Other build systems would have their own syntax for building a library.

This is an instance where we use CXXD override macros which we have not seen before, but it is perfectly valid since we want to force a particular dual library choice when we build our library. The CXXD override macros for the CXXD-mod regex are specified in the build and not in a header file. This allows us to force a particular dual library when we build the library but otherwise leave the choice of the dual library up to the CXXD mechanism for choosing the library when MyLib is being used.

When the end-user builds the MyLib library he will pass appropriate command line switches which determine whether or not the C++ standard regex library is supported. If the C++ standard regex library is not supported by appropriate command line switches the build of 'MyLib' should still succeed but the build of 'MyLib_std' will fail. This is as it should be. A build of MyLib with the appropriate command line switches (usually C++11 support) will succeed in also building 'MyLib_std'.

Linking to the library

The end-user of our built library will include our MyHeader.hpp and whether the Boost regex library is chosen or the C++ standard regex library is chosen will be determined at compile time when the end-user compiles his code. Our problem is that when the end-user includes MyHeader.hpp in a compilation we would like to be able to specify at that time, based on which dual regex library is being used, the correctly named library to which to link.

The solution to our problem lies in auto-linking for those compilers which support it. Without auto-linking what we can do is provide some sort of documentation specifying the name of our library depending on the dual library choice which occurs when the end-user uses our library.

In either case, whether the compiler supports auto-linking or not, what we want to do in MyLib is to create a header file which will encode the correct name of our library as a preprocessor macro depending on our dual library choice. This can be easily done in our example because each dual library has a macro which will tell us at compile time which choice has been made. For our example with the regex library the macro is CXXD_HAS_STD_REGEX, which is 1 if the C++ standard regex library is being used and 0 if the Boost regex library is being used.

Furthermore, for those compilers which support auto-linking, we want to provide whatever compiler mechanism is available to auto-link to our correct library name. For my example I will use the auto-linking mechanism that is built into Boost.Config. We only want this auto-link header file's functionality to be used when we are not building MyLib, ie. when MyLib is being used by another library or executable.

In our example we will add a header file to our MyLib library and call it MyLibName.hpp:

```
// Header file MyLibName.hpp
#include <boost/cxx_dual/regex.hpp>
#if CXXD_HAS_STD_REGEX
    #define MYLIB_LIBRARY_NAME MyLib_std
#else
    #define MYLIB_LIBRARY_NAME MyLib
#endif
```

We will also add a header file to our MyLib library and call it MyLibLink.hpp:

```
// Header file MyLibLink.hpp
#include <boost/config.hpp>
#if !defined(BOOST_ALL_NO_LIB) && !defined(BOOST_MY_LIBRARY_NO_LIB) && !defined(MYLIB_BEING_BUILT)
#include "MyLibName.hpp"
#define BOOST_LIB_NAME MYLIB_LIBRARY_NAME
#if defined MYLIB_DYN_LINK
    #define BOOST_DYN_LINK
#endif
#include <boost/config/auto_link.hpp>
#endif
```

We alter the header file which the end-user sees, MyHeader.hpp, to include our auto-linking code in MyLibLink.hpp:

Using macros	**Using aliases**

`// Header file MyHeader.hpp`
`#include <boost/cxx_dual/regex.hpp>`
`#include CXXD_REGEX_HEADER`
`#include "MyLibLink.hpp"`
`class MyClass`
`    {`
`    public:`
`    void MyFunction(CXXD_REGEX_NS::regex &);`
`    };`
	`// Header file MyHeader.hpp`
	`#include <boost/cxx_dual/impl/regex.hpp>`
	`#include "MyLibLink.hpp"`
	`class MyClass`
	`    {`
	`    public:`
	`    void MyFunction(cxxd_regex_ns::regex &);`
	`    };`

We want to change our code which builds the MyLib variants to define the MYLIB_BEING_BUILT macro:

```
lib MyLib : MyHeader.cpp : <define>CXXD_REGEX_USE_BOOST <define>MYLIB_BEING_BUILT ;
lib MyLib_std : MyHeader.cpp : <define>CXXD_REGEX_USE_STD <define>MYLIB_BEING_BUILT ;
```

As part of the auto-linking mechanism of Boost.Config the end-user of our library could also define the MYLIB_DYN_LINK, before including MyHeader.hpp in his code, to auto-link to the shared library version instead of the static library version of the appropriate MyLib variant.

For the end-user which is using a compiler that does not support auto-linking we should make sure to document the built variant names of our library depending on whether the end-user compilation chooses the Boost regex library or the C++ standard regex library. That end-user will have to manually link the correct library name in his own code. This is usually done by adding the name of the library to the linker command line.

As an added tool for that end-user whose compiler does not support auto-linking the MyLib build could also create a program which prints to standard output the name of the library to link using the library name generated in our MyLibName.hpp header file. We could do that with a source file called MyLibName.cpp:

```
// Source file MyLibName.cpp
#include <iostream>
#include "MyLibName.hpp"
#define MYLIB_LIBRARY_NAME_AS_STRING #MYLIB_LIBRARY_NAME
int main()
    {
    std::cout << "The MyLib library name is: '" << MYLIB_LIBRARY_NAME_AS_STRING << "'.";
    return 0;
    }
```

In our example this could be added to the Boost jam file that builds the library variants as:

```
exe MyLibName : MyLibName.cpp ;
```

When the end-user invokes our build jamfile the MyLibName program is built with whatever command line options the end-user uses. Depending on those command line options the MyLibName executable will print the appropriate name of the MyLib library for those options. For an end-user without auto-linking he could then use that name to successful link MyLib to his own library or executable.

In Boost the library name as specified by the library author is often not the actual final generated name of the library, since Boost can decorate the library author's name for his library with other information. Using Boost auto-linking facilities this discrepancy between what the library author passes to the Boost.Config auto-linking mechanism and the actual generated final name of the library is automatically handled. For compilers that do not support auto-linking it is up to the end-user to understand what the generated final name of the library will be as compared to the name given by the library author. The MyLibName program will generate the correct name given by the library author, so it is then up to the user of the library to understand the generated final name. This is no different from what happens when CXXD is not being used.

Working with possible library variants

I have chosen a very simple example in using CXXD in the built portion of a non-header only library. This is because my example uses a single CXXD-mod in the built code, and the built code consists of a single source file with its corresponding header file. It is quite probable, however, that a non-header only library uses many more CXXD-mods in the built portion of the library and that the use of CXXD-mods occurs in many more source files with their corresponding header files. While the exact same principles apply when using a number of CXXD-mods as using a single one, an important issue that comes up when using a number of CXXD-mods is whether we have to support all possible library variants and how we would limit the possible variants if necessary.

Let's consider that if the number of separate CXXD-mods in the built portion of a non-header only library is designated as 'n', we have 2^n of different CXXD combinations which the end-user of our library could use. As an example, for just 4 different CXXD-mods being included, we now have 16 different variants of our library to support and build. Furthermore, when counting the CXXD-mods being included we must take into account the number of CXXD-mods being used in the built portion of a non-header only library as not only being those CXXD-mods our library is directly using but also possibly CXXD-mods being used by other libraries which our own non-header only library is using in its built portion. Clearly the number of CXXD-mods in the built portion of a non-header only library may proliferate to an unmanageable amount if we have to build every possible variant.

Despite the fact that there are 2^n possible variants it is quite possible that the library developer will choose to support many less variants. In fact very often he may choose to support only two variants. Those two variants are often divided by whether or not the library is being compiled in C++11 on up mode or not. For most, if not all, of the CXXD-mods in a given compiler implementation, when compiling in C++11 on up mode the C++ standard library version of the dual library is available and when not compiling in C++11 mode on up only the Boost version will be available. So realistically, even when 'n' different CXXD-mods are being used by the built portion of a non-header only library, the library developer may choose to support only two variants, either all Boost libraries or all C++ standard library equivalents.

Whatever the choice of the number of variants supported in a non-header only library compared to the maximum number of variants possible based on the CXXD-mods used by that library in its built portion, we need some way to name each of the variants supported and we may need some way to intelligently limit the number of variants supported.

Let's take a look at our example MyLib and let's see what we have to do to support more library variants and possibly limit the number of variants. Imagine that in the source of the library being built we are supporting more than simply the regex CXXD-mod but also other CXXD-mods. For the sake of this updated example let's suppose we also are using the CXXD array, CXXD function, and CXXD tuple mods in the built portion of MyLib. We will do this by adding some functionality to our MyHeader interface. In actuality the built portion of a non-header only library will usually encompass far more than a single source/header file pairing, but for the sake of our still fairly simple example we will continue to imagine a single source/header pairing.

Our updated Myheader.hpp will now look like:

Using macros

```
// Header file MyHeader.hpp
#include <boost/cxx_dual/array.hpp>
#include <boost/cxx_dual/function.hpp>
#include <boost/cxx_dual/regex.hpp>
#include <boost/cxx_dual/tuple.hpp>
#include CXXD_ARRAY_HEADER
#include CXXD_FUNCTION_HEADER
#include CXXD_REGEX_HEADER
#include CXXD_TUPLE_HEADER
#include "MyLibLink.hpp"
class MyClass
    {
    public:
    void MyFunction(CXXD_REGEX_NS::regex &);
    void AFunction(CXXD_ARRAY_NS::ar↲
ray<int,5> &);
    void AnotherFunction(CXXD_FUNC↲
TION_NS::function<long (double)> &);
    void YetAnotherFunc↲
tion(CXXD_TUPLE_NS::tuple<float,short> &);
    };
```

Using aliases

```
// Header file MyHeader.hpp
#include <boost/cxx_dual/impl/array.hpp>
#include <boost/cxx_dual/impl/function.hpp>
#include <boost/cxx_dual/impl/regex.hpp>
#include <boost/cxx_dual/impl/tuple.hpp>
#include "MyLibLink.hpp"
class MyClass
    {
    public:
    void MyFunction(cxxd_regex_ns::regex &);
    void AFunction(cxxd_array_ns::ar↲
ray<int,5> &);
    void AnotherFunction(cxxd_func↲
tion_ns::function<long (double)> &);
    void YetAnotherFunc↲
tion(cxxd_tuple_ns::tuple<float,short> &);
    };
```

Our updated MyHeader.cpp will now look like:

Using macros

```
// Source file MyHeader.cpp
#include "MyHeader.hpp"
void MyClass::MyFunc↲
tion(CXXD_REGEX_NS::regex & rx)
    {
    /* Some Functionality using 'rx' */
    }
void MyClass::AFunction(CXXD_ARRAY_NS::ar↲
ray<int,5> & arr)
    {
    /* Some Functionality using 'arr' */
    }
void MyClass::AnotherFunction(CXXD_FUNC↲
TION_NS::function<long (double)> & fun)
    {
    /* Some Functionality using 'fun' */
    }
void MyClass::YetAnotherFunc↲
tion(CXXD_TUPLE_NS::tuple<float,short> &tup)
    {
    /* Some Functionality using 'tup' */
    }
```

Using aliases

```
// Source file MyHeader.cpp
#include "MyHeader.hpp"
void MyClass::MyFunc↲
tion(cxxd_regex_ns::regex & rx)
    {
    /* Some Functionality using 'rx' */
    }
void MyClass::AFunction(cxxd_array_ns::ar↲
ray<int,5> & arr)
    {
    /* Some Functionality using 'arr' */
    }
void MyClass::AnotherFunction(cxxd_func↲
tion_ns::function<long (double)> & fun)
    {
    /* Some Functionality using 'fun' */
    }
void MyClass::YetAnotherFunc↲
tion(cxxd_tuple_ns::tuple<float,short> &tup)
    {
    /* Some Functionality using 'tup' */
    }
```

If we look back at our MyLibName.hpp we can both extend the name of the library to encompass more variants as well as limit the variants we wish to allow. Allowing all our new variants could give us something like:

```
// Header file MyLibName.hpp
#include <boost/cxx_dual/array.hpp>
#include <boost/cxx_dual/function.hpp>
#include <boost/cxx_dual/regex.hpp>
#include <boost/cxx_dual/tuple.hpp>
#if CXXD_HAS_STD_ARRAY
    #if CXXD_HAS_STD_FUNCTION
        #if CXXD_HAS_STD_REGEX
            #if CXXD_HAS_STD_TUPLE
                #define MYLIB_LIBRARY_NAME MyLib_std
            #else
                #define MYLIB_LIBRARY_NAME MyLib_ar_fn_rx
            #endif
        #elif CXXD_HAS_STD_TUPLE
            #define MYLIB_LIBRARY_NAME MyLib_ar_fn_tp
        #else
            #define MYLIB_LIBRARY_NAME MyLib_ar_fn
        #endif
    #elif CXXD_HAS_STD_REGEX
        #if CXXD_HAS_STD_TUPLE
            #define MYLIB_LIBRARY_NAME MyLib_ar_rx_tp
        #else
            #define MYLIB_LIBRARY_NAME MyLib_ar_rx
        #endif
    #elif CXXD_HAS_STD_TUPLE
        #define MYLIB_LIBRARY_NAME MyLib_ar_tp
    #else
        #define MYLIB_LIBRARY_NAME MyLib_ar
    #endif
#elif CXXD_HAS_STD_FUNCTION
    #if CXXD_HAS_STD_REGEX
        #if CXXD_HAS_STD_TUPLE
            #define MYLIB_LIBRARY_NAME MyLib_fn_rx_tp
        #else
            #define MYLIB_LIBRARY_NAME MyLib_fn_rx
        #endif
    #elif CXXD_HAS_STD_TUPLE
        #define MYLIB_LIBRARY_NAME MyLib_fn_tp
    #else
        #define MYLIB_LIBRARY_NAME MyLib_fn
    #endif
#elif CXXD_HAS_STD_REGEX
    #if CXXD_HAS_STD_TUPLE
        #define MYLIB_LIBRARY_NAME MyLib_rx_tp
    #else
        #define MYLIB_LIBRARY_NAME MyLib_rx
    #endif
#elif CXXD_HAS_STD_TUPLE
    #define MYLIB_LIBRARY_NAME MyLib_tp
#else
    #define MYLIB_LIBRARY_NAME MyLib
#endif
```

That is some complicated-looking code to just give a name to each variant, but it is really just a series of tests of all 16 combinations of CXXD_HAS_STD_'XXX' for the possible 'XXX' CXXD-mod being used. It is actually more tedious than really complicated. We just have to choose a naming convention for our variants so that all variant names are unique. In our case the manual naming convention chosen is that each CXXD-mod where the C++ standard library is being chosen has a two character abbreviation preceded by an underscore appended to the base 'MyLib' name, but if all the CXXD mods have the C++ standard library being chosen the mnemonic '_std' is appended to the base 'MyLib' name instead. Furthermore the appends are done in the alphabetical order of CXXD mods being used. This manual scheme is purely arbitrary for naming variants when building a non-header only library and, of course, the library developer using CXXD may choose whatever manual scheme he wishes to give each variant he chooses to support a unique name.

Later in this documentation section I will discuss CXXD support in the form of a CXXD macro which which the library developer can use to automate the naming scheme if he wishes with a single invocation of the macro.

Now suppose we do not want to support all variants in MyLib but just a subset of the 16 possible variants in our example. As one case suppose we want to support only those variants where both CXXD array and CXXD function are both using either the Boost library or both are using the C++ standard library. Our changed MyLibName.hpp could then be:

```
// Header file MyLibName.hpp
#include <boost/cxx_dual/array.hpp>
#include <boost/cxx_dual/function.hpp>
#include <boost/cxx_dual/regex.hpp>
#include <boost/cxx_dual/tuple.hpp>
#if CXXD_HAS_STD_ARRAY
    #if CXXD_HAS_STD_FUNCTION
        #if CXXD_HAS_STD_REGEX
            #if CXXD_HAS_STD_TUPLE
                #define MYLIB_LIBRARY_NAME MyLib_std
            #else
                #define MYLIB_LIBRARY_NAME MyLib_ar_fn_rx
            #endif
        #elif CXXD_HAS_STD_TUPLE
            #define MYLIB_LIBRARY_NAME MyLib_ar_fn_tp
        #else
            #define MYLIB_LIBRARY_NAME MyLib_ar_fn
        #endif
    #else
        #error MyLib - CXXD configuration not supported where array and function differ.
    #endif
#elif CXXD_HAS_STD_FUNCTION
    #error MyLib - CXXD configuration not supported where array and function differ.
#elif CXXD_HAS_STD_REGEX
    #if CXXD_HAS_STD_TUPLE
        #define MYLIB_LIBRARY_NAME MyLib_rx_tp
    #else
        #define MYLIB_LIBRARY_NAME MyLib_rx
    #endif
#elif CXXD_HAS_STD_TUPLE
    #define MYLIB_LIBRARY_NAME MyLib_tp
#else
    #define MYLIB_LIBRARY_NAME MyLib
#endif
```

We have simplified the file producing half the number of variants as before, and we have produced a means by which we notify the end-user at compile time of the variants we do not support. Of course we could and should also notify the end-user of MyLib in the documentation about the variants we do or do not support. But the nice thing is that we can produce a compile-time error, in the form of a preprocessor error, while still using CXXD to limit the subset of variants.

Later in this documentation section I will discuss CXXD support in the form of a CXXD macro which which the library developer can use to test for all his valid variant possibilities with a single macro invocation.

Suppose we simplify further in a manner I suggested previously where we will only support the two variants which most likely corresponds to the end-user compiling with C++11 on up support or not. Now our MyLibName.hpp gets much simpler:

```
// Header file MyLibName.hpp
#include <boost/cxx_dual/array.hpp>
#include <boost/cxx_dual/function.hpp>
#include <boost/cxx_dual/regex.hpp>
#include <boost/cxx_dual/tuple.hpp>
#if CXXD_HAS_STD_ARRAY && CXXD_HAS_STD_FUNCTION && CXXD_HAS_STD_REGEX && CXXD_HAS_STD_TUPLE
    #define MYLIB_LIBRARY_NAME MyLib_std
#elif !(CXXD_HAS_STD_ARRAY || CXXD_HAS_STD_FUNCTION || CXXD_HAS_STD_REGEX || CXXD_HAS_STD_TUPLE)
    #define MYLIB_LIBRARY_NAME MyLib
#else
    #error CXXD configuration only supported if all dual libraries are Boost or all dual librar↵
ies are C++ standard.
#endif
```

Here we have further simplified the number of variants, theoretically producing builds which either support C++11 on up or do not support C++11 on up, the latter usually meaning C++03 support. Of course this may produce problems if the end-user wants to compile in C++11 mode but the compiler the end-user uses has C++ standard support for one or more of the CXXD modules we are using but not all of those we are using. In that case the end-user would encounter the preprocessor error directive in he example above. However even in that case the end-user could turn off C++11 mode when compiling and link to the MyLib variant which supports all Boost libraries for the CXXD mods without getting an error.

In these examples the library author needs to adjust the build of his library to only build the variants which he allows. In Boost terms this means modifying the jam file which builds his library.

With our first extended example, for all 16 variants, the updated jam file would be, at minimum:

```
lib MyLib : MyHeader.cpp : <define>CXXD_ARRAY_USE_BOOST <define>CXXD_FUNC↵
TION_USE_BOOST <define>CXXD_REGEX_USE_BOOST <define>CXXD_TUPLE_USE_BOOST <define>MYLIB_BE↵
ING_BUILT ;
lib MyLib_std : MyHeader.cpp : <define>CXXD_ARRAY_USE_STD <define>CXXD_FUNC↵
TION_USE_STD <define>CXXD_REGEX_USE_STD <define>CXXD_TUPLE_USE_STD <define>MYLIB_BEING_BUILT ;
lib MyLib_ar_fn_rx : MyHeader.cpp : <define>CXXD_ARRAY_USE_STD <define>CXXD_FUNC↵
TION_USE_STD <define>CXXD_REGEX_USE_STD <define>CXXD_TUPLE_USE_BOOST <define>MYLIB_BEING_BUILT ;
lib MyLib_ar_fn_tp : MyHeader.cpp : <define>CXXD_ARRAY_USE_STD <define>CXXD_FUNC↵
TION_USE_STD <define>CXXD_REGEX_USE_BOOST <define>CXXD_TUPLE_USE_STD <define>MYLIB_BEING_BUILT ;
lib MyLib_ar_fn : MyHeader.cpp : <define>CXXD_ARRAY_USE_STD <define>CXXD_FUNC↵
TION_USE_STD <define>CXXD_REGEX_USE_BOOST <define>CXXD_TUPLE_USE_BOOST <define>MYLIB_BEING_BUILT ;
lib MyLib_ar_rx_tp : MyHeader.cpp : <define>CXXD_ARRAY_USE_STD <define>CXXD_FUNC↵
TION_USE_BOOST <define>CXXD_REGEX_USE_STD <define>CXXD_TUPLE_USE_STD <define>MYLIB_BEING_BUILT ;
lib MyLib_ar_rx : MyHeader.cpp : <define>CXXD_ARRAY_USE_STD <define>CXXD_FUNC↵
TION_USE_BOOST <define>CXXD_REGEX_USE_STD <define>CXXD_TUPLE_USE_BOOST <define>MYLIB_BEING_BUILT ;
lib MyLib_ar_tp : MyHeader.cpp : <define>CXXD_ARRAY_USE_STD <define>CXXD_FUNC↵
TION_USE_BOOST <define>CXXD_REGEX_USE_BOOST <define>CXXD_TUPLE_USE_STD <define>MYLIB_BEING_BUILT ;
lib MyLib_ar : MyHeader.cpp : <define>CXXD_ARRAY_USE_STD <define>CXXD_FUNC↵
TION_USE_BOOST <define>CXXD_REGEX_USE_BOOST <define>CXXD_TUPLE_USE_BOOST <define>MYLIB_BE↵
ING_BUILT ;
lib MyLib_fn_rx_tp : MyHeader.cpp : <define>CXXD_ARRAY_USE_BOOST <define>CXXD_FUNC↵
TION_USE_STD <define>CXXD_REGEX_USE_STD <define>CXXD_TUPLE_USE_STD <define>MYLIB_BEING_BUILT ;
lib MyLib_fn_rx : MyHeader.cpp : <define>CXXD_ARRAY_USE_BOOST <define>CXXD_FUNC↵
TION_USE_STD <define>CXXD_REGEX_USE_STD <define>CXXD_TUPLE_USE_BOOST <define>MYLIB_BEING_BUILT ;
lib MyLib_fn_tp : MyHeader.cpp : <define>CXXD_ARRAY_USE_BOOST <define>CXXD_FUNC↵
TION_USE_STD <define>CXXD_REGEX_USE_BOOST <define>CXXD_TUPLE_USE_STD <define>MYLIB_BEING_BUILT ;
lib MyLib_fn : MyHeader.cpp : <define>CXXD_ARRAY_USE_BOOST <define>CXXD_FUNC↵
TION_USE_STD <define>CXXD_REGEX_USE_BOOST <define>CXXD_TUPLE_USE_BOOST <define>MYLIB_BEING_BUILT ;
lib MyLib_rx_tp : MyHeader.cpp : <define>CXXD_ARRAY_USE_BOOST <define>CXXD_FUNC↵
TION_USE_BOOST <define>CXXD_REGEX_USE_STD <define>CXXD_TUPLE_USE_STD <define>MYLIB_BEING_BUILT ;
lib MyLib_rx : MyHeader.cpp : <define>CXXD_ARRAY_USE_BOOST <define>CXXD_FUNC↵
TION_USE_BOOST <define>CXXD_REGEX_USE_STD <define>CXXD_TUPLE_USE_BOOST <define>MYLIB_BEING_BUILT ;
lib MyLib_tp : MyHeader.cpp : <define>CXXD_ARRAY_USE_BOOST <define>CXXD_FUNC↵
TION_USE_BOOST <define>CXXD_REGEX_USE_BOOST <define>CXXD_TUPLE_USE_STD <define>MYLIB_BEING_BUILT ;
```

With our second extended example, with its 8 variants, the updated jam file would be, at minimum:

```
lib MyLib : MyHeader.cpp : <define>CXXD_ARRAY_USE_BOOST <define>CXXD_FUNC↵
TION_USE_BOOST <define>CXXD_REGEX_USE_BOOST <define>CXXD_TUPLE_USE_BOOST <define>MYLIB_BE↵
ING_BUILT ;
lib MyLib_std : MyHeader.cpp : <define>CXXD_ARRAY_USE_STD <define>CXXD_FUNC↵
TION_USE_STD <define>CXXD_REGEX_USE_STD <define>CXXD_TUPLE_USE_STD <define>MYLIB_BEING_BUILT ;
lib MyLib_ar_fn_rx : MyHeader.cpp : <define>CXXD_ARRAY_USE_STD <define>CXXD_FUNC↵
TION_USE_STD <define>CXXD_REGEX_USE_STD <define>CXXD_TUPLE_USE_BOOST <define>MYLIB_BEING_BUILT ;
lib MyLib_ar_fn_tp : MyHeader.cpp : <define>CXXD_ARRAY_USE_STD <define>CXXD_FUNC↵
TION_USE_STD <define>CXXD_REGEX_USE_BOOST <define>CXXD_TUPLE_USE_STD <define>MYLIB_BEING_BUILT ;
lib MyLib_ar_fn : MyHeader.cpp : <define>CXXD_ARRAY_USE_STD <define>CXXD_FUNC↵
TION_USE_STD <define>CXXD_REGEX_USE_BOOST <define>CXXD_TUPLE_USE_BOOST <define>MYLIB_BEING_BUILT ;
lib MyLib_rx_tp : MyHeader.cpp : <define>CXXD_ARRAY_USE_BOOST <define>CXXD_FUNC↵
TION_USE_BOOST <define>CXXD_REGEX_USE_STD <define>CXXD_TUPLE_USE_STD <define>MYLIB_BEING_BUILT ;
lib MyLib_rx : MyHeader.cpp : <define>CXXD_ARRAY_USE_BOOST <define>CXXD_FUNC↵
TION_USE_BOOST <define>CXXD_REGEX_USE_STD <define>CXXD_TUPLE_USE_BOOST <define>MYLIB_BEING_BUILT ;
lib MyLib_tp : MyHeader.cpp : <define>CXXD_ARRAY_USE_BOOST <define>CXXD_FUNC↵
TION_USE_BOOST <define>CXXD_REGEX_USE_BOOST <define>CXXD_TUPLE_USE_STD <define>MYLIB_BEING_BUILT ;
```

With our third extended example, with its 2 variants, the updated jam file would be, at minimum:

```
lib MyLib : MyHeader.cpp : <define>CXXD_ARRAY_USE_BOOST <define>CXXD_FUNC↵
TION_USE_BOOST <define>CXXD_REGEX_USE_BOOST <define>CXXD_TUPLE_USE_BOOST <define>MYLIB_BE↵
ING_BUILT ;
lib MyLib_std : MyHeader.cpp : <define>CXXD_ARRAY_USE_STD <define>CXXD_FUNC↵
TION_USE_STD <define>CXXD_REGEX_USE_STD <define>CXXD_TUPLE_USE_STD <define>MYLIB_BEING_BUILT ;
```

Aside from the functional changes in MyLib itself, to use more CXXD mods in the built portion of the library, the changes to a proposed MyLibName.hpp in our example and the changes to the jam file used to build the non-header only library are the only ones that need to be done to support more variants, and possibly limit the variants compared to the total number possible. The rest of the infrastructure in our example remains in place.

Support for naming library variants and testing all valid possibilities

CXXD offers support, based on the CXXD-mods which are being included, in the form of macros for naming a library variant and for testing all valid possibilities. These macros can be used to provide a more automatic methodology for accomplishing the goals which can be found in the manual methods employed in the MyLibName.hpp sample headers specified previously. It is completely up to the end-user of CXXD whether he finds it easier to use the macros which I will describe or whether he finds it easier to use the manual methods previously explained and illustrated.

Both macros I will be discussing use the mod-IDs which have been previously explained for identifying the various CXXD-mods.

In additional a further identifier of 'CXXD_MODS_ALL' is used in each macro. The CXXD_MODS_ALL identifier refers to all CXXD-mods in slightly different ways in the two macros which follow.

Macro naming a library variant

There is a macro in CXXD for naming a library variant based on the dual library chosen for the CXXD-mods being included when the macro is actually invoked. This macro provides an automatic methodology for naming which can replace a manual method the end-user may have chosen to use. The macro is a variadic macro whose name and signature is:

```
CXXD_LIBRARY_NAME(...)
```

The macro is in the header file <boost/cxx_dual/library_name.hpp>

The macro takes one or more variadic parameters. The single required variadic parameter is the base name of the library. The remaining variadic parameters are optional and are Boost PP tuples.

Each Boost PP tuple has one to three elements. The first element, which is mandatory, is a mod-ID or the CXXD_MODS_ALL identifier. The second optional element, which may be empty, is a preprocessor identifier to be appended to the base name if the particular CXXD-mod is being included and the C++ standard library has been chosen as the dual library. The third optional element, which may also be empty, is a preprocessor identifier to be added to the base name if the particular CXXD-mod is being included and the Boost library has been chosen as the dual library.

If the first element CXXD-mod identifier of the Boost PP tuple is not recognized, the entire tuple is ignored.

The optional parameters allow the macro invoker to specify his own identifiers to be appended to the base library name for each CXXD-mod being included. If he does not specify his own identifier for a particular CXXD-mod by using an optional variadic parameter, defaults values are chosen instead.

Given the required base name for a library and the optional remaining variadic parameters the macro expands to a library variant name.

The default values to be appended to the base name for each CXXD-mod, depending on whether that CXXD-mod uses the C++ standard library or the Boost library, are given in the following table:

Table 4. Library name defaults

CXXD-mod ID	C++ standard
CXXD_ARRAY	_ar
CXXD_ATOMIC	_at
CXXD_BIND	_bd
CXXD_CHRONO	_ch
CXXD_CONDITION_VARIABLE	_cv
CXXD_ENABLE_SHARED_FROM_THIS	_es
CXXD_FUNCTION	_fn
CXXD_HASH	_ha
CXXD_MAKE_SHARED	_ms
CXXD_MEM_FN	_mf
CXXD_MOVE	_mv
CXXD_MUTEX	_mx
CXXD_RANDOM	_rd
CXXD_RATIO	_ra
CXXD_REF	_rf
CXXD_REGEX	_rx
CXXD_SHARED_MUTEX	_sm
CXXD_SHARED_PTR	_sp
CXXD_SYSTEM_ERROR	_se
CXXD_THREAD	_th
CXXD_TUPLE	_tu
CXXD_TYPE_INDEX	_ti
CXXD_TYPE_TRAITS	_tt
CXXD_UNORDERED_MAP	_um
CXXD_UNORDERED_MULTIMAP	_up
CXXD_UNORDERED_MULTISET	_ut
CXXD_UNORDERED_SET	_us

CXXD-mod ID	C++ standard
CXXD_WEAK_PTR	_wp
CXXD_MODS_ALL	_std

As can be seen in the default table above:

- Preprocessor identifiers are appended to the base name only when the C++ standard library has been chosen as the dual library for a particular CXXD-mod. Nothing is appended to the base name when the Boost library has been chosen for a particular CXXD-mod.
- The particular preprocessor identifiers added are two-character mnemonics preceded by an underscore. This was done to provide the smallest unique identifier while more clearly separating each mnemonic in the final name.

The 'CXXD_MODS_ALL' identifier specifies a single mnemonic, in place of all the individual mnemonics, to be added to the base name if all the CXXD-mods being used have either their C++ standard chosen as the dual library or their Boost library chosen as the dual library. In the default case above if all the CXXD-mods being used have the C++ standard library as the dual library the final library variant name is the base name of the library with '_std' appended, while if all the CXXD-mods being used have the Boost library as the dual library the final library variant name is just the base name with nothing further appended.

In the default case any appended mnemonics are added to the base name in the alphabetical order in which the individual CXXD-mods occur in the default table above and not in the order in which the CXXD-mods are included.

Note

The default appended mnemonics all begin with an underscore and lowercase letters so they should not clash with any macros of the same name. Nonetheless because there is a small chance of a macro clash if the compiler, or some other header file, defines macros with the same name as one of the default mnemonics, CXXD saves and restores possible macros definitions with the default appended mnemonics names. Saving is done automatically when the <boost/cxx_dual/library_name.hpp> is included. Restoring must be done manually by the end-user by including the header file <boost/cxx_dual/library_name_post.hpp> after the CXXD_LIBRARY_NAME macro is invoked. If you are sure that no other macro in the TU in which CXXD_LIBRARY_NAME is being used has the name of any one of the default appended mnemonics, you do not have to include <boost/cxx_dual/library_name_post.hpp> after the CXXD_LIBRARY_NAME is invoked. Defining macro names that are not all uppercase letters and/or that start with an underscore is really bad macro design, but one never knows what compiler vendors or third-party libraries are capable of doing.

The way that the CXXD_LIBRARY_NAME works, when the single required base name is passed to the macro, is exactly the manual method I have previously used when specifying unique variant names in the MyLibName.hpp samples given above. I chose to illustrate the manual method in the documentation based on the way that the CXXD_LIBRARY_NAME macro is designed to work when no optional parameters are specified and the default values are chosen. But of course an end-user, either manually specifying unique library variant names instead of using the CXXD_LIBRARY_NAME macro, or using the CXXD_LIBRARY_NAME macro and providing his own appended mnemonics as optional parameters, can create any schema he wants.

The default naming of CXXD_LIBRARY_NAME guarantees that if Boost is consistently being chosen as the dual library instead of the equivalent C++ standard library, which is what will naturally happen in the vast majority of cases if C++11 on up mode is not being used during compilation, the library name generated by CXXD_LIBRARY_NAME is the same as the base name. This latter is, I believe, what Boost library authors would expect in their non-header only libraries.

The end-user changes the default naming scheme by passing as optional parameters one or more Boost PP tuples for each CXXD-mod where he wants a different mnemonic to be appended to the base name. In this case the Boost PP tuple specifies a different mnemonic to be used for a particular CXXD-mod, depending on whether the dual library to be chosen for that CXXD-mod is the C++ standard library or the Boost library. The order of the mnemonics to be appended also changes according to the order of the CXXD-mod identifiers he specifies in the optional parameters, so that optional parameter mnemonics always precede default mnemonic choices in the final expanded name.

A number of examples will be given using the four CXXD-mods we have previously chosen in our examples for MyLibName.hpp header file and some arbitrarily chosen combinations. These arbitrarily chosen combinations are specified purely to illustrate how the macro works:

```
#include <boost/cxx_dual/array.hpp>
#include <boost/cxx_dual/function.hpp>
#include <boost/cxx_dual/regex.hpp>
#include <boost/cxx_dual/tuple.hpp>
#include <boost/cxx_dual/library_name.hpp>

/* If array uses Boost, function uses C++ standard, regex uses C++ standard, tuple uses Boost ↵
then: */

CXXD_LIBRARY_NAME(MyLib) expands to 'MyLib_fn_rx'

/* If array uses C++ standard, function uses C++ standard, regex uses C++ standard, tuple uses ↵
C++ standard then: */

CXXD_LIBRARY_NAME(MyLib) expands to 'MyLib_std'

/* If array uses Boost, function uses Boost, regex uses Boost, tuple uses Boost then: */

CXXD_LIBRARY_NAME(MyLib) expands to 'MyLib'

/* If array uses Boost, function uses C++ standard, regex uses C++ standard, tuple uses Boost ↵
then: */

CXXD_LIBRARY_NAME(MyLib,(CXXD_REGEX,RStd,_rboost)) expands to 'MyLibRStd_fn'

/* If array uses Boost, function uses C++ standard, regex uses Boost, tuple uses Boost then: */

CXXD_LIBRARY_NAME(MyLib,(CXXD_TUPLE,_cpp_tuple,_boost_tuple),(CXXD_ARRAY,,_bboost)) ex↵
pands to 'MyLib_boost_tuple_bboost_fn'

/* If array uses Boost, function uses Boost, regex uses Boost, tuple uses Boost then: */

CXXD_LIBRARY_NAME(MyLib,(CXXD_ALL,Std,Boost)) expands to 'MyLibBoost'

/* If array uses C++ standard, function uses C++ standard, regex uses C++ standard, tuple uses ↵
C++ standard then: */

CXXD_LIBRARY_NAME(MyLib,(CXXD_ALL,Std,Boost)) expands to 'MyLibStd'
```

If we look at the MyLibName.hpp as presented previously we can use the CXXD_LIBRARY_NAME to simplify the syntax of the file. In the first case, where we do not try to limit the variants, our file would be:

```
// Header file MyLibName.hpp
#include <boost/cxx_dual/array.hpp>
#include <boost/cxx_dual/function.hpp>
#include <boost/cxx_dual/regex.hpp>
#include <boost/cxx_dual/tuple.hpp>
#include <boost/cxx_dual/library_name.hpp>
#define MYLIB_LIBRARY_NAME CXXD_LIBRARY_NAME(MyLib)
```

In the second case where we do limit the variants so that CXXD-mod array and CXXD-mod function always choose the same dual library, our file would be:

```
// Header file MyLibName.hpp
#include <boost/cxx_dual/array.hpp>
#include <boost/cxx_dual/function.hpp>
#include <boost/cxx_dual/regex.hpp>
#include <boost/cxx_dual/tuple.hpp>
#include <boost/cxx_dual/library_name.hpp>
#if (CXXD_HAS_STD_ARRAY && !CXXD_HAS_STD_FUNCTION) || (!CXXD_HAS_STD_ARRAY && CXXD_HAS_STD_FUNC↵
TION)
    #error MyLib - CXXD configuration not supported where array and function differ.
#else
    #define MYLIB_LIBRARY_NAME CXXD_LIBRARY_NAME(MyLib)
#endif
```

In the third case where we limit our variants to the two choices of all Boost or all C++ standard libraries, our file would be:

```
// Header file MyLibName.hpp
#include <boost/cxx_dual/array.hpp>
#include <boost/cxx_dual/function.hpp>
#include <boost/cxx_dual/regex.hpp>
#include <boost/cxx_dual/tuple.hpp>
#include <boost/cxx_dual/library_name.hpp>
#if (CXXD_HAS_STD_ARRAY && CXXD_HAS_STD_FUNCTION && CXXD_HAS_STD_REGEX && CXXD_HAS_STD_TUPLE) || \
    !(CXXD_HAS_STD_ARRAY || CXXD_HAS_STD_FUNCTION || CXXD_HAS_STD_REGEX || CXXD_HAS_STD_TUPLE)
    #define MYLIB_LIBRARY_NAME CXXD_LIBRARY_NAME(MyLib)
#else
    #error CXXD configuration only supported if all dual libraries are Boost or all dual librar↵
ies are C++ standard.
#endif
```

In all cases the invocation of the single CXXD_LIBRARY_NAME macro simplifies our MyLibName.hpp header file. Nonetheless the choice of using the CXXD_LIBRARY_NAME macro or manually name the library variants is totally up to the library author.

Macro testing all possibilities

There is a macro in CXXD for testing whether the CXXD-mods which are included, when the macro is actually invoked, match one or more combinations of dual library possibilities. This macro provides an automatic method for testing all dual library possibilities at once and offers an alternative to the manual testing of dual library possibilities which the end-user may have chosen to use. The macro is a variadic macro whose name and signature is:

```
CXXD_VALID_VARIANTS(...)
```

The macro is in the header file <boost/cxx_dual/valid_variants.hpp>

The variadic parameters are ways of encoding the various CXXD dual library combinations which the library author considers valid for his library. If the dual libraries chosen for the CXXD-mods being included match any of combinations represented by the variadic parameters the macro expands to 1, otherwise it expands to 0.

Each variadic parameter represents a 'possibility'. Each possibility is a series of Boost PP tuples. This 'series' in the VMD library is called a VMD sequence.

Each Boost PP tuple in the series has two elements. The first Boost PP tuple element is a mod-ID or CXXD_MODS_ALL. The second Boost PP tuple element is either 0 or 1, meaning respectively that for the CXXD-mod specified the dual library is Boost (0) or the dual library is the C++ standard library (1). If a CXXD-mod identifier as the first tuple element is not recognized, or if the second tuple element is not 0 or 1, that tuple is ignored as part of a possibility but the possibility itself is still considered.

The Boost PP tuple encoding identifies a valid choice for the library being built for a single CXXD-mod. The series of Boost PP tuples combine the valid choices into a single possibility. When a particular CXXD-mod is not part of a possibility, as encoded by a CXXD-mod identifier, it simply means that if that CXXD-mod's header is included it can validly use either its Boost or C++ standard library as far as the possibility is concerned.

As an arbitrary example of a possibility:

```
(CXXD_REGEX,0)(CXXD_ARRAY,1)(CXXD_TUPLE,1)
```

as one of our variadic parameters says that a situation where the regex CXXD-mod uses the Boost library while the CXXD array and tuple mods use the C++ standard library is a valid possibility for the library.

The library author can specify as many valid possibilities as he chooses as variadic parameters but at least one possibility has to be specified to use the macro. If all possible combinations of the CXXD-mods being included are valid there is absolutely no point in using the CXXD_VALID_VARIANTS macro as we have no intention of limiting the possible variants by using the macro.

For the CXXD_MODS_ALL identifier only one tuple should define the possibility. The designation as a variadic parameter of:

```
(CXXD_MODS_ALL,0)
```

says that a valid possibility is that every CXXD-mod included uses Boost as the dual library. The designation as a variadic parameter of:

```
(CXXD_MODS_ALL,1)
```

says that a valid possibility is that every CXXD-mod included uses the C++ standard library as the dual library. If the CXXD_MODS_ALL tuple is combined with other tuples as part of a possibility the other tuples are ignored.

A number of examples will be given using the four CXXD-mods we have previously chosen in our examples for MyLibName.hpp header file and some arbitrarily chosen valid possibilities. These arbitrarily chosen valid possibilities are specified purely to illustrate how the macro works:

```
#include <boost/cxx_dual/array.hpp>
#include <boost/cxx_dual/function.hpp>
#include <boost/cxx_dual/regex.hpp>
#include <boost/cxx_dual/tuple.hpp>
#include <boost/cxx_dual/valid_variants.hpp>

/* Valid possibilities are:
1) All four CXXD-mods use the Boost library
2) CXXD-mod array uses the Boost library,
   CXXD-mod function uses the C++ standard library,
   CXXD-mod tuple uses the Boost library,
   CXXD-mod regex uses either the Boost library or the C++ standard library
*/

CXXD_VALID_VARIANTS((CXXD_MODS_ALL,0),(CXXD_ARRAY,0)(CXXD_FUNCTION,1)(CXXD_TUPLE,0))

/* Valid possibilities are:
1) All four CXXD-mods use the C++standard library
2) CXXD-mod array uses the Boost library,
   CXXD-mod function uses the Boost library
   CXXD-mod tuple uses the Boost library or the C++ standard library,
   CXXD-mod regex uses the Boost library or the C++ standard library
*/

CXXD_VALID_VARIANTS((CXXD_MODS_ALL,1),(CXXD_ARRAY,0)(CXXD_FUNCTION,0))

/* Valid possibilities are:
1) CXXD-mod array uses the Boost library,
   CXXD-mod function uses the Boost library
   CXXD-mod tuple uses the Boost library or the C++ standard library,
   CXXD-mod regex uses the Boost library or the C++ standard library
2) CXXD-mod array uses the C++ standard library,
   CXXD-mod function uses the C++ standard library
   CXXD-mod tuple uses the Boost library or the C++ standard library,
   CXXD-mod regex uses the Boost library or the C++ standard library
*/

CXXD_VALID_VARIANTS((CXXD_ARRAY,0)(CXXD_FUNCTION,0),(CXXD_ARRAY,1)(CXXD_FUNCTION,1))

/* Valid possibilities are:
1) All four CXXD-mods use the Boost library
2) All four CXXD-mods use the C++standard library
3) CXXD-mod regex uses the Boost library
   CXXD-mod tuple uses the C++ standard library
   CXXD-mod array uses the Boost library or the C++ standard library,
   CXXD-mod function uses the Boost library or the C++ standard library
*/

CXXD_VALID_VARIANTS((CXXD_MODS_ALL,0),(CXXD_MODS_ALL,1),(CXXD_REGEX,0)(CXXD_TUPLE,1))
```

Essentially the CXXD_VALID_VARIANTS present an alternate form of '#if (CXXD_HAS_STD_XXX ...)' preprocessor constructs which are used to limit the CXXD-mod possibilities which a particular library decides is valid. If we take our example MyLibName.hpp which limit the possibilities of dual libraries combinations we can re-code our second and third case using this macro. Since our first case, which does not limit the possibilities, has no use for using the CXXD_VALID_VARIANTS macro, we do not illustrate its use with the macro.

In the second case where we do limit the variants so that CXXD-mod array and CXXD-mod function always choose the same dual library, our file would now be:

```
// Header file MyLibName.hpp
#include <boost/cxx_dual/array.hpp>
#include <boost/cxx_dual/function.hpp>
#include <boost/cxx_dual/regex.hpp>
#include <boost/cxx_dual/tuple.hpp>
#include <boost/cxx_dual/valid_variants.hpp>
#if CXXD_VALID_VARIANTS((CXXD_ARRAY,0)(CXXD_FUNCTION,0),(CXXD_ARRAY,1)(CXXD_FUNCTION,1))
    #define MYLIB_LIBRARY_NAME CXXD_LIBRARY_NAME(MyLib)
#else
    #error MyLib - CXXD configuration not supported where array and function differ.
#endif
```

In the third case where we limit our variants to the two choices of all Boost or all C++ standard libraries, our file would now be:

```
// Header file MyLibName.hpp
#include <boost/cxx_dual/array.hpp>
#include <boost/cxx_dual/function.hpp>
#include <boost/cxx_dual/regex.hpp>
#include <boost/cxx_dual/tuple.hpp>
#include <boost/cxx_dual/valid_variants.hpp>
#if CXXD_VALID_VARIANTS((CXXD_MODS_ALL,0),(CXXD_MODS_ALL,1))
    #define MYLIB_LIBRARY_NAME CXXD_LIBRARY_NAME(MyLib)
#else
    #error CXXD configuration only supported if all dual libraries are Boost or all dual librar↵
ies are C++ standard.
#endif
```

In both cases we produce shorter versions of our MyLibName.hpp code.

Using CXXD_VALID_VARIANTS for a header-only library

Even though there is little practical benefit of limiting possibilities of CXXD-mod dual library choices when the situation is a header only library as opposed to the non-header only library we have been discussing, the CXXD_VALID_VARIANTS macro could be used in that situation also. As an example let us suppose we have a header-only library which uses CXXD-mod array and CXXD-mod function along with other CXXD-mods. As we have done in one of our non-header only MyLibName.hpp examples we arbitrarily decide that the array and function dual library choices should always match. We could code up a header file for the sole purpose of producing a preprocessor error if our array and function dual library choices do not match, and then include that header file in all are header-only library header files as a check to make sure they match whenever any of our header files is included. Our checking header file might be called MyHeaderOnlyLibCheck.hpp:

```
// Header file MyHeaderOnlyLibCheck.hpp
#include <boost/cxx_dual/array.hpp>
#include <boost/cxx_dual/function.hpp>
#include <boost/cxx_dual/valid_variants.hpp>
#if !CXXD_VALID_VARIANTS((CXXD_ARRAY,0)(CXXD_FUNCTION,0),(CXXD_ARRAY,1)(CXXD_FUNCTION,1))
    #error MyLib - CXXD configuration not supported where array and function differ.
#endif
```

We now include that header file in all of the header files of our header-only library and every time the end-user uses one of our header files a check is made to ensure that the dual libraries for array and function match.

Of course we could simply eschew the use of the CXXD_VALID_VARIANTS macro in this case and simply code up our MyHeaderOnlyLibCheck.hpp header file in this way:

```
// Header file MyHeaderOnlyLibCheck.hpp
#include <boost/cxx_dual/array.hpp>
#include <boost/cxx_dual/function.hpp>
#if !((CXXD_HAS_STD_ARRAY && CXXD_HAS_STD_FUNCTION) || (!CXXD_HAS_STD_ARRAY && !CXXD_HAS_STD_FUNC↵
TION))
    #error MyLib - CXXD configuration not supported where array and function differ.
#endif
```

Whether the possibility of using the CXXD_VALID_VARIANTS macro is considered for a non-header only library, its main purpose, or a header-only library, the choice of using the CXXD_VALID_VARIANTS macro or manually test the individual CXXD_HAS_STD_'XXX' macros is totally up to the library author.

Preprocessing errors

CXXD errors are preprocessor errors issue by the CXXD library using the preprocessor '#error' directive. These errors occur when the inclusion of a CXXD-mod produces inconsistencies which CXXD does not allow. These inconsistencies can be found, in a TU, either the first time CXXD includes a particular CXXD header file or subsequent times CXXD includes a particular CXXD header.

The errors which can occur involve four different situations for which CXXD issues the "#error' directive. These situations are, for any given CXXD-mod called 'xxx':

- If override macros for both Boost and the C++ standard library occur when a particular CXXD header is included, an '#error' directive is emitted of 'CXXD: Using C++ standard and using Boost are both defined for xxx'.
- If the C++ standard library has been previously chosen and is subsequently overridden by the Boost library, an '#error' directive is emitted of 'CXXD: Previous use of C++ standard xxx erroneously overridden'.
- If the Boost standard library has been previously chosen and is subsequently overridden by the C++ standard library, an '#error' directive is emitted of 'CXXD: Previous use of Boost xxx erroneously overridden'.
- If override macros for the C++ standard library occur but the C++ standard library is not available for the CXXD-mod xxx, an '#error' directive is emitted of 'CXXD: C++ standard xxx is not available'.

As previously stated in the documentation CXXD-mod header files are processed each time they are included to avoid inconsistencies.

The first error listed above could happen either the first time a particular CXXD header is included or a subsequent time a particular CXXD header is included, since override macros could occur at any time in a TU. It is not possible to eliminate this error message for the situation in which it would occur, since conflicting CXXD override macros always means that CXXD cannot choose the correct dual library for a particular CXXD-mod.

The second and third errors listed above can only happen at a subsequent time a particular CXXD-mod is included, since either message occurs when an override macro changes the dual library choice from a previous inclusion of the CXXD header. As explained previously when discussing overriding the default dual library consistency, you can eliminate the second and third errors for the situation in which it would occur by defining CXXD_NO_CONSISTENCY, although it is generally not a recommended thing to do so.

The last error message can only occur the first time a particular CXXD-mod is included, where an override macro for the C++ standard library fails because the library is not available. Like the second and third errors listed above it is possible to eliminate this last error message for the situation in which it would occurs. You can do this by defining the object-like macro CXXD_NO_CONFIG to nothing as in:

```
#define CXXD_NO_CONFIG
```

before including the particular CXXD-mod. This object-like macro allows an override macro for the C++ standard library to choose the particular CXXD-mod's C++ standard library equivalent as the dual library even when Boost.Config determines that the C++ standard library equivalent is not available when compiling the TU.

There are two reasons for allowing this object-macro to eliminate the particular preprocessor error message:

- The end-user may still want CXXD to use the equivalent C++ standard library for a particular CXX-mod even when Boost.Config says it is not available. This could be because of some obscure configuration that Boost.Config does not know about for a given OS/compiler combination. A compiler error or linker error, as opposed to a preprocessor error, might still show up when the end-user codes as if the C++ standard library for that CXXD-mod exists when it does not do so.
- The object-like macro allows tests for CXXD facilities to be run as if the C++ standard library were available even when it is not. This includes tests for the functionality of the CXXD_LIBRARY_NAME and CXXD_VALID_VARIANTS macro. In other words if no further use of the particular CXXD-mod's C++ standard library equivalent subsequently exists in the TU there is no reason to disallow it when including the particular CXXD header.

Eliminating this last of CXXD's preprocessor #error messages, and allowing a macro override to choose a particular CXXD-mod's C++ standard equivalent dual library even when Boost.Config says it does not exist, should very rarely ever be done.

Build support

The CXXD library offers support for build systems.

Sometimes during a build it is necessary to know whether or not a particular CXXD-mod uses the Boost implementation or the C++ standard library implementation for the compiler being used during a build. This could be because, depending on the dual library implementation chosen, a library needs to be linked or an include path needs to be added or a define needs to be made or for any number of other practical build-type reasons.

CXXD offers support for the Boost Build system specifically or for other build systems in general. This topic explains that support.

Boost Build

CXXD has build-time support for Boost Build so that depending on whether a particular CXXD-mod uses the Boost implementation or the C++ standard library implementation, a requirement or usage requirement can be set in a Boost Build rule. Among the useful built-in rules where this functionality can be used are the run, compile, link, exe, and lib rules; but any rule which allows the addition of requirements or usage requirements can be used.

CXXD implements this support by its own Boost Build rules, which can be used to conditionally set requirements depending on whether or not specific CXXD-mods use the Boost implementation or the C++ standard implementation.

The first thing an end-user of CXXD needs to do to use this functionality is to import the CXXD Boost build support module. This module is in a CXXD subdirectory called 'checks' in a jam file called 'cxxd'. So to import the Boost build support module the the end-user adds to his own jamfile:

```
import path_to_cxxd_library/checks/cxxd ;
```

Within the 'cxxd' jam file are three different rules the end-user can use, both taking the same Boost Build parameters. These rules are:

- cxxd.requires.boost
- cxxd.requires.std
- cxxd.requires.specify

Each rule takes, as a first parameter, a list of one or more names which identify CXXD-mods. For the first two rules, 'cxxd.requires.boost' and 'cxxd.requires.std', these names are the the CXXD-mods identifying a particular CXXD-mod. As an example 'regex' identifies the regex CXXD-mod. For the third rule above, 'cxxd.requires.specify', these names are also the CXXD-mods identifying a particular CXXD_mod, but are followed by a comma (',') and then 0 to indicate Boost or 1 to indicate the C++ standard. As examples 'regex,0' identifies the regex CXXD-mod with Boost chosen while 'tuple,1' identifies the tuple CXXD-mod with the C++ standard chosen.

Essentially the first parameter of 'cxxd.requires.boost' is a shorthand for passing each CXXD-mod followed by a comma (',') and then 0 in 'cxxd.requires.specify' while the first parameter of 'cxxd.requires.std' is a shorthand for passing each CXXD-mod followed by a comma (',') and then 1 in 'cxxd.requires.specify'.

Each rule takes, as an optional second parameter, 0 or more Boost Build requirement or usage requirements.

Each rule takes, as an optional third parameter, 0 or more Boost Build requirement or usage requirements.

A Boost Build requirement or usage requirement is a Boost Build feature.

The way that the 'cxxd.requires.boost' rule works is that if each of the CXXD-mods specified use its Boost library implementation, the Boost Build requirements or usage requirements specified by the optional second parameter is chosen; otherwise the Boost Build requirements or usage requirements specified by the optional third parameter is chosen.

The way that the 'cxxd.requires.std' rule works is that if each of the CXXD-mods specified use its C++ standard library implementation, the Boost Build requirements or usage requirements specified by the optional second parameter is chosen; otherwise the Boost Build requirements or usage requirements specified by the optional third parameter is chosen.

The way that the 'cxxd.requires.specific' rule works is that if each of the CXXD-mods specified use whichever implementation is chosen through the 0 or 1 addition, the Boost Build requirements or usage requirements specified by the optional second parameter is chosen; otherwise the Boost Build requirements or usage requirements specified by the optional third parameter is chosen. The 'cxxd.requires.specific' rule gives maximum flexibility in matching each CXXD-mod with the dual library implementation chosen.

When we use any of these rules we invoke them as:

```
cxxd.requires.boost CXXD-mod... : optional_requirements_may_be_empty : optional_require↵
ments_may_be_empty ;
cxxd.requires.std CXXD-mod... : optional_requirements_may_be_empty : optional_require↵
ments_may_be_empty ;
cxxd.requires.specific CXXD-mod,0-or-1...   : optional_requirements_may_be_empty : optional_re↵
quirements_may_be_empty ;
```

To use the results of these rules in one's own rules you surround these rule invocations in Boost Build with brackets and spaces (' [' and '] ').

As an example, taken directly from the CXXD tests for the CXXD regex implementation, we have:

```
run test_regex.cpp : : :
[ cxxd.requires.boost regex : <library>/boost/regex//boost_regex ] ;
```

This says that when we build the test regex example if the Boost regex implementation is being chosen we want to link in the Boost regex library. As you can see it isn't necessary to specify all optional requirements when invoking the rule. The exact same effect could be coded as:

```
run test_regex.cpp : : :
[ cxxd.requires.std regex : : <library>/boost/regex//boost_regex ] ;
```

or as:

```
run test_regex.cpp : : :
[ cxxd.requires.specific regex,0 : <library>/boost/regex//boost_regex ] ;
```

or even as:

```
run test_regex.cpp : : :
[ cxxd.requires.specific regex,1 : : <library>/boost/regex//boost_regex ] ;
```

We can have more than one CXXD-mod as our first parameter to any of our three rules, although this is a less common scenario. Let's make up an arbitrary case just to show how this works. Let's say that when compiling our source file 'my_source.cpp' if CXXD is choosing both the Boost bind and the Boost function implementations we want to define a macro called MY_MACRO to 1, else we define MY_MACRO to 0. Here is what our Boost Build code for this would be:

```
compile my_source.cpp :
[ cxxd.requires.boost bind function : <define>MY_MACRO=1 : <define>MY_MACRO=0 ] ;
```

We could have more than one feature as our requirements, as in:

```
compile my_source.cpp :
[ cxxd.requires.boost bind function :
  <define>MY_MACRO=1 <include>some_directory_path :
  <define>MY_MACRO=0 ] ;
```

We could have more than one invocation of cxxd.requires.boost, cxxd.requires.std, or cxxd.requires.specific in our requirements or usage requirements section of a rule, as in:

```
compile my_source.cpp :
[ cxxd.requires.boost bind function :
  <define>MY_MACRO=1 <include>some_directory_path :
  <define>MY_MACRO=0 ]
[ cxxd.requires.std regex : : <library>/boost/regex//boost_regex ]
[ cxxd.requires.specific tuple,1 : <cxxflags>-someflag ] ;
```

With the 'cxxd.requires.specific' rule we can mix Boost and C++ standard implementations to set our requirements, as in:

```
compile my_source.cpp :
[ cxxd.requires.specific array,1 hash,0 : <include>some_path : <define>some_macro=some_value ] ;
```

Here we are saying that if the array CXXD-mod uses the C++ standard implementation and the hash CXXD-mod uses the Boost implementation, add 'some_path' to the include paths, else define a macro called 'some_macro' to 'some_value'.

General build support

There are two different means by which CXXD offers general build support to end-users.

Running a program

For general build support there is a program in the CXXD build directory, which the end-user can build using Boost Build with a particular compiler, which when run will subsequently tell him whether or not particular CXXD-mods use their Boost or C++ standard implementations. The program is called 'cxxd_choice'. The resulting program is specific to the compiler used when building the program and is self-contained, not relying on any shared libraries.

When cxxd_choice is invoked it takes as arguments one or more CXXD-mod designations followed by a comma (',') and a 0 indicating a Boost implementation or a 1 indicating a C++ standard implementation. Examples for a single argument would be 'regex,0' to designate the Boost implementation for the regex CXXD-mod implementation or 'tuple,1' to designate the C++ standard implementation for the tuple CXXD-mod implementation. If the arguments all match the actual dual library implementations chosen the cxxd_choice program returns 0, otherwise it returns the number of arguments that did not match. This return value could then be subsequently used in general build systems to add whatever general build features are needed when a particular CXXD-mod dual library choice, or a combination of choices, is made.

It needs to be emphasized that the results of the dual libraries chosen by default is completely reliant on the compiler invocation used when building the cxxd_choice program. Please recall that the default algorithm used by CXXD is that if the C++ standard library implementation of a CXXD-mod is available during compilation that choice is made, otherwise the Boost library implementation for that CXXD-mod is chosen. Not only will different compilers offer different default dual library choices in their compiler implementations but different compiler flags, such as ones designating the level of C++ support (c++03, c++11, c++14), will change the dual library choices.

The cxxd_choice program can also write to standard output the results of its dual library choices. If the first command-line parameter is '-d' or '--debug', besides returning its 0 or non-zero value, the program will write to standard output the result of each dual library test made by the subsequent parameters. The results are in the exact form of:

```
Processing 'argument'.
Parameter 'argument' succeeds.
Parameter 'argument' fails.
Parameter 'argument' has an invalid format.
```

each on its own line as applicable, where argument is the actual parameter subsequently passed as a command-line parameter to the program.

The cxxd_choice program, when passed no parameters, will return 0 but will also write to standard output the choice for every CXXD-mod in the form of individual lines of:

```
CXXD-mod name = 0-or-1
```

where CXXD-mod name is the lowercase name of the CXXD-mod, such as 'regex' or 'tuple', and 0 means the Boost library being chosen while 1 means the C++ standard library being chosen. The CXXD-mod names are listed in alphabetical order.

By default when Boost Build builds the cxxd_choice program it will put its resulting exe in different directories depending on the compiler chosen when building the program. This means that each compiler will have its own copy of cxxd_choice, which is what the end-user wants, but the end-user needs to find the directory where Boost Build will put the exe.

The jamfile for building cxxd_choice in the CXXD 'build' subdirectory has a commented out line, which the end-user could uncomment, for specifying the exact location for putting the resulting exe. He could use Boost Build conditional requirements with the <location> property in the commented out line to specify where the exe should be installed for each compiler toolset he uses, as long as he gives the 'install' rule a different name for each compiler toolset being used. If he wanted to change the commented out install rule to manually specify different locations for different compilers his Boost Build code might look like:

```
install cxxd_choice_install_gcc : cxxd_choice : toolset=gcc:<location>gcc_path ;
install cxxd_choice_install_clang : cxxd_choice : toolset=clang:<location>clang_path ;
install cxxd_choice_install_msvc : cxxd_choice : toolset=msvc:<location>msvc_path ;
```

The names for the install rule are purely arbitrary, but need to be different for each toolset chosen.

Testing variants

For on-the-fly testing of dual library choices there is a source file in the 'test' directory called 'test_vv.cpp'. This test is represented by a Boost Build explicit alias called 'tvv'. This means that the test is only run when you explicitly pass the mnemonic 'tvv' to the command line when running tests for the CXXD library.

The 'test_vv.cpp' test is just a compile of a program that either succeeds as a compile or fails. In order to succeed compilation the invoker of the test must pass on the command line a macro definition for a macro called 'CXXD_VV' which is equal to a single valid variant macro parameter. A valid variant macro parameter, which has previously been explained, is a way of encoding what dual library possibility is wanted. It takes the form of a VMD sequence of two element tuples where the first element is the mod-ID and the second element is 0 for the Boost implementation or 1 for the C++ standard library implementation.

To test the compilation of test_vv.cpp the b2 command line, run in rhe CXXD test directory, would look like:

```
b2 toolset=some_compiler tvv define=CXXD_VV=some_vmd_sequence
```

A typical VMD sequence might look like:

```
(CXXD_REGEX,0)
```

or

```
(CXXD_BIND,1)(CXXD_FUNCTION,1)(CXXD_REF,1)
```

so the command line might be something like:

```
b2 toolset=gcc tvv define=CXXD_VV=(CXXD_REGEX,0)
```

or

```
b2 toolset=clang tvv define=CXXD_VV=(CXXD_BIND,1)(CXXD_FUNCTION,1)(CXXD_REF,1)
```

If the VMD sequence matches at compile time what the compiler provides in the way of the default dual library choice for the CXXD-mods in the sequence the compilation succeeds, otherwise the compilation fails.

This on-the-fly invocation of a CXXD test to match dual library choices could be used by build tools which can specify different features based on whether a program execution fails or not. It can also be used by an end-user to determine on-the-fly whether or not particular CXXD-mods match in their dual library choices some desired configuration.

Header files

As previously explained each CXXD-mod has its own header file and a general mod header 'cxx_mods.hpp' is also available for including all of the individual CXXD-mods. For a given CXXD-mod header the end-user would include, for instance:

```
#include <boost/cxx_dual/regex.hpp>
```

and for the general mod header file the end-user would include:

```
#include <boost/cxx_dual/cxx_mods.hpp>
```

There are also header files for supporting macros in 'library_name.hpp' and 'valid_variants.hpp'. These can be included separately as:

```
#include <boost/cxx_dual/library_name.hpp>
#include <boost/cxx_dual/valid_variants.hpp>
```

A separate header file also exists for the various mod_IDs used by the supporting macros:

```
#include <boost/cxx_dual/mod_ids.hpp>
```

There is also a header file which includes all the above headers in the library. This header file is 'cxx_dual.hpp'. This header file includes the individual CXXD-mod headers as well as the macro code for the two supporting header files and the mod-ID header file. Including this header file is simply:

```
#include <boost/cxx_dual/cxx_dual.hpp>
```

All of the code in these header files is preprocessor code and the macros which allow the library to work. The macros are dependent on the Boost Preprocessor library, the Boost.Config library, and for the supporting macros and their header files the Boost VMD library. All of these other Boost libraries are header only libraries.

There are implementation headers for each CXXD-mod in the 'impl' include sub-directory. These header files create dependencies on the particular CXXD-mod since they include the appropriate headers needed and create a namespace alias to the appropriate namespace, depending on which dual library has been chosen. For a given CXXD-mod implementation header the end-user would include, for instance:

```
#include <boost/cxx_dual/impl/regex.hpp>
```

None of the individual CXXD-mod implementation headers are included by any of the general header files above. An implementation header always includes its appropriate CXXD-mod header but is then subsequently only processed once in a particular TU.

Tests

The CXXD tests encompass tests for the individual variadic macros which CXXD supports to make using CXXD in a library easier, for the individual CXXD-mods, and for the test_vv.cpp compilation previously discussed. The tests are in the CXXD 'test' sub-directory so running the tests in general is a matter of being in that subdirectory and invoking 'b2' as appropriate:

```
b2 any_other_b2_parameters...
```

The tests for the individual variadic macros aiding library support, CXXD_LIBRARY_NAME and CXXD_VALID_VARIANTS, require variadic macro support but do not necessarily require C++11 on up mode in most compilers, since the major compilers support variadic macros without necessarily supporting C++11. These include clang, gcc, and VC++. However because these compilers may issue numerous warnings about variadic macro usage without being in C++11 mode, while still working correctly as far as the two variadic macros are concerned, warnings are turned off in the tests so as not to flood the end-user with spurious warnings. These tests are not dependent on any CXXD-mod and are therefore always run when the tests are run.

These tests are divided into two separate Boost Build aliases, 'ln' for the library name tests and 'vv' for the valid variant tests. So you could decide to run only the tests for either of the particular aliases by specifying them on the command line, as in:

```
b2 ln any_other_b2_parameters...
```

or

```
b2 vv any_other_b2_parameters...
```

The tests for the individual CXXD-mods do depend on other Boost libraries when the Boost dual library is chosen for the individual CXXD-mod. Therefore these tests are marked 'explicit' in terms of Boost Build, which means that they are not run normally when running the CXXD tests. All of these tests for the individual CXXD-mods are under a Boost Build alias called 'mods'. Therefore in order to just run these tests the command line should be:

```
b2 mods any_other_b2_parameters...
```

The valid variant test call test_vv.cpp under the alias 'tvv' has already been discussed. This test, like the tests for the individual CXXD-mods is also marked 'explicit', which means that it will not be ordinarily run unless it is explicitly invoked by:

```
b2 tvv any_other_b2_parameters...
```

The other b2 parameters necessary to run the valid variant test have previously been discussed.

In order to run the default tests as well as any of the explicit tests you can execute:

```
b2 . explicit_test_aliases... any_other_b2_parameters...
```

as in:

```
b2 . mods any_other_b2_parameters...
```

in order to run the default tests and the CXXD-mod explicit tests.

Reference

Header <boost/cxx_dual/array.hpp>

Dual library for the array implementation.

Chooses either the Boost array implementation or the C++ standard array implementation.

```


CXXD_HAS_STD_ARRAY
CXXD_ARRAY_NS
CXXD_ARRAY_HEADER
CXXD_ARRAY_USE_STD
CXXD_ARRAY_USE_BOOST
```

Macro CXXD_HAS_STD_ARRAY

CXXD_HAS_STD_ARRAY — Determines whether the C++ standard array implementation or the Boost array implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/array.hpp>

CXXD_HAS_STD_ARRAY
```

Description

The object-like macro expands to: 1 if the C++ standard array implementation has been chosen 0 if the Boost array implementation has been chosen.

Macro CXXD_ARRAY_NS

CXXD_ARRAY_NS — The array namespace.

Synopsis

```
// In header: <boost/cxx_dual/array.hpp>

CXXD_ARRAY_NS
```

Description

The object-like macro expands to the namespace for the array implementation.

Macro CXXD_ARRAY_HEADER

CXXD_ARRAY_HEADER — The array header file name.

Synopsis

```
// In header: <boost/cxx_dual/array.hpp>

CXXD_ARRAY_HEADER
```

Description

The object-like macro expands to the include header file designation for the array header file. The macro is used with the syntax: #include CXXD_ARRAY_HEADER

Macro CXXD_ARRAY_USE_STD

CXXD_ARRAY_USE_STD — Override macro for C++ standard array implementation.

Synopsis

```
// In header: <boost/cxx_dual/array.hpp>

CXXD_ARRAY_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard array implementation. If the C++ standard array implementation is not available a preprocessor error is generated.

Macro CXXD_ARRAY_USE_BOOST

CXXD_ARRAY_USE_BOOST — Override macro for Boost array implementation.

Synopsis

```
// In header: <boost/cxx_dual/array.hpp>

CXXD_ARRAY_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost array implementation.

Header <boost/cxx_dual/atomic.hpp>

Dual library for atomic data type.

Chooses either the Boost atomic implementation or the C++ standard atomic implementation.

```
CXXD_HAS_STD_ATOMIC
CXXD_ATOMIC_NS
CXXD_ATOMIC_HEADER
CXXD_ATOMIC_MACRO(macro)
CXXD_ATOMIC_USE_STD
CXXD_ATOMIC_USE_BOOST
```

Macro CXXD_HAS_STD_ATOMIC

CXXD_HAS_STD_ATOMIC — Determines whether the C++ standard atomic implementation or the Boost atomic implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/atomic.hpp>

CXXD_HAS_STD_ATOMIC
```

Description

The object-like macro expands to: 1 if the C++ standard atomic implementation has been chosen 0 if the Boost atomic implementation has been chosen.

Macro CXXD_ATOMIC_NS

CXXD_ATOMIC_NS — The atomic namespace.

Synopsis

```
// In header: <boost/cxx_dual/atomic.hpp>

CXXD_ATOMIC_NS
```

Description

The object-like macro expands to the namespace for the atomic implementation.

Macro CXXD_ATOMIC_HEADER

CXXD_ATOMIC_HEADER — The atomic header file name.

Synopsis

```
// In header: <boost/cxx_dual/atomic.hpp>

CXXD_ATOMIC_HEADER
```

Description

The object-like macro expands to the include header file designation for the atomic header file. The macro is used with the syntax: #include CXXD_ATOMIC_HEADER

Macro CXXD_ATOMIC_MACRO

CXXD_ATOMIC_MACRO — Generates an object-like macro name from the 'macro' name passed to it.

Synopsis

```
// In header: <boost/cxx_dual/atomic.hpp>

CXXD_ATOMIC_MACRO(macro)
```

Description

The function-like macro expands to the name of an atmomic object-like macro name for any given atomic macro name passed to it.

Macro CXXD_ATOMIC_USE_STD

CXXD_ATOMIC_USE_STD — Override macro for C++ standard atomic implementation.

Synopsis

```
// In header: <boost/cxx_dual/atomic.hpp>

CXXD_ATOMIC_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard atomic implementation. If the C++ standard atomic implementation is not available a preprocessor error is generated.

Macro CXXD_ATOMIC_USE_BOOST

CXXD_ATOMIC_USE_BOOST — Override macro for Boost atomic implementation.

Synopsis

```
// In header: <boost/cxx_dual/atomic.hpp>

CXXD_ATOMIC_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost atomic implementation.

Header <boost/cxx_dual/bind.hpp>

Dual library for the bind implementation.

Chooses either the Boost bind implementation or the C++ standard bind implementation.

```
CXXD_HAS_STD_BIND
CXXD_BIND_NS
CXXD_BIND_HEADER
CXXD_BIND_USE_STD
CXXD_BIND_USE_BOOST
```

Macro CXXD_HAS_STD_BIND

CXXD_HAS_STD_BIND — Determines whether the C++ standard bind implementation or the Boost bind implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/bind.hpp>

CXXD_HAS_STD_BIND
```

Description

The object-like macro expands to: 1 if the C++ standard bind implementation has been chosen 0 if the Boost bind implementation has been chosen.

Macro CXXD_BIND_NS

CXXD_BIND_NS — The bind namespace.

Synopsis

```
// In header: <boost/cxx_dual/bind.hpp>

CXXD_BIND_NS
```

Description

The object-like macro expands to the namespace for the bind implementation.

Macro CXXD_BIND_HEADER

CXXD_BIND_HEADER — The bind header file name.

Synopsis

```
// In header: <boost/cxx_dual/bind.hpp>

CXXD_BIND_HEADER
```

Description

The object-like macro expands to the include header file designation for the bind header file. The macro is used with the syntax: #include CXXD_BIND_HEADER

Macro CXXD_BIND_USE_STD

CXXD_BIND_USE_STD — Override macro for C++ standard bind implementation.

Synopsis

```
// In header: <boost/cxx_dual/bind.hpp>

CXXD_BIND_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard bind implementation. If the C++ standard bind implementation is not available a preprocessor error is generated.

Macro CXXD_BIND_USE_BOOST

CXXD_BIND_USE_BOOST — Override macro for Boost bind implementation.

Synopsis

```
// In header: <boost/cxx_dual/bind.hpp>

CXXD_BIND_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost bind implementation.

Header <boost/cxx_dual/chrono.hpp>

Dual library for chrono implementation.

Chooses either the Boost chrono implementation or the C++ standard chrono implementation.

```


CXXD_HAS_STD_CHRONO
CXXD_CHRONO_NS
CXXD_CHRONO_HEADER
CXXD_CHRONO_USE_STD
CXXD_CHRONO_USE_BOOST
```

Macro CXXD_HAS_STD_CHRONO

CXXD_HAS_STD_CHRONO — Determines whether the C++ standard chrono implementation or the Boost chrono implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/chrono.hpp>

CXXD_HAS_STD_CHRONO
```

Description

The object-like macro expands to: 1 if the C++ standard chrono implementation has been chosen 0 if the Boost chrono implementation has been chosen.

Macro CXXD_CHRONO_NS

CXXD_CHRONO_NS — The chrono namespace.

Synopsis

```
// In header: <boost/cxx_dual/chrono.hpp>

CXXD_CHRONO_NS
```

Description

The object-like macro expands to the namespace for the chrono implementation.

Macro CXXD_CHRONO_HEADER

CXXD_CHRONO_HEADER — The chrono header file name.

Synopsis

```
// In header: <boost/cxx_dual/chrono.hpp>

CXXD_CHRONO_HEADER
```

Description

The object-like macro expands to the include header file designation for the chrono header file. The macro is used with the syntax: #include CXXD_CHRONO_HEADER

Macro CXXD_CHRONO_USE_STD

CXXD_CHRONO_USE_STD — Override macro for C++ standard chrono implementation.

Synopsis

```
// In header: <boost/cxx_dual/chrono.hpp>

CXXD_CHRONO_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard chrono implementation. If the C++ standard chrono implementation is not available a preprocessor error is generated.

Macro CXXD_CHRONO_USE_BOOST

CXXD_CHRONO_USE_BOOST — Override macro for Boost chrono implementation.

Synopsis

```
// In header: <boost/cxx_dual/chrono.hpp>

CXXD_CHRONO_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost chrono implementation.

Header <boost/cxx_dual/condition_variable.hpp>

Dual library for the condition variable implementation.

Chooses either the Boost condition variable implementation or the C++ standard condition variable implementation.

```


CXXD_HAS_STD_CONDITION_VARIABLE
CXXD_CONDITION_VARIABLE_NS
CXXD_CONDITION_VARIABLE_HEADER
CXXD_CONDITION_VARIABLE_USE_STD
CXXD_CONDITION_VARIABLE_USE_BOOST
```

Macro CXXD_HAS_STD_CONDITION_VARIABLE

CXXD_HAS_STD_CONDITION_VARIABLE — Determines whether the C++ standard condition variable implementation or the Boost condition variable implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/condition_variable.hpp>

CXXD_HAS_STD_CONDITION_VARIABLE
```

Description

The object-like macro expands to: 1 if the C++ standard condition variable implementation has been chosen 0 if the Boost condition variable library has been chosen.

Macro CXXD_CONDITION_VARIABLE_NS

CXXD_CONDITION_VARIABLE_NS — The condition variable namespace.

Synopsis

```
// In header: <boost/cxx_dual/condition_variable.hpp>

CXXD_CONDITION_VARIABLE_NS
```

Description

The object-like macro expands to the namespace for the condition variable implementation.

Macro CXXD_CONDITION_VARIABLE_HEADER

CXXD_CONDITION_VARIABLE_HEADER — The condition variable header file name.

Synopsis

```
// In header: <boost/cxx_dual/condition_variable.hpp>

CXXD_CONDITION_VARIABLE_HEADER
```

Description

The object-like macro expands to the include header file designation for the condition variable header file. The macro is used with the syntax: #include CXXD_CONDITION_VARIABLE_HEADER

Macro CXXD_CONDITION_VARIABLE_USE_STD

CXXD_CONDITION_VARIABLE_USE_STD — Override macro for C++ standard condition_variable implementation.

Synopsis

```
// In header: <boost/cxx_dual/condition_variable.hpp>

CXXD_CONDITION_VARIABLE_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard condition variable implementation. If the C++ standard condition variable implementation is not available a preprocessor error is generated.

Macro CXXD_CONDITION_VARIABLE_USE_BOOST

CXXD_CONDITION_VARIABLE_USE_BOOST — Override macro for Boost condition_variable implementation.

Synopsis

```
// In header: <boost/cxx_dual/condition_variable.hpp>

CXXD_CONDITION_VARIABLE_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost condition variable implementation.

Header <boost/cxx_dual/cxx_mods.hpp>

Includes all dual libraries in a single header.

Header file to include all the dual libraries with a single include.

```


CXXD_NO_CONFIG
CXXD_NO_CONSISTENCY
CXXD_USE_STD
CXXD_USE_BOOST
```

Macro CXXD_NO_CONFIG

CXXD_NO_CONFIG — Macro which allows an override for the C++ standard implementation of a CXXD-mod to be successful even when it is unavailable.

Synopsis

```
// In header: <boost/cxx_dual/cxx_mods.hpp>

CXXD_NO_CONFIG
```

Description

The object-like macro, when defined to nothing prior to including any CXXD-mod header file, allows the dual choice of the C++ standard implementation through a macro override to be successful even when the C++ standard implementation for that CXXD-mod is unavailable. CXXD determines availability of the C++ standard implementation of a particular CXXD-mod through settings in Boost.Config. When an override macro is used to force the dual library choice of the C++ standard implementation for a particular CXXD-mod, and CXXD determines through Boost.Config that the C++ standard implementation is not available, a preprocessing error normally occurs. Using this macro tells CXXD ro set the dual library choice to the C++ standard implementation without producing a preprocessor error.

Macro CXXD_NO_CONSISTENCY

CXXD_NO_CONSISTENCY — Macro which turns off CXXD consistency for any CXXD-mod header file.

Synopsis

```
// In header: <boost/cxx_dual/cxx_mods.hpp>

CXXD_NO_CONSISTENCY
```

Description

The object-like macro, when defined to nothing prior to including any CXXD-mod header file, turns off CXXD consistency checking and error reporting for that CXXD-mod header file. CXXD consistency makes sure that a dual library choice for a particular CXXD-mod does not change no matter how many times that CXXD-mod is included in a TU (translation unit). When CXXD consistency is enforced changing the dual library choice in a TU causes a preprocessing error. Using this macro allows the dual library choice for a particular CXXD-mod to change at different points of a TU.

Macro CXXD_USE_STD

CXXD_USE_STD — Override macro for any C++ standard implementation.

Synopsis

```
// In header: <boost/cxx_dual/cxx_mods.hpp>

CXXD_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including any CXXD-mod header file, forces the dual library choice of the C++ standard implementation for that CXXD-mod. If the C++ standard implementation for that CXXD-mod is not available a preprocessor error is generated.

Macro CXXD_USE_BOOST

CXXD_USE_BOOST — Override macro for any Boost implementation.

Synopsis

```
// In header: <boost/cxx_dual/cxx_mods.hpp>

CXXD_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including any CXXD-mod header file, forces the dual library choice of the Boost implementation for that CXXD-mod.

Header <boost/cxx_dual/enable_shared_from_this.hpp>

Dual library for the enable_shared_from_this implementation.

Chooses either the Boost enable_shared_from_this_implementation or the C++ standard enable_shared_from_this implementation.

```


CXXD_HAS_STD_ENABLE_SHARED_FROM_THIS
CXXD_ENABLE_SHARED_FROM_THIS_NS
CXXD_ENABLE_SHARED_FROM_THIS_HEADER
CXXD_ENABLE_SHARED_FROM_THIS_USE_STD
CXXD_ENABLE_SHARED_FROM_THIS_USE_BOOST
```

Macro CXXD_HAS_STD_ENABLE_SHARED_FROM_THIS

CXXD_HAS_STD_ENABLE_SHARED_FROM_THIS — Determines whether the C++ standard enable_shared_from_this implementation or the Boost enable_shared_from_this implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/enable_shared_from_this.hpp>

CXXD_HAS_STD_ENABLE_SHARED_FROM_THIS
```

Description

The object-like macro expands to: 1 if the C++ standard enable_shared_from_this implementation has been chosen 0 if the Boost enable_shared_from_this implementation has been chosen.

Macro CXXD_ENABLE_SHARED_FROM_THIS_NS

CXXD_ENABLE_SHARED_FROM_THIS_NS — The enable_shared_from_this namespace.

Synopsis

```
// In header: <boost/cxx_dual/enable_shared_from_this.hpp>

CXXD_ENABLE_SHARED_FROM_THIS_NS
```

Description

The object-like macro expands to the namespace for the enable_shared_from_this implementation.

Macro CXXD_ENABLE_SHARED_FROM_THIS_HEADER

CXXD_ENABLE_SHARED_FROM_THIS_HEADER — The enable_shared_from_this header file name.

Synopsis

```
// In header: <boost/cxx_dual/enable_shared_from_this.hpp>

CXXD_ENABLE_SHARED_FROM_THIS_HEADER
```

Description

The object-like macro expands to the include header file designation for the enable_shared_from_this header file. The macro is used with the syntax: #include CXXD_ENABLE_SHARED_FROM_THIS_HEADER

Macro CXXD_ENABLE_SHARED_FROM_THIS_USE_STD

CXXD_ENABLE_SHARED_FROM_THIS_USE_STD — Override macro for C++ standard enable_shared_from_this implementation.

Synopsis

```
// In header: <boost/cxx_dual/enable_shared_from_this.hpp>

CXXD_ENABLE_SHARED_FROM_THIS_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard enable_shared_from_this implementation. If the C++ standard enable_shared_from_this implementation is not available a preprocessor error is generated.

Macro CXXD_ENABLE_SHARED_FROM_THIS_USE_BOOST

CXXD_ENABLE_SHARED_FROM_THIS_USE_BOOST — Override macro for Boost enable_shared_from_this implementation.

Synopsis

```
// In header: <boost/cxx_dual/enable_shared_from_this.hpp>

CXXD_ENABLE_SHARED_FROM_THIS_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost enable_shared_from_this implementation.

Header <boost/cxx_dual/function.hpp>

Dual library for the function implementation.

Chooses either the Boost function implementation or the C++ standard function implementation.

```


CXXD_HAS_STD_FUNCTION
CXXD_FUNCTION_NS
CXXD_FUNCTION_HEADER
CXXD_FUNCTION_USE_STD
CXXD_FUNCTION_USE_BOOST
```

Macro CXXD_HAS_STD_FUNCTION

CXXD_HAS_STD_FUNCTION — Determines whether the C++ standard function implementation or the Boost function implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/function.hpp>

CXXD_HAS_STD_FUNCTION
```

Description

The object-like macro expands to: 1 if the C++ standard function implementation has been chosen 0 if the Boost function implementation has been chosen.

Macro CXXD_FUNCTION_NS

CXXD_FUNCTION_NS — The function namespace.

Synopsis

```
// In header: <boost/cxx_dual/function.hpp>

CXXD_FUNCTION_NS
```

Description

The object-like macro expands to the namespace for the function implementation.

Macro CXXD_FUNCTION_HEADER

CXXD_FUNCTION_HEADER — The function header file name.

Synopsis

```
// In header: <boost/cxx_dual/function.hpp>

CXXD_FUNCTION_HEADER
```

Description

The object-like macro expands to the include header file designation for the function header file. The macro is used with the syntax: #include CXXD_FUNCTION_HEADER

Macro CXXD_FUNCTION_USE_STD

CXXD_FUNCTION_USE_STD — Override macro for C++ standard function implementation.

Synopsis

```
// In header: <boost/cxx_dual/function.hpp>

CXXD_FUNCTION_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard function implementation. If the C++ standard function implementation is not available a preprocessor error is generated.

Macro CXXD_FUNCTION_USE_BOOST

CXXD_FUNCTION_USE_BOOST — Override macro for Boost function implementation.

Synopsis

```
// In header: <boost/cxx_dual/function.hpp>

CXXD_FUNCTION_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost function implementation.

Header <boost/cxx_dual/hash.hpp>

Dual library for the hash implementation.

Chooses either the Boost hash implementation or the C++ standard hash implementation.

```


CXXD_HAS_STD_HASH
CXXD_HASH_NS
CXXD_HASH_HEADER
CXXD_HASH_USE_STD
CXXD_HASH_USE_BOOST
```

Macro CXXD_HAS_STD_HASH

CXXD_HAS_STD_HASH — Determines whether the C++ standard hash implementation or the Boost hash implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/hash.hpp>

CXXD_HAS_STD_HASH
```

Description

The object-like macro expands to: 1 if the C++ standard hash implementation has been chosen 0 if the Boost hash implementation has been chosen.

Macro CXXD_HASH_NS

CXXD_HASH_NS — The hash namespace.

Synopsis

```
// In header: <boost/cxx_dual/hash.hpp>

CXXD_HASH_NS
```

Description

The object-like macro expands to the namespace for the hash implementation.

Macro CXXD_HASH_HEADER

CXXD_HASH_HEADER — The hash header file name.

Synopsis

```
// In header: <boost/cxx_dual/hash.hpp>

CXXD_HASH_HEADER
```

Description

The object-like macro expands to the include header file designation for the hash header file. The macro is used with the syntax: #include CXXD_HASH_HEADER

Macro CXXD_HASH_USE_STD

CXXD_HASH_USE_STD — Override macro for C++ standard hash implementation.

Synopsis

```
// In header: <boost/cxx_dual/hash.hpp>

CXXD_HASH_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard hash implementation. If the C++ standard hash implementation is not available a preprocessor error is generated.

Macro CXXD_HASH_USE_BOOST

CXXD_HASH_USE_BOOST — Override macro for Boost hash implementation.

Synopsis

```
// In header: <boost/cxx_dual/hash.hpp>

CXXD_HASH_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost hash implementation.

Header <boost/cxx_dual/library_name.hpp>

Contains CXXD_LIBRARY_NAME macro.

```
CXXD_LIBRARY_NAME(...)
```

Macro CXXD_LIBRARY_NAME

CXXD_LIBRARY_NAME — Automatically names a non-header only library.

Synopsis

```
// In header: <boost/cxx_dual/library_name.hpp>

CXXD_LIBRARY_NAME(...)
```

Description

The function-like macro expands to the name of a non-header only library. It is useful for non-header only libraries in which different library variants are generated depending on the dual library choices for the CXXD-mods being used in the built portion of the library. The macro by default will expand to a unique library name depending on the dual library choices. The macro works based on the CXXD-mods included before the macro is invoked.

The required first variadic parameter is the base library name. The generated library name will be the base library name and possibly other mnemonics appended to it. The base library name should be the library name as it would be called if no CXXD-mods were being used in the built portion of the library.

Each optional variadic parameter is a Boost PP tuple with one to three elements. The Boost PP tuple designates:

- A CXXD-mod identifier. The CXXD-mod identifier is 'CXXD_' followed by the name of a CXXD-mod in uppercase. The CXXD-mod identifiers are specified in a separate list below.
- A mnemonic to be appended to the base library name if the CXXD-mod is using its C++ standard implementation.
- A mnemonic to be appended to the base library name if the CXXD-mod is using its Boost implementation.

The first tuple element is required. The second tuple element may be empty or left out. The third tuple element may be empty or left out.

A tuple element that is left out is considered 'empty'. An empty element is valid and means that nothing will be appended to the base name if the case is met.

When a tuple is specified as an optional parameter it is important that the mnemonic to be appended be different whether the CXXD-mod is using its C++ standard implementation or its Boost implementation, else there is no guarantee that a unique library name will be generated depending on the CXXD-mods being included. However lack of a difference in the mnemonics, if it occurs, is not flagged as an error in the processing of the macro.

If an optional parameter is not specified for a particular CXXD-mod which is included, a default value is appended as a mnemonic for the particular CXXD-mod if the CXXD-mod is using its C++ standard implementation; otherwise by default if the CXXD-mod is using its Boost implementation no value is appended by default.

The CXXD-mod identifiers and their default values for the C++ standard implementation are:

- CXXD_ARRAY,_ar
- CXXD_ATOMIC,_at
- CXXD_BIND,_bd
- CXXD_CHRONO,_ch

- CXXD_CONDITION_VARIABLE,_cv
- CXXD_ENABLE_SHARED_FROM_THIS,_es
- CXXD_FUNCTION,_fn
- CXXD_HASH,_ha
- CXXD_MAKE_SHARED,_ms
- CXXD_MEM_FN,_mf
- CXXD_MOVE,_mv
- CXXD_MUTEX,_mx
- CXXD_RANDOM,_rd
- CXXD_RATIO,_ra
- CXXD_REF,_rf
- CXXD_REGEX,_rx
- CXXD_SHARED_MUTEX,_sm
- CXXD_SHARED_PTR,_sp
- CXXD_SYSTEM_ERROR,_se
- CXXD_THREAD,_th
- CXXD_TUPLE,_tu
- CXXD_TYPE_INDEX,_ti
- CXXD_TYPE_TRAITS,_tt
- CXXD_UNORDERED_MAP,_um
- CXXD_UNORDERED_MULTIMAP,_up
- CXXD_UNORDERED_MULTISET,_ut
- CXXD_UNORDERED_SET,_us
- CXXD_WEAK_PTR,_wp
- CXXD_MODS_ALL,_std

The CXXD_MODS_ALL name refers to what happens if all the included CXXD-mods use either the C++ standard implementation or the Boost implementation. In this case, instead of each individual CXXD-mod having its mnemonic appended to the base name, a single mnemonic is appended to the base name. In the default case for CXXD_MODS_ALL the mnemonic '_std' is appended to the base name if all the included CXXD-mods use the C++ standard implementation and nothing is appended to the base name if all the included CXXD-mods use the Boost implementation.

The use of the optional parameters is the way to override the default processing for any particular CXXD-mod, or for all CXXD-mods.

Header <boost/cxx_dual/make_shared.hpp>

Dual library for the make_shared implementation.

Chooses either the Boost make_shared implementation or the C++ standard make_shared implementation.

```
CXXD_HAS_STD_MAKE_SHARED
CXXD_MAKE_SHARED_NS
CXXD_MAKE_SHARED_HEADER
CXXD_MAKE_SHARED_USE_STD
CXXD_MAKE_SHARED_USE_BOOST
```

Macro CXXD_HAS_STD_MAKE_SHARED

CXXD_HAS_STD_MAKE_SHARED — Determines whether the C++ standard make_shared implementation or the Boost make_shared implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/make_shared.hpp>

CXXD_HAS_STD_MAKE_SHARED
```

Description

The object-like macro expands to: 1 if the C++ standard make_shared implementation has been chosen 0 if the Boost make_shared implementation has been chosen.

Macro CXXD_MAKE_SHARED_NS

CXXD_MAKE_SHARED_NS — The make_shared namespace.

Synopsis

```
// In header: <boost/cxx_dual/make_shared.hpp>

CXXD_MAKE_SHARED_NS
```

Description

The object-like macro expands to the namespace for the make_shared implementation.

Macro CXXD_MAKE_SHARED_HEADER

CXXD_MAKE_SHARED_HEADER — The make_shared header file name.

Synopsis

```
// In header: <boost/cxx_dual/make_shared.hpp>

CXXD_MAKE_SHARED_HEADER
```

Description

The object-like macro expands to the include header file designation for the make_shared header file. The macro is used with the syntax: #include CXXD_MAKE_SHARED_HEADER

Macro CXXD_MAKE_SHARED_USE_STD

CXXD_MAKE_SHARED_USE_STD — Override macro for C++ standard make_shared implementation.

Synopsis

```
// In header: <boost/cxx_dual/make_shared.hpp>

CXXD_MAKE_SHARED_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard make_shared implementation. If the C++ standard make_shared implementation is not available a preprocessor error is generated.

Macro CXXD_MAKE_SHARED_USE_BOOST

CXXD_MAKE_SHARED_USE_BOOST — Override macro for Boost make_shared implementation.

Synopsis

```
// In header: <boost/cxx_dual/make_shared.hpp>

CXXD_MAKE_SHARED_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost make_shared implementation.

Header <boost/cxx_dual/mem_fn.hpp>

Dual library for the mem_fn implementation.

Chooses either the Boost mem_fn implementation or the C++ standard mem_fn implementation.

```


CXXD_HAS_STD_MEM_FN
CXXD_MEM_FN_NS
CXXD_MEM_FN_HEADER
CXXD_MEM_FN_USE_STD
CXXD_MEM_FN_USE_BOOST
```

Macro CXXD_HAS_STD_MEM_FN

CXXD_HAS_STD_MEM_FN — Determines whether the C++ standard mem_fn implementation or the Boost mem_fn implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/mem_fn.hpp>

CXXD_HAS_STD_MEM_FN
```

Description

The object-like macro expands to: 1 if the C++ standard mem_fn implementation has been chosen 0 if the Boost mem_fn implementation has been chosen.

Macro CXXD_MEM_FN_NS

CXXD_MEM_FN_NS — The mem_fn namespace.

Synopsis

```
// In header: <boost/cxx_dual/mem_fn.hpp>

CXXD_MEM_FN_NS
```

Description

The object-like macro expands to the namespace for the mem_fn implementation.

Macro CXXD_MEM_FN_HEADER

CXXD_MEM_FN_HEADER — The mem_fn header file name.

Synopsis

```
// In header: <boost/cxx_dual/mem_fn.hpp>

CXXD_MEM_FN_HEADER
```

Description

The object-like macro expands to the include header file designation for the mem_fn header file. The macro is used with the syntax: #include CXXD_MEM_FN_HEADER

Macro CXXD_MEM_FN_USE_STD

CXXD_MEM_FN_USE_STD — Override macro for C++ standard mem_fn implementation.

Synopsis

```
// In header: <boost/cxx_dual/mem_fn.hpp>

CXXD_MEM_FN_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard mem_fn implementation. If the C++ standard mem_fn implementation is not available a preprocessor error is generated.

Macro CXXD_MEM_FN_USE_BOOST

CXXD_MEM_FN_USE_BOOST — Override macro for Boost mem_fn implementation.

Synopsis

```
// In header: <boost/cxx_dual/mem_fn.hpp>

CXXD_MEM_FN_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost mem_fn implementation.

Header <boost/cxx_dual/move.hpp>

Dual library for the move implementation.

Chooses either the Boost move implementation or the C++ standard move implementation.

```


CXXD_HAS_STD_MOVE
CXXD_MOVE_NS
CXXD_MOVE_HEADER
CXXD_MOVE_USE_STD
CXXD_MOVE_USE_BOOST
```

Macro CXXD_HAS_STD_MOVE

CXXD_HAS_STD_MOVE — Determines whether the C++ standard move implementation or the Boost move implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/move.hpp>

CXXD_HAS_STD_MOVE
```

Description

The object-like macro expands to: 1 if the C++ standard move implementation has been chosen 0 if the Boost move implementation has been chosen.

Macro CXXD_MOVE_NS

CXXD_MOVE_NS — The move namespace.

Synopsis

```
// In header: <boost/cxx_dual/move.hpp>

CXXD_MOVE_NS
```

Description

The object-like macro expands to the namespace for the move implementation.

Macro CXXD_MOVE_HEADER

CXXD_MOVE_HEADER — The move header file name.

Synopsis

```
// In header: <boost/cxx_dual/move.hpp>

CXXD_MOVE_HEADER
```

Description

The object-like macro expands to the include header file designation for the move header file. The macro is used with the syntax: #include CXXD_MOVE_HEADER

Macro CXXD_MOVE_USE_STD

CXXD_MOVE_USE_STD — Override macro for C++ standard move implementation.

Synopsis

```
// In header: <boost/cxx_dual/move.hpp>

CXXD_MOVE_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard move implementation. If the C++ standard move implementation is not available a preprocessor error is generated.

Macro CXXD_MOVE_USE_BOOST

CXXD_MOVE_USE_BOOST — Override macro for Boost move implementation.

Synopsis

```
// In header: <boost/cxx_dual/move.hpp>

CXXD_MOVE_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost move implementation.

Header <boost/cxx_dual/mutex.hpp>

Dual library for the mutex implementation.

Chooses either the Boost mutex implementation or the C++ standard mutex implementation.

```
CXXD_HAS_STD_MUTEX
CXXD_MUTEX_NS
CXXD_MUTEX_HEADER
CXXD_MUTEX_USE_STD
CXXD_MUTEX_USE_BOOST
```

Macro CXXD_HAS_STD_MUTEX

CXXD_HAS_STD_MUTEX — Determines whether the C++ standard mutex implementation or the Boost mutex implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/mutex.hpp>

CXXD_HAS_STD_MUTEX
```

Description

The object-like macro expands to: 1 if the C++ standard mutex implementation has been chosen 0 if the Boost mutex implementation has been chosen.

Macro CXXD_MUTEX_NS

CXXD_MUTEX_NS — The mutex namespace.

Synopsis

```
// In header: <boost/cxx_dual/mutex.hpp>

CXXD_MUTEX_NS
```

Description

The object-like macro expands to the namespace for the mutex implementation.

Macro CXXD_MUTEX_HEADER

CXXD_MUTEX_HEADER — The mutex header file name.

Synopsis

```
// In header: <boost/cxx_dual/mutex.hpp>

CXXD_MUTEX_HEADER
```

Description

The object-like macro expands to the include header file designation for the mutex header file. The macro is used with the syntax: #include CXXD_MUTEX_HEADER

Macro CXXD_MUTEX_USE_STD

CXXD_MUTEX_USE_STD — Override macro for C++ standard mutex implementation.

Synopsis

```
// In header: <boost/cxx_dual/mutex.hpp>

CXXD_MUTEX_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard mutex implementation. If the C++ standard mutex implementation is not available a preprocessor error is generated.

Macro CXXD_MUTEX_USE_BOOST

CXXD_MUTEX_USE_BOOST — Override macro for Boost mutex implementation.

Synopsis

```
// In header: <boost/cxx_dual/mutex.hpp>

CXXD_MUTEX_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost mutex implementation.

Header <boost/cxx_dual/random.hpp>

Dual library for the random implementation.

Chooses either the Boost random implementation or the C++ standard random implementation.

```


CXXD_HAS_STD_RANDOM
CXXD_RANDOM_NS
CXXD_RANDOM_HEADER
CXXD_RANDOM_USE_STD
CXXD_RANDOM_USE_BOOST
```

Macro CXXD_HAS_STD_RANDOM

CXXD_HAS_STD_RANDOM — Determines whether the C++ standard random implementation or the Boost random implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/random.hpp>

CXXD_HAS_STD_RANDOM
```

Description

The object-like macro expands to: 1 if the C++ standard random implementation has been chosen 0 if the Boost random implementation has been chosen.

Macro CXXD_RANDOM_NS

CXXD_RANDOM_NS — The random namespace.

Synopsis

```
// In header: <boost/cxx_dual/random.hpp>

CXXD_RANDOM_NS
```

Description

The object-like macro expands to the namespace for the random implementation.

Macro CXXD_RANDOM_HEADER

CXXD_RANDOM_HEADER — The random header file name.

Synopsis

```
// In header: <boost/cxx_dual/random.hpp>

CXXD_RANDOM_HEADER
```

Description

The object-like macro expands to the include header file designation for the random header file. The macro is used with the syntax: #include CXXD_RANDOM_HEADER

Macro CXXD_RANDOM_USE_STD

CXXD_RANDOM_USE_STD — Override macro for C++ standard random implementation.

Synopsis

```
// In header: <boost/cxx_dual/random.hpp>

CXXD_RANDOM_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard random implementation. If the C++ standard random implementation is not available a preprocessor error is generated.

Macro CXXD_RANDOM_USE_BOOST

CXXD_RANDOM_USE_BOOST — Override macro for Boost random implementation.

Synopsis

```
// In header: <boost/cxx_dual/random.hpp>

CXXD_RANDOM_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost random implementation.

Header <boost/cxx_dual/ratio.hpp>

Dual library for the ratio implementation.

Chooses either the Boost ratio implementation or the C++ standard ratio implementation.

```


CXXD_HAS_STD_RATIO
CXXD_RATIO_NS
CXXD_RATIO_HEADER
CXXD_RATIO_USE_STD
CXXD_RATIO_USE_BOOST
```

Macro CXXD_HAS_STD_RATIO

CXXD_HAS_STD_RATIO — Determines whether the C++ standard ratio implementation or the Boost ratio implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/ratio.hpp>

CXXD_HAS_STD_RATIO
```

Description

The object-like macro expands to: 1 if the C++ standard ratio implementation has been chosen 0 if the Boost ratio implementation has been chosen.

Macro CXXD_RATIO_NS

CXXD_RATIO_NS — The ratio namespace.

Synopsis

```
// In header: <boost/cxx_dual/ratio.hpp>

CXXD_RATIO_NS
```

Description

The object-like macro expands to the namespace for the ratio implementation.

Macro CXXD_RATIO_HEADER

CXXD_RATIO_HEADER — The ratio header file name.

Synopsis

```
// In header: <boost/cxx_dual/ratio.hpp>

CXXD_RATIO_HEADER
```

Description

The object-like macro expands to the include header file designation for the ratio header file. The macro is used with the syntax: #include CXXD_RATIO_HEADER

Macro CXXD_RATIO_USE_STD

CXXD_RATIO_USE_STD — Override macro for C++ standard ratio implementation.

Synopsis

```
// In header: <boost/cxx_dual/ratio.hpp>

CXXD_RATIO_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard ratio implementation. If the C++ standard ratio implementation is not available a preprocessor error is generated.

Macro CXXD_RATIO_USE_BOOST

CXXD_RATIO_USE_BOOST — Override macro for Boost ratio implementation.

Synopsis

```
// In header: <boost/cxx_dual/ratio.hpp>

CXXD_RATIO_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost ratio implementation.

Header <boost/cxx_dual/ref.hpp>

Dual library for the ref implementation.

Chooses either the Boost ref implementation or the C++ standard ref implementation.

```


CXXD_HAS_STD_REF
CXXD_REF_NS
CXXD_REF_HEADER
CXXD_REF_USE_STD
CXXD_REF_USE_BOOST
```

Macro CXXD_HAS_STD_REF

CXXD_HAS_STD_REF — Determines whether the C++ standard ref implementation or the Boost ref implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/ref.hpp>

CXXD_HAS_STD_REF
```

Description

The object-like macro expands to: 1 if the C++ standard ref implementation has been chosen 0 if the Boost ref implementation has been chosen.

Macro CXXD_REF_NS

CXXD_REF_NS — The ref namespace.

Synopsis

```
// In header: <boost/cxx_dual/ref.hpp>

CXXD_REF_NS
```

Description

The object-like macro expands to the namespace for the ref implementation.

Macro CXXD_REF_HEADER

CXXD_REF_HEADER — The ref header file name.

Synopsis

```
// In header: <boost/cxx_dual/ref.hpp>

CXXD_REF_HEADER
```

Description

The object-like macro expands to the include header file designation for the ref header file. The macro is used with the syntax: #include CXXD_REF_HEADER

Macro CXXD_REF_USE_STD

CXXD_REF_USE_STD — Override macro for C++ standard ref implementation.

Synopsis

```
// In header: <boost/cxx_dual/ref.hpp>

CXXD_REF_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard ref implementation. If the C++ standard ref implementation is not available a preprocessor error is generated.

Macro CXXD_REF_USE_BOOST

CXXD_REF_USE_BOOST — Override macro for Boost ref implementation.

Synopsis

```
// In header: <boost/cxx_dual/ref.hpp>

CXXD_REF_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost ref implementation.

Header <boost/cxx_dual/regex.hpp>

Dual library for the regex implementation.

Chooses either the Boost regex implementation or the C++ standard regex implementation.

```


CXXD_HAS_STD_REGEX
CXXD_REGEX_NS
CXXD_REGEX_HEADER
CXXD_REGEX_USE_STD
CXXD_REGEX_USE_BOOST
```

Macro CXXD_HAS_STD_REGEX

CXXD_HAS_STD_REGEX — Determines whether the C++ standard regex implementation or the Boost regex implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/regex.hpp>

CXXD_HAS_STD_REGEX
```

Description

The object-like macro expands to: 1 if the C++ standard regex implementation has been chosen 0 if the Boost regex implementation has been chosen.

Macro CXXD_REGEX_NS

CXXD_REGEX_NS — The regex namespace.

Synopsis

```
// In header: <boost/cxx_dual/regex.hpp>

CXXD_REGEX_NS
```

Description

The object-like macro expands to the namespace for the regex implementation.

Macro CXXD_REGEX_HEADER

CXXD_REGEX_HEADER — The regex header file name.

Synopsis

```
// In header: <boost/cxx_dual/regex.hpp>

CXXD_REGEX_HEADER
```

Description

The object-like macro expands to the include header file designation for the regex header file. The macro is used with the syntax: #include CXXD_REGEX_HEADER

Macro CXXD_REGEX_USE_STD

CXXD_REGEX_USE_STD — Override macro for C++ standard regex implementation.

Synopsis

```
// In header: <boost/cxx_dual/regex.hpp>

CXXD_REGEX_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard regex implementation. If the C++ standard regex implementation is not available a preprocessor error is generated.

Macro CXXD_REGEX_USE_BOOST

CXXD_REGEX_USE_BOOST — Override macro for Boost regex implementation.

Synopsis

```
// In header: <boost/cxx_dual/regex.hpp>

CXXD_REGEX_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost regex implementation.

Header <boost/cxx_dual/shared_mutex.hpp>

Dual library for the shared mutex implementation.

Chooses either the Boost shared mutex implementation or the C++ standard shared mutex implementation.

```
CXXD_HAS_STD_SHARED_MUTEX
CXXD_SHARED_MUTEX_NS
CXXD_SHARED_MUTEX_HEADER
CXXD_SHARED_MUTEX_USE_STD
CXXD_SHARED_MUTEX_USE_BOOST
```

Macro CXXD_HAS_STD_SHARED_MUTEX

CXXD_HAS_STD_SHARED_MUTEX — Determines whether the C++ standard shared mutex implementation or the Boost shared mutex implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/shared_mutex.hpp>

CXXD_HAS_STD_SHARED_MUTEX
```

Description

The object-like macro expands to: 1 if the C++ standard shared mutex implementation has been chosen 0 if the Boost shared mutex implementation has been chosen.

Macro CXXD_SHARED_MUTEX_NS

CXXD_SHARED_MUTEX_NS — The shared mutex namespace.

Synopsis

```
// In header: <boost/cxx_dual/shared_mutex.hpp>

CXXD_SHARED_MUTEX_NS
```

Description

The object-like macro expands to the namespace for the shared mutex implementation.

Macro CXXD_SHARED_MUTEX_HEADER

CXXD_SHARED_MUTEX_HEADER — The shared mutex header file name.

Synopsis

```
// In header: <boost/cxx_dual/shared_mutex.hpp>

CXXD_SHARED_MUTEX_HEADER
```

Description

The object-like macro expands to the include header file designation for the shared mutex header file. The macro is used with the syntax: #include CXXD_SHARED_MUTEX_HEADER

Macro CXXD_SHARED_MUTEX_USE_STD

CXXD_SHARED_MUTEX_USE_STD — Override macro for C++ standard shared_mutex implementation.

Synopsis

```
// In header: <boost/cxx_dual/shared_mutex.hpp>

CXXD_SHARED_MUTEX_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard shared_mutex implementation. If the C++ standard shared_mutex implementation is not available a preprocessor error is generated.

Macro CXXD_SHARED_MUTEX_USE_BOOST

CXXD_SHARED_MUTEX_USE_BOOST — Override macro for Boost shared_mutex implementation.

Synopsis

```
// In header: <boost/cxx_dual/shared_mutex.hpp>

CXXD_SHARED_MUTEX_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost shared_mutex implementation.

Header <boost/cxx_dual/shared_ptr.hpp>

Dual library for the shared_ptr implementation.

Chooses either the Boost shared_ptr implementation or the C++ standard shared_ptr implementation.

```


CXXD_HAS_STD_SHARED_PTR
CXXD_SHARED_PTR_NS
CXXD_SHARED_PTR_HEADER
CXXD_SHARED_PTR_ONLY_HEADER
CXXD_SHARED_PTR_USE_STD
CXXD_SHARED_PTR_USE_BOOST
```

Macro CXXD_HAS_STD_SHARED_PTR

CXXD_HAS_STD_SHARED_PTR — Determines whether the C++ standard shared_ptr implementation or the Boost shared_ptr implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/shared_ptr.hpp>

CXXD_HAS_STD_SHARED_PTR
```

Description

The object-like macro expands to: 1 if the C++ standard shared_ptr implementation has been chosen 0 if the Boost shared_ptr implementation has been chosen.

Macro CXXD_SHARED_PTR_NS

CXXD_SHARED_PTR_NS — The shared_ptr namespace.

Synopsis

```
// In header: <boost/cxx_dual/shared_ptr.hpp>

CXXD_SHARED_PTR_NS
```

Description

The object-like macro expands to the namespace for the shared_ptr implementation.

Macro CXXD_SHARED_PTR_HEADER

CXXD_SHARED_PTR_HEADER — The shared_ptr header file name.

Synopsis

```
// In header: <boost/cxx_dual/shared_ptr.hpp>

CXXD_SHARED_PTR_HEADER
```

Description

The object-like macro expands to the include header file designation for the shared_ptr header file. The macro is used with the syntax: #include CXXD_SHARED_PTR_HEADER

The included header file includes the shared_ptr implementation as well as the weak_ptr, make_shared, and enable_shared_from_this implementations.

Macro CXXD_SHARED_PTR_ONLY_HEADER

CXXD_SHARED_PTR_ONLY_HEADER — The shared_ptr header file name.

Synopsis

```
// In header: <boost/cxx_dual/shared_ptr.hpp>

CXXD_SHARED_PTR_ONLY_HEADER
```

Description

The object-like macro expands to the include header file designation for the shared_ptr header file. The macro is used with the syntax: #include CXXD_SHARED_PTR_HEADER

The included header file includes only the shared_ptr implementation.

Macro CXXD_SHARED_PTR_USE_STD

CXXD_SHARED_PTR_USE_STD — Override macro for C++ standard shared_ptr implementation.

Synopsis

```
// In header: <boost/cxx_dual/shared_ptr.hpp>

CXXD_SHARED_PTR_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard shared_ptr implementation. If the C++ standard shared_ptr implementation is not available a preprocessor error is generated.

Macro CXXD_SHARED_PTR_USE_BOOST

CXXD_SHARED_PTR_USE_BOOST — Override macro for Boost shared_ptr implementation.

Synopsis

```
// In header: <boost/cxx_dual/shared_ptr.hpp>

CXXD_SHARED_PTR_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost shared_ptr implementation.

Header <boost/cxx_dual/system_error.hpp>

Dual library for the system error implementation.

Chooses either the Boost system error implementation or the C++ standard system error implementation.

```


CXXD_HAS_STD_SYSTEM_ERROR
CXXD_SYSTEM_ERROR_NS
CXXD_SYSTEM_ERROR_HEADER
CXXD_SYSTEM_ERROR_USE_STD
CXXD_SYSTEM_ERROR_USE_BOOST
```

Macro CXXD_HAS_STD_SYSTEM_ERROR

CXXD_HAS_STD_SYSTEM_ERROR — Determines whether the C++ standard system error implementation or the Boost system error implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/system_error.hpp>

CXXD_HAS_STD_SYSTEM_ERROR
```

Description

The object-like macro expands to: 1 if the C++ standard system error implementation has been chosen 0 if the Boost system error implementation has been chosen.

Macro CXXD_SYSTEM_ERROR_NS

CXXD_SYSTEM_ERROR_NS — The system error namespace.

Synopsis

```
// In header: <boost/cxx_dual/system_error.hpp>

CXXD_SYSTEM_ERROR_NS
```

Description

The object-like macro expands to the namespace for the system error implementation.

Macro CXXD_SYSTEM_ERROR_HEADER

CXXD_SYSTEM_ERROR_HEADER — The system error header file name.

Synopsis

```
// In header: <boost/cxx_dual/system_error.hpp>

CXXD_SYSTEM_ERROR_HEADER
```

Description

The object-like macro expands to the include header file designation for the system error header file. The macro is used with the syntax: #include CXXD_SYSTEM_ERROR_HEADER

Macro CXXD_SYSTEM_ERROR_USE_STD

CXXD_SYSTEM_ERROR_USE_STD — Override macro for C++ standard system error implementation.

Synopsis

```
// In header: <boost/cxx_dual/system_error.hpp>

CXXD_SYSTEM_ERROR_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard system error implementation. If the C++ standard system error implementation is not available a preprocessor error is generated.

Macro CXXD_SYSTEM_ERROR_USE_BOOST

CXXD_SYSTEM_ERROR_USE_BOOST — Override macro for Boost system error implementation.

Synopsis

```
// In header: <boost/cxx_dual/system_error.hpp>

CXXD_SYSTEM_ERROR_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost system error implementation.

Header <boost/cxx_dual/thread.hpp>

Dual library for the thread implementation.

Chooses either the Boost thread implementation or the C++ standard thread implementation.

```


CXXD_HAS_STD_THREAD
CXXD_THREAD_NS
CXXD_THREAD_HEADER
CXXD_THREAD_USE_STD
CXXD_THREAD_USE_BOOST
```

Macro CXXD_HAS_STD_THREAD

CXXD_HAS_STD_THREAD — Determines whether the C++ standard thread implementation or the Boost thread implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/thread.hpp>

CXXD_HAS_STD_THREAD
```

Description

The object-like macro expands to: 1 if the C++ standard thread implementation has been chosen 0 if the Boost thread implementation has been chosen.

Macro CXXD_THREAD_NS

CXXD_THREAD_NS — The thread namespace.

Synopsis

```
// In header: <boost/cxx_dual/thread.hpp>

CXXD_THREAD_NS
```

Description

The object-like macro expands to the namespace for the thread implementation.

Macro CXXD_THREAD_HEADER

CXXD_THREAD_HEADER — The thread header file name.

Synopsis

```
// In header: <boost/cxx_dual/thread.hpp>

CXXD_THREAD_HEADER
```

Description

The object-like macro expands to the include header file designation for the thread header file. The macro is used with the syntax: #include CXXD_THREAD_HEADER

Macro CXXD_THREAD_USE_STD

CXXD_THREAD_USE_STD — Override macro for C++ standard thread implementation.

Synopsis

```
// In header: <boost/cxx_dual/thread.hpp>

CXXD_THREAD_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard thread implementation. If the C++ standard thread implementation is not available a preprocessor error is generated.

Macro CXXD_THREAD_USE_BOOST

CXXD_THREAD_USE_BOOST — Override macro for Boost thread implementation.

Synopsis

```
// In header: <boost/cxx_dual/thread.hpp>

CXXD_THREAD_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost thread implementation.

Header <boost/cxx_dual/tuple.hpp>

Dual library for the tuple implementation.

Chooses either the Boost tuple implementation or the C++ standard tuple implementation.

```


CXXD_HAS_STD_TUPLE
CXXD_TUPLE_NS
CXXD_TUPLE_HEADER
CXXD_TUPLE_USE_STD
CXXD_TUPLE_USE_BOOST
```

Macro CXXD_HAS_STD_TUPLE

CXXD_HAS_STD_TUPLE — Determines whether the C++ standard tuple implementation or the Boost tuple implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/tuple.hpp>

CXXD_HAS_STD_TUPLE
```

Description

The object-like macro expands to: 1 if the C++ standard tuple implementation has been chosen 0 if the Boost tuple implementation has been chosen.

Macro CXXD_TUPLE_NS

CXXD_TUPLE_NS — The tuple namespace.

Synopsis

```
// In header: <boost/cxx_dual/tuple.hpp>

CXXD_TUPLE_NS
```

Description

The object-like macro expands to the namespace for the tuple implementation.

Macro CXXD_TUPLE_HEADER

CXXD_TUPLE_HEADER — The tuple header file name.

Synopsis

```
// In header: <boost/cxx_dual/tuple.hpp>

CXXD_TUPLE_HEADER
```

Description

The object-like macro expands to the include header file designation for the tuple header file. The macro is used with the syntax: #include CXXD_TUPLE_HEADER

Macro CXXD_TUPLE_USE_STD

CXXD_TUPLE_USE_STD — Override macro for C++ standard tuple implementation.

Synopsis

```
// In header: <boost/cxx_dual/tuple.hpp>

CXXD_TUPLE_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard tuple implementation. If the C++ standard tuple implementation is not available a preprocessor error is generated.

Macro CXXD_TUPLE_USE_BOOST

CXXD_TUPLE_USE_BOOST — Override macro for Boost tuple implementation.

Synopsis

```
// In header: <boost/cxx_dual/tuple.hpp>

CXXD_TUPLE_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost tuple implementation.

Header <boost/cxx_dual/type_index.hpp>

Dual library for the type index implementation.

Chooses either the Boost type index implementation or the C++ standard type index implementation.

```


CXXD_HAS_STD_TYPE_INDEX
CXXD_TYPE_INDEX_NS
CXXD_TYPE_INDEX_HEADER
CXXD_TYPE_INDEX_USE_STD
CXXD_TYPE_INDEX_USE_BOOST
```

Macro CXXD_HAS_STD_TYPE_INDEX

CXXD_HAS_STD_TYPE_INDEX — Determines whether the C++ standard type index implementation or the Boost type index implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/type_index.hpp>

CXXD_HAS_STD_TYPE_INDEX
```

Description

The object-like macro expands to: 1 if the C++ standard type index implementation has been chosen 0 if the Boost type index implementation has been chosen.

Macro CXXD_TYPE_INDEX_NS

CXXD_TYPE_INDEX_NS — The type index namespace.

Synopsis

```
// In header: <boost/cxx_dual/type_index.hpp>

CXXD_TYPE_INDEX_NS
```

Description

The object-like macro expands to the namespace for the type index implementation.

Macro CXXD_TYPE_INDEX_HEADER

CXXD_TYPE_INDEX_HEADER — The type index header file name.

Synopsis

```
// In header: <boost/cxx_dual/type_index.hpp>

CXXD_TYPE_INDEX_HEADER
```

Description

The object-like macro expands to the include header file designation for the type index header file. The macro is used with the syntax: #include CXXD_TYPE_INDEX_HEADER

Macro CXXD_TYPE_INDEX_USE_STD

CXXD_TYPE_INDEX_USE_STD — Override macro for C++ standard type_index implementation.

Synopsis

```
// In header: <boost/cxx_dual/type_index.hpp>

CXXD_TYPE_INDEX_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard type_index implementation. If the C++ standard type_index implementation is not available a preprocessor error is generated.

Macro CXXD_TYPE_INDEX_USE_BOOST

CXXD_TYPE_INDEX_USE_BOOST — Override macro for Boost type_index implementation.

Synopsis

```
// In header: <boost/cxx_dual/type_index.hpp>

CXXD_TYPE_INDEX_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost type_index implementation.

Header <boost/cxx_dual/type_traits.hpp>

Dual library for the type traits implementation.

Chooses either the Boost type traits implementation or the C++ standard type traits implementation.

```


CXXD_HAS_STD_TYPE_TRAITS
CXXD_TYPE_TRAITS_NS
CXXD_TYPE_TRAITS_HEADER
CXXD_TYPE_TRAITS_USE_STD
CXXD_TYPE_TRAITS_USE_BOOST
```

Macro CXXD_HAS_STD_TYPE_TRAITS

CXXD_HAS_STD_TYPE_TRAITS — Determines whether the C++ standard type traits implementation or the Boost type traits implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/type_traits.hpp>

CXXD_HAS_STD_TYPE_TRAITS
```

Description

The object-like macro expands to: 1 if the C++ standard type traits implementation has been chosen 0 if the Boost type traits implementation has been chosen.

Macro CXXD_TYPE_TRAITS_NS

CXXD_TYPE_TRAITS_NS — The type traits namespace.

Synopsis

```
// In header: <boost/cxx_dual/type_traits.hpp>

CXXD_TYPE_TRAITS_NS
```

Description

The object-like macro expands to the namespace for the type traits implementation.

Macro CXXD_TYPE_TRAITS_HEADER

CXXD_TYPE_TRAITS_HEADER — The type traits header file name.

Synopsis

```
// In header: <boost/cxx_dual/type_traits.hpp>

CXXD_TYPE_TRAITS_HEADER
```

Description

The object-like macro expands to the include header file designation for the type traits header file. The macro is used with the syntax: #include CXXD_TYPE_TRAITS_HEADER

Macro CXXD_TYPE_TRAITS_USE_STD

CXXD_TYPE_TRAITS_USE_STD — Override macro for C++ standard type_traits implementation.

Synopsis

```
// In header: <boost/cxx_dual/type_traits.hpp>

CXXD_TYPE_TRAITS_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard type_traits implementation. If the C++ standard type_traits implementation is not available a preprocessor error is generated.

Macro CXXD_TYPE_TRAITS_USE_BOOST

CXXD_TYPE_TRAITS_USE_BOOST — Override macro for Boost type_traits implementation.

Synopsis

```
// In header: <boost/cxx_dual/type_traits.hpp>

CXXD_TYPE_TRAITS_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost type_traits implementation.

Header <boost/cxx_dual/unordered_map.hpp>

Dual library for the unordered map implementation.

Chooses either the Boost unordered map implementation or the C++ standard unordered map implementation.

```


CXXD_HAS_STD_UNORDERED_MAP
CXXD_UNORDERED_MAP_NS
CXXD_UNORDERED_MAP_HEADER
CXXD_UNORDERED_MAP_USE_STD
CXXD_UNORDERED_MAP_USE_BOOST
```

Macro CXXD_HAS_STD_UNORDERED_MAP

CXXD_HAS_STD_UNORDERED_MAP — Determines whether the C++ standard unordered map implementation or the Boost unordered map implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/unordered_map.hpp>

CXXD_HAS_STD_UNORDERED_MAP
```

Description

The object-like macro expands to: 1 if the C++ standard unordered map implementation has been chosen 0 if the Boost unordered map implementation has been chosen.

Macro CXXD_UNORDERED_MAP_NS

CXXD_UNORDERED_MAP_NS — The unordered map namespace.

Synopsis

```
// In header: <boost/cxx_dual/unordered_map.hpp>

CXXD_UNORDERED_MAP_NS
```

Description

The object-like macro expands to the namespace for the unordered map implementation.

Macro CXXD_UNORDERED_MAP_HEADER

CXXD_UNORDERED_MAP_HEADER — The unordered map header file name.

Synopsis

```
// In header: <boost/cxx_dual/unordered_map.hpp>

CXXD_UNORDERED_MAP_HEADER
```

Description

The object-like macro expands to the include header file designation for the unordered map header file. The macro is used with the syntax: #include CXXD_UNORDERED_MAP_HEADER

Macro CXXD_UNORDERED_MAP_USE_STD

CXXD_UNORDERED_MAP_USE_STD — Override macro for C++ standard unordered map implementation.

Synopsis

```
// In header: <boost/cxx_dual/unordered_map.hpp>

CXXD_UNORDERED_MAP_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard unordered map implementation. If the C++ standard unordered map implementation is not available a preprocessor error is generated.

Macro CXXD_UNORDERED_MAP_USE_BOOST

CXXD_UNORDERED_MAP_USE_BOOST — Override macro for Boost unordered map implementation.

Synopsis

```
// In header: <boost/cxx_dual/unordered_map.hpp>

CXXD_UNORDERED_MAP_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost unordered map implementation.

Header <boost/cxx_dual/unordered_multimap.hpp>

Dual library for the unordered multimap implementation.

Chooses either the Boost unordered multimap implementation or the C++ standard unordered multimap implementation.

```


CXXD_HAS_STD_UNORDERED_MULTIMAP
CXXD_UNORDERED_MULTIMAP_NS
CXXD_UNORDERED_MULTIMAP_HEADER
CXXD_UNORDERED_MULTIMAP_USE_STD
CXXD_UNORDERED_MULTIMAP_USE_BOOST
```

Macro CXXD_HAS_STD_UNORDERED_MULTIMAP

CXXD_HAS_STD_UNORDERED_MULTIMAP — Determines whether the C++ standard unordered multimap implementation or the Boost unordered multimap implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/unordered_multimap.hpp>

CXXD_HAS_STD_UNORDERED_MULTIMAP
```

Description

The object-like macro expands to: 1 if the C++ standard unordered multimap implementation has been chosen 0 if the Boost unordered multimap implementation has been chosen.

Macro CXXD_UNORDERED_MULTIMAP_NS

CXXD_UNORDERED_MULTIMAP_NS — The unordered multimap namespace.

Synopsis

```
// In header: <boost/cxx_dual/unordered_multimap.hpp>

CXXD_UNORDERED_MULTIMAP_NS
```

Description

The object-like macro expands to the namespace for the unordered multimap implementation.

Macro CXXD_UNORDERED_MULTIMAP_HEADER

CXXD_UNORDERED_MULTIMAP_HEADER — The unordered multimap header file name.

Synopsis

```
// In header: <boost/cxx_dual/unordered_multimap.hpp>

CXXD_UNORDERED_MULTIMAP_HEADER
```

Description

The object-like macro expands to the include header file designation for the unordered multimap header file. The macro is used with the syntax: #include CXXD_UNORDERED_MULTIMAP_HEADER

Macro CXXD_UNORDERED_MULTIMAP_USE_STD

CXXD_UNORDERED_MULTIMAP_USE_STD — Override macro for C++ standard unordered multimap implementation.

Synopsis

```
// In header: <boost/cxx_dual/unordered_multimap.hpp>

CXXD_UNORDERED_MULTIMAP_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard unordered multimap implementation. If the C++ standard unordered multimap implementation is not available a preprocessor error is generated.

Macro CXXD_UNORDERED_MULTIMAP_USE_BOOST

CXXD_UNORDERED_MULTIMAP_USE_BOOST — Override macro for Boost unordered multimap implementation.

Synopsis

```
// In header: <boost/cxx_dual/unordered_multimap.hpp>

CXXD_UNORDERED_MULTIMAP_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost unordered multimap implementation.

Header <boost/cxx_dual/unordered_multiset.hpp>

Dual library for the unordered multiset implementation.

Chooses either the Boost unordered multiset implementation or the C++ standard unordered multiset implementation.

```
CXXD_HAS_STD_UNORDERED_MULTISET
CXXD_UNORDERED_MULTISET_NS
CXXD_UNORDERED_MULTISET_HEADER
CXXD_UNORDERED_MULTISET_USE_STD
CXXD_UNORDERED_MULTISET_USE_BOOST
```

Macro CXXD_HAS_STD_UNORDERED_MULTISET

CXXD_HAS_STD_UNORDERED_MULTISET — Determines whether the C++ standard unordered multiset implementation or the Boost unordered multiset implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/unordered_multiset.hpp>

CXXD_HAS_STD_UNORDERED_MULTISET
```

Description

The object-like macro expands to: 1 if the C++ standard unordered multiset implementation has been chosen 0 if the Boost unordered multiset implementation has been chosen.

Macro CXXD_UNORDERED_MULTISET_NS

CXXD_UNORDERED_MULTISET_NS — The unordered multiset namespace.

Synopsis

```
// In header: <boost/cxx_dual/unordered_multiset.hpp>

CXXD_UNORDERED_MULTISET_NS
```

Description

The object-like macro expands to the namespace for the unordered multiset implementation.

Macro CXXD_UNORDERED_MULTISET_HEADER

CXXD_UNORDERED_MULTISET_HEADER — The unordered multiset header file name.

Synopsis

```
// In header: <boost/cxx_dual/unordered_multiset.hpp>

CXXD_UNORDERED_MULTISET_HEADER
```

Description

The object-like macro expands to the include header file designation for the unordered multiset header file. The macro is used with the syntax: #include CXXD_UNORDERED_MULTISET_HEADER

Macro CXXD_UNORDERED_MULTISET_USE_STD

CXXD_UNORDERED_MULTISET_USE_STD — Override macro for C++ standard unordered multiset implementation.

Synopsis

```
// In header: <boost/cxx_dual/unordered_multiset.hpp>

CXXD_UNORDERED_MULTISET_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard unordered multiset implementation. If the C++ standard unordered multiset implementation is not available a preprocessor error is generated.

Macro CXXD_UNORDERED_MULTISET_USE_BOOST

CXXD_UNORDERED_MULTISET_USE_BOOST — Override macro for Boost unordered multiset implementation.

Synopsis

```
// In header: <boost/cxx_dual/unordered_multiset.hpp>

CXXD_UNORDERED_MULTISET_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost unordered multiset implementation.

Header <boost/cxx_dual/unordered_set.hpp>

Dual library for the unordered set implementation.

Chooses either the Boost unordered set implementation or the C++ standard unordered set implementation.

```


CXXD_HAS_STD_UNORDERED_SET
CXXD_UNORDERED_SET_NS
CXXD_UNORDERED_SET_HEADER
CXXD_UNORDERED_SET_USE_STD
CXXD_UNORDERED_SET_USE_BOOST
```

Macro CXXD_HAS_STD_UNORDERED_SET

CXXD_HAS_STD_UNORDERED_SET — Determines whether the C++ standard unordered set implementation or the Boost unordered set implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/unordered_set.hpp>

CXXD_HAS_STD_UNORDERED_SET
```

Description

The object-like macro expands to: 1 if the C++ standard unordered set implementation has been chosen 0 if the Boost unordered set implementation has been chosen.

Macro CXXD_UNORDERED_SET_NS

CXXD_UNORDERED_SET_NS — The unordered set namespace.

Synopsis

```
// In header: <boost/cxx_dual/unordered_set.hpp>

CXXD_UNORDERED_SET_NS
```

Description

The object-like macro expands to the namespace for the unordered set implementation.

Macro CXXD_UNORDERED_SET_HEADER

CXXD_UNORDERED_SET_HEADER — The unordered set header file name.

Synopsis

```
// In header: <boost/cxx_dual/unordered_set.hpp>

CXXD_UNORDERED_SET_HEADER
```

Description

The object-like macro expands to the include header file designation for the unordered set header file. The macro is used with the syntax: #include CXXD_UNORDERED_SET_HEADER

Macro CXXD_UNORDERED_SET_USE_STD

CXXD_UNORDERED_SET_USE_STD — Override macro for C++ standard unordered set implementation.

Synopsis

```
// In header: <boost/cxx_dual/unordered_set.hpp>

CXXD_UNORDERED_SET_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard unordered set implementation. If the C++ standard unordered set implementation is not available a preprocessor error is generated.

Macro CXXD_UNORDERED_SET_USE_BOOST

CXXD_UNORDERED_SET_USE_BOOST — Override macro for Boost unordered set implementation.

Synopsis

```
// In header: <boost/cxx_dual/unordered_set.hpp>

CXXD_UNORDERED_SET_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost unordered set implementation.

Header <boost/cxx_dual/valid_variants.hpp>

Contains CXXD_VALID_VARIANTS macro.

```


CXXD_VALID_VARIANTS(...)
```

Macro CXXD_VALID_VARIANTS

CXXD_VALID_VARIANTS — Tests for valid dual library variants.

Synopsis

```
// In header: <boost/cxx_dual/valid_variants.hpp>

CXXD_VALID_VARIANTS(...)
```

Description

The function-like macro tests for valid dual library variants. A variant consists of a series of CXXD-mod choices encoded as a variadic parameter. A CXXD-mod choice refers to whether the CXXD-mod chooses the C++ standard implementation or the Boost implementation.

If the CXXD-mod choices, based on the CXXD headers being included before the macro, is invoked match any one of the variants, the macro expands to 1, otherwise the macro expands to 0.

The macro provides a single invocation where the creator of a library or executable can test whether or not combinations of CXXD-mod choices match what the programmer wants to allow. In cases of header-only libraries or executables it should never be necessary to limit the combinations in any way, but in the case of a non-header only library the library implementor may well want to limit the acceptable combinations because supporting all possible variants, each with their own name and need to be built, might well prove onerous.

A variant is encoded by a series, known as a VMD sequence, of two-element Boost PP tuples. The first element is a particular CXXD-mod identifier, given in a following list, and the second element is 1 if the desired choice is the C++ standard implementation of that CXXD-mod or 0 if the desired choice is the Boost implementation of that CXXD-mod.

The VMD sequence of each choice makes up a combination of dual library choices, which denote a valid variant which the macro invoker says that his library will allow.

The list of CXXD-mod identifiers and their CXXD-mod is:

- CXXD_ARRAY,array
- CXXD_ATOMIC,atomic
- CXXD_BIND,bind
- CXXD_CHRONO,chrono
- CXXD_CONDITION_VARIABLE,condition_variable
- CXXD_ENABLE_SHARED_FROM_THIS,enable_shared_from_this
- CXXD_FUNCTION,function
- CXXD_HASH,hash
- CXXD_MAKE_SHARED,make_shared
- CXXD_MEM_FN,mem_fn
- CXXD_MOVE,move
- CXXD_MUTEX,mutex
- CXXD_RANDOM,random
- CXXD_RATIO,ratio
- CXXD_REF,ref
- CXXD_REGEX,regex
- CXXD_SHARED_MUTEX,shared_mutex
- CXXD_SHARED_PTR,shared_ptr
- CXXD_SYSTEM_ERROR,system_error
- CXXD_THREAD,thread
- CXXD_TUPLE,tuple
- CXXD_TYPE_INDEX,type_index
- CXXD_TYPE_TRAITS,type_traits
- CXXD_UNORDERED_MAP,unordered_map
- CXXD_UNORDERED_MULTIMAP,unordered_multimap
- CXXD_UNORDERED_MULTISET,unordered_multiset
- CXXD_UNORDERED_SET,unorderd_set

- CXXD_WEAK_PTR,weak_ptr
- CXXD_MODS_ALL,all mods

As can be seen each CXXD-mod identifier is 'CXXD_' followed by the uppercase name of the CXXD-mod.

The CXXD_MODS_ALL identifier refers to all of the included CXXD headers choosing either the C++ standard implementation or the Boost implementation. Therefore if this identifier is used its Boost PP tuple should be the only one in the VMD sequence for that variant.

The macro invoker must pass at least one variant as a variadic parameter otherwise there is no point in using this macro, but may specify any number of further variants as variadic parameters.

A protoypical variant will look like: (CXXD_XXX,1 or 0)(CXXD_YYY,1 or 0)(CXXD_ZZZ,1 or 0)...

where CCXD_XXX, CXXD_YYY, and CXXD_ZZZ are one of the CXXD-mod identifiers listed above and the '1 or 0' denotes either the C++ standard implementation or Boost implementation as a choice for that CXXD-mod. There can be one or more Boost PP tuples in the VMD sequence which denote the variant. A variant is an 'AND' proposition where each Boost PP tuple in the VMD sequence must be true for the variant to match. Each variant as a variadic parameter is an 'OR' proposition where any variant must match for the macro to return 1. Otherwise the macro returns 0 if none of the variants match.

Header <boost/cxx_dual/weak_ptr.hpp>

Dual library for the weak_ptr implementation.

Chooses either the Boost weak_ptr implementation or the C++ standard weak_ptr implementation.

```


CXXD_HAS_STD_WEAK_PTR
CXXD_WEAK_PTR_NS
CXXD_WEAK_PTR_HEADER
CXXD_WEAK_PTR_USE_STD
CXXD_WEAK_PTR_USE_BOOST
```

Macro CXXD_HAS_STD_WEAK_PTR

CXXD_HAS_STD_WEAK_PTR — Determines whether the C++ standard weak_ptr implementation or the Boost weak_ptr implementation has been chosen.

Synopsis

```
// In header: <boost/cxx_dual/weak_ptr.hpp>

CXXD_HAS_STD_WEAK_PTR
```

Description

The object-like macro expands to: 1 if the C++ standard weak_ptr implementation has been chosen 0 if the Boost weak_ptr implementation has been chosen.

Macro CXXD_WEAK_PTR_NS

CXXD_WEAK_PTR_NS — The weak_ptr namespace.

Synopsis

```
// In header: <boost/cxx_dual/weak_ptr.hpp>

CXXD_WEAK_PTR_NS
```

Description

The object-like macro expands to the namespace for the weak_ptr implementation.

Macro CXXD_WEAK_PTR_HEADER

CXXD_WEAK_PTR_HEADER — The weak_ptr header file name.

Synopsis

```
// In header: <boost/cxx_dual/weak_ptr.hpp>

CXXD_WEAK_PTR_HEADER
```

Description

The object-like macro expands to the include header file designation for the weak_ptr header file. The macro is used with the syntax: #include CXXD_WEAK_PTR_HEADER

Macro CXXD_WEAK_PTR_USE_STD

CXXD_WEAK_PTR_USE_STD — Override macro for C++ standard weak_ptr implementation.

Synopsis

```
// In header: <boost/cxx_dual/weak_ptr.hpp>

CXXD_WEAK_PTR_USE_STD
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the C++ standard weak_ptr implementation. If the C++ standard weak_ptr implementation is not available a preprocessor error is generated.

Macro CXXD_WEAK_PTR_USE_BOOST

CXXD_WEAK_PTR_USE_BOOST — Override macro for Boost weak_ptr implementation.

Synopsis

```
// In header: <boost/cxx_dual/weak_ptr.hpp>

CXXD_WEAK_PTR_USE_BOOST
```

Description

The object-like macro, when defined to nothing prior to including this header file, forces the dual library choice of the Boost weak_ptr implementation.

Dual library knowledge and concerns

Using a dual library, as CXXD does, means that a programmer should have knowledge of any differences which may exist between the Boost implementation of a dual library and the C++ standard implementation of that same dual library.

There are a number of areas for the programmer to consider when using a particular dual library:

1. Does the particular syntax being used function in the same way for each dual library implentation ?
2. Can one-off code be written for situations in which the implementation of a dual library differs ?
3. Does a compiler implementation of the C++ standard library implementation of a dual library correspond to what the C++ standard requires ?
4. If the compiler being used changes will dual library code still work as expected ?

All of the above are valid concerns. I have had expressed to me that using CXXD is not a valid choice to use because it masks the issues I bring up above. I can understand that point of view. Furthermore I have had it expressed to me that CXXD should somehow document any differences between the implementation of the Boost library and the C++ standard library for each of the dual libraries supported. Again this is a valid point of view.

The design of CXXD is that the advantages of being able to offer to the programmer the choice of using either the Boost implementation or the C++ standard implementation of a dual library, while using both with the same code, offsets the issues above and the concerns of others in using CXXD in the designer's mind. But I am aware of the issues and concerns stated above and want to mention them here in the documentation to CXXD.

The issues and concerns in my own mind are very much the same whenever a particular implementation of any library is chosen, whether that library is a Boost library, a C++ standard library, or a 3rd party library. That each CXXD-mod really involves two different implementations, or libraries if you will, is something I have made plain in the documentation, and I want to reiterate it here. Knowledge of those libraries, for any particular CXXD-mod, is still of first importance in programming using CXXD. Essentially CXXD is a framework for supporting dual libraries, but it is not an excuse for not understanding the functionality of a particular dual library in the first place.

Although I have done so in other areas of the documentation I would like to again emphasize the importance of documenting when a dual library is being used by the end-user. This is especially true when CXXD is being used in a library, whether that library is a header-only library or non-header only library where some part of the library is being built into a shared or static library. Without documenting that a particular dual library is being used a programmer cannot know, depending on the compiler implementation and the compiler's command line parameters, that either the Boost implementation or the C++ standard implementation of the dual library is being used in the code.

Index