Document Number: P2581R2
Date: 2022-10-12
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

# Specifying the Interoperability of Built Module Interface Files

## Changes

- R2
  - Introduce the “Problem Statement” section to help bridge the gap in understanding that was present in previous SG15 meetings.
  - Expand the example that existed before in the requirements section and move it to a new section.
  - Broad rewrite of the sections “Producing a new BMI in the context of another translation unit” and “Built Module Interface Compatibility Identifiers”.
- R1
  - Change “Binary Module Interface” to “Built Module Interface”.
  - Introduce a new section exploring the outcome of having independent parsing context between the translation unit importing a module and the translation unit declaring a module interface.
  - Editorial changes throughout the paper for better readability, including removing redundancy with the new section.

## Abstract

This paper specifies a mechanism to allow build systems to identify if a built module interface shipped with a pre-built library can be used directly, or if the build system needs to produce its own version of the built module interface file, as well as instructions on how the build system should assemble the command line to produce that BMI.

## Introduction

Built module interface files are an implementation aspect on how modules are reused across different invocations of the compiler without the need for a new translation. The format of those files is implementation-defined. With the exception of MSVC, implementations are defining them to be only as interoperable as precompiled headers were.

While there is no fundamental problem with that approach, it does create an additional problem for libraries being shipped as prebuilt artifacts. In some cases, built module interface files in those artifacts are going to be reusable in a given compiler invocation, and in other cases they won’t.

Document Number: P2581R2
Date: 2022-10-12
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

**The purpose of this paper is to establish a mechanism to allow for the clear identification of whether a given built module interface file will be usable within a given build context without the need to actually read the BMI files and without the need to invoke the compiler for each module, as well as establish a mechanism for the build system to correctly assemble a compiler invocation for the translation of a module interface from a different project.**.

# Problem Statement

The surface of compatibility of the BMI file, in most implementations, is significantly narrower than the surface of ABI compatibility. Follows a non-exhaustive list of scenarios that illustrate that difference:

- It's possible, and relatively common, for different parts of the same program to be compiled using different compilers, however, none of the implementations currently support importing the BMI file of a different compiler.
- Clang and GCC currently have a much narrower compatibility surface for the generated BMI file, where a change in compiler version is sufficient to generate an incompatible BMI.
- Even when using the same compiler version, some options may result in a BMI produced by the same compiler executable to be unusable when being imported in a different translation unit.

Early implementations of build system support for C++ Modules have mostly focused on scenarios where all the BMIs involved were compatible through all translation units in the project. This is reasonably easy to achieve in scenarios where the build of all of the C++ code happens within a single build system configuration, and where *the use of an option that would create such incompatibility across translation units could be considered an user error*.

However, as the scope of the support for C++ modules is expanded, the assumptions that can be made on a "single build system" context can no longer be considered. The distinctions that arise when you have multiple build systems collaborating to produce a single program were the subject of a previous paper[1].

## Semantics of the import declaration

The C++ standard defines the semantics of the import declaration only in terms of the impact on the interpretation of the language itself. While module code is still bound by the One-Definition-Rule, that is far from a concrete direction for implementers. In practice, there are

[1]Ruoso, Daniel (2021). Requirements for Usage of C++ Modules at Bloomberg. https://wg21.link/P2409R0

no additional restrictions on the implementation of modules than there were for simple source inclusion.

In principle, there is nothing in the standard that indicates that it would be invalid for an implementation to always independently and redundantly translate the imported modules directly from their source code. Even when that would result in multiple independent translations of the same module code contributing to the final program.

At the same time, one of the biggest motivators for introducing modules was the expectation that implementations should be able to translate the module interface only once, and reuse that translation for every import declaration of the same module.

The disconnect between those two motivations created a gap between the understanding of the semantics of the language, and the requirements imposed on build systems that haven't been fully bridged yet.

## Pre-compiled headers as prior art

One of the most influential aspects for the work on modules by the GCC and Clang implementations was the fact that C++ Modules were seen as an extension of a feature that was previously supported by both, which was the concept of "pre-compiled headers"[2].

The difference is that pre-compiled headers were initially implemented entirely as an optimization to the build process, without any direct impact on the semantics of the languages. However, it was from that implementation that the support for modules has been extended from.

That approach meant that the BMI files produced by both Clang and GCC are oriented towards the optimization requirements, reducing as much as possible the amount of duplicated work between two translation units that import the same module.

That means that different versions of the compiler likely can't share the same BMI files, it also means that some specific compiler flags may result in internal changes to the layout of the BMI file that make it unusable when the compiler receives a different set of arguments.

While the benefits of such optimization are relevant, at the same time it pushes the build systems in the direction of semantically considering the BMI files as an optimization step that is subject to pessimization at any point.

## Compiler arguments as opaque tokens

While in some codebases it is viable to enforce a strict governance of how much access the developers have over setting potentially-incompatible flags in order to mitigate the risk of

[2] Clang Modules are, to a large extent, a direct extension of the concept of pre-compiled headers.

producing BMI files that can't be used where they're needed, the reality is that for the industry as a whole, particularly in the Open Source world, that requirement is a non-starter.

CMake is a widely used build system for C++ code, and a cursory look at various projects in Github will demonstrate that developers will often configure different targets in the same project with different compiler arguments.

While there has been significant work in CMake to abstract those expected features and options into higher-level concepts, it is not realistic to presume that we can encode the features of all compilers, present and future, in abstract concepts. Build systems, therefore, have to consider compiler flags as opaque tokens.

## What happens when it goes wrong

In the beginning of this section, this paper made the contrast between large projects where it would be reasonable to assume that any mismanagement of flags that would result in a BMI not being usable where expected to be "user error".

In the wider industry, particularly in the Open Source world or more decentralized organizations, this approach is not viable, as the outcome would be an ever increasing support cost for the maintenance of C++ projects, which would hinder the reuse of C++.

It is important that the user should be able to communicate with the build system enough information to allow the build system to understand when a BMI file created in one context is going to be usable in another context, and when the build system needs to create the plans for alternative translations of the same module interface.

## Mitigating ODR violations in the ecosystem

Another important motivator in the approach to implementing C++ Modules is that they offer us with mechanisms to mitigate ODR violations. The transition from simple source inclusion to a full independent translation of the module interface offers an important opportunity to reduce the amount of confusion that exists on the use of include directories and compiler definitions.

In the source inclusion ecosystem, there is no way to isolate the translation requirements from a library being consumed from the translation requirements from the code using that library. The only solution for managing the requirements of transitive dependencies is to concatenate compiler options, which sometimes leads to issues that are difficult to debug.

With modules, however, we can separate the translation context of the module interface from the context of the translation unit importing that module. This allows greater control over how a module interface is translated, and should have a positive effect on the mitigations of ODR violations in the ecosystem.

## Balancing independent translation context with compatibility

This, finally, is the crux of the problem. Build systems need to balance the need for the independent translation context between the module interface unit and the unit importing that module with the need to produce a BMI for the same interface that can actually be used as input for the translation of the unit importing that module.

To use CMake as an example, if the build system presumes that the BMI file is “owned” by the library target that contains the module interface unit, there’s a significant risk that the BMI will not be compatible with the translation unit importing it, resulting in a build error if it happens that the user has set compiler arguments that make the BMI unusable in that context.

And this compounds with the fact that it is common for a CMake project to either build into itself third-party code, or to import targets produced by a different invocation of CMake, means that the level of coordination that would be required to ensure the BMI is usable by the importing unit is not realistic for the industry as a whole.

The result of that reasoning is that the assembling of the compiler arguments for the production of the BMI for a given module needs to be started in the context of the translation unit declaring the import, in order to guarantee that the produced BMI will be usable.

On the other hand, if every translation unit doing an import had to do its own translation of the module interface units, it would result in an exponential increase in the number of translations. Therefore the build system needs to be able to optimize that away by being able to identify when BMIs produced in different contexts are usable.

At the same time, if we just use the same compiler arguments from the translation unit declaring the import when translating an interface, we would lose the isolation of arguments that was set as a goal to mitigate ODR violations, not to mention that it would significantly reduce the amount of reusability of the BMI files across different translation units declaring the same import.

Therefore we need to be able to isolate which compiler arguments to carry from the metadata associated with the module interface unit when producing the BMI in the context of the unit declaring the import, but we also need to be able to declare which arguments from the translation unit declaring the import need to be suppressed when producing the BMIs it needs.

# Example

This section will illustrate the problem through an example. For the purposes of this illustration, we will assume the scenario is using GCC 11 on GNU/Linux. In this example, we have three different translation units, which we should assume would be in different projects and therefore not with a unified control of all the compiler arguments.

Let’s imagine that the first project publishes a module A, then a second project publishes a module B that imports module A, and finally the third project publishes a module C that imports both A and B. Follow the source code of the three different modules:

a.cpp

```
module;
#include <string>
#include <some_header.h>
export module A;
export std::string foo() {
 return DEFINED_IN_SOME_HEADER_H;
}
```

b.cpp

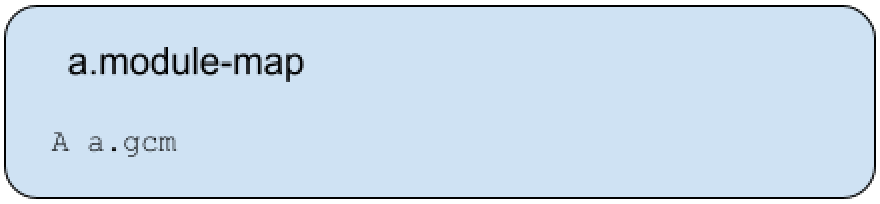
```
module;
#include <string>
#include <other_header.h>
export module B;
import A;
export std::string bar() {
  return foo() +
   DEFINED_IN_OTHER_HEADER_H;
}
```

c.cpp

```
module;
#include <string>
#include <another_header.h>
export module C;
import A;
import B;
export std::string baz() {
 return foo() + bar() +
   DEFINED_IN_ANOTHER_HEADER_H; }
```

Each project will produce a BMI and an object file output for their module, and when doing that, the control over the arguments sent to the compiler is local to the specific project. In the case of gcc, the path to the BMI files (both for input and output) is provided through the `-fmodule-mapper=file` option.

Therefore on the first compilation, we need an additional file to instruct where the BMI output should be:



a.module-map

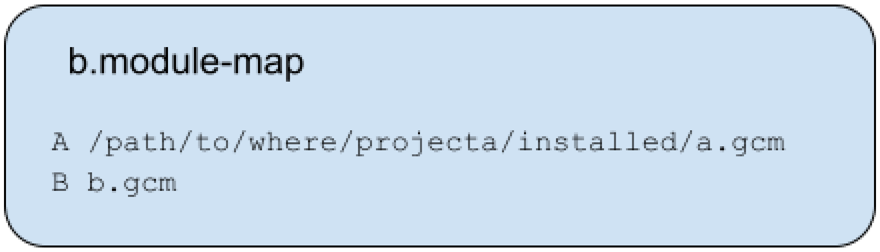
```
A a.gcm
```

At which point we should be able to invoke gcc to translate a.cpp in the context of project A.

```
g++ -std=c++20 -fmodules-ts \
  -fmodule-mapper=a.module-map \
  -fictional-option-1 \
  -I/path/to/some_header/ \
  -DOPTION_FOR_SOME_HEADER=1 \
  -o a.o -c a.cpp
```

This will produce both `a.o` and `a.gcm`, which is meant to be used in other projects.

On project B, however, the module map will need to include the path to the BMIs of both A and B. The build system will need to assemble a module map to be used in that translation unit, which would look something like:



b.module-map

```
A /path/to/where/projecta/installed/a.gcm
B b.gcm
```

At which point, the invocation of the compiler can be assembled from the context of project B.

```
g++ -std=c++20 -fmodules-ts \
  -fmodule-mapper=b.module-map \
  -fictional-option-2 \
  -I/path/to/other_header/ \
  -DOPTION_FOR_OTHER_HEADER=1 \
  -o b.o -c b.cpp
```

If the BMI for module A is compatible with this invocation of the compiler, everything should work. The build system, however, has no way of evaluating whether that is true or not. At this moment, if they are incompatible, this will just be an error in the translation of `b.cpp`, because `a.gcm` couldn't be used in this context.

In environments where the choice of compiler and build flags can be centrally managed, we could conceivably consider such incompatibility to be an user error, and the solution would be to fix the flags in order for the BMI to become compatible. However, as discussed in the previous section, this is not a viable approach for decentralized organizations or the Open Source ecosystem.

For the sake of this illustration, let's consider that the use of `-fictional-option-1` generates a BMI that is incompatible with an invocation where `-fictional-option-2` is used, even though it's perfectly valid to link objects produced with those options into the same program.

**The first unanswered question here, then, is: How can the build system tell whether `a.gcm` is usable in the context of the translation of `b.cpp`?**

Now, let's pretend that the build system had that problem solved, and knew that `a.gcm` was incompatible, and for that reason it knows that it needs to produce a new version of it that could be used in the context of the translation of `b.cpp`.

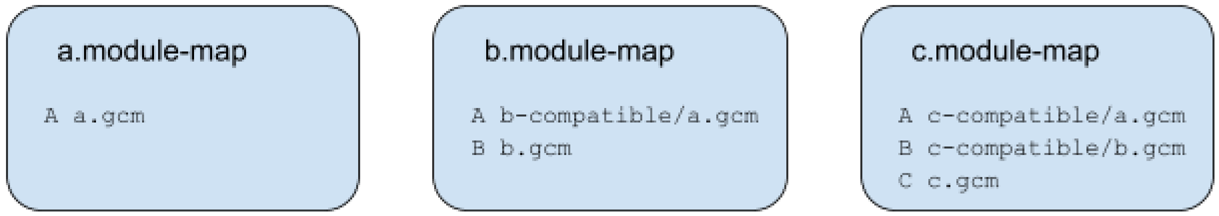
Since the build system has to treat the compiler arguments as opaque tokens, it can't really try to infer why the BMI was incompatible and try to correct those inconsistencies from the invocation used to produce the original `a.gcm`.

However, if it just uses the arguments from the translation of `b.cpp`, it will at the same time be missing the preprocessor arguments that are necessary to translate `a.cpp` and incorrectly exposing the preprocessor arguments from `b.cpp` to that translation unit.

**The second unanswered question: How can the build system decide the compiler arguments required to produce a version of `a.gcm` that is usable in the context of the translation of `b.cpp`?**

The final step, then, will happen with the translation of `c.cpp`. For the sake of illustration, let's imagine it uses `-fictional-option-3`, which would make all other previous BMIs referenced in this example unusable in this context, even if they're all still ABI-compatible and could be linked into a single program.

The end result is that we have three different module maps, resulting in three different versions of a.gcm, and two different versions of b.gcm, and they all contribute to the final program.


a.module-map

```
A a.gcm
```

b.module-map

```
A b-compatible/a.gcm
B b.gcm
```

c.module-map

```
A c-compatible/a.gcm
B c-compatible/b.gcm
C c.gcm
```

In this particular illustration, it generated the worst of all scenarios, where we had to translate every module interface once for each translation unit that imported them. And it's likely that in very decentralized organizations, the amount of reuse of BMIs across projects will in fact be substantially smaller than in organizations where flags are centrally managed.

However, there is a wide range of possibilities, and therefore it should be possible to optimize the reuse of the BMIs by governance on the use of flags even if that won't strictly apply to the entire codebase. Any amount of reuse is a significant optimization.

## Producing a new BMI in the context of another translation unit

In the previous section we worked through an example that demonstrated that we need to be able to reason about which flags need to be used coherently, and which flags are allowed to be different. This paper introduces the following categories to organize those arguments:

- *Basic Toolchain Configuration Arguments*: The coherency on the toolchain configuration is not a new requirement. This is the same set of arguments that need to be coherent when two independent translation units in the same program include the same header today. This needs to account for the "benign ODR violations" that are common practice today.
- *BMI-Sensitive Arguments*: Current implementations, particularly clang and gcc, currently have a BMI output that is sensitive to changes that would otherwise not cause an incoherency on the basic toolchain configuration. For instance: libstdc++ supports translation units using different language standard versions to communicate within the same program (with documented limitations), however it is not valid to use a BMI translated with a different language standard version than the unit importing it.
- *Local Preprocessor Arguments*: Preprocessor arguments used when translating the module interface, but that would otherwise not be needed in the translation of the unit importing that module, also arguments needed in the translation unit importing a module but that are not needed when translating the module interface being imported.
- *Other Arguments*: Those are arguments that do not affect the preprocessor, and are also not relevant either for ABI coherency or the BMI compatibility. They do not substantially change how the translation is made and therefore this paper will not cover their usage.

Those categories, however, represent higher-level semantics. It is not the case that the build system can introspect the command line used to produce a BMI on another project and decide which arguments fall on which of the categories. They need to be authored by the engineer maintaining the build system.

On the other hand, we don't need to materialize the distinction between the Basic Toolchain Configuration arguments and the BMI-Sensitive arguments or the Other Arguments. The coherency between Basic Toolchain Configuration arguments across different translation units is a problem that maintainers of build systems and package managers need to solve today, regardless of modules.

The only category of arguments that we need to be concerned with in this context is actually the Local Preprocessor Arguments, in the modules ecosystem we will need the users to specify which of the arguments given to the compiler are in that category.

**This paper proposes that build systems should offer a mechanism to identify which of the options used in a translation unit are a Local Preprocessor Argument.**

**This paper proposes that in order to assemble the compiler invocation for a BMI compatible with the translation unit importing a module, you start with the invocation of the translation unit declaring the import, remove the Local Preprocessor Arguments from that invocation and then append the Local Preprocessor Arguments from the invocation originally used in the module interface unit.**

In the example used in the previous section, the user would document the intent that the arguments `-I/path/to/some_header/ -DOPTION_FOR_SOME_HEADER=1` are the Local Preprocessor Arguments for a.cpp, and this is the only part of that invocation that we need to take.

Likewise, for the translation of b.cpp, the user would document that the intent that the arguments `-I/path/to/other_header/ -DOPTION_FOR_OTHER_HEADER=1` are the Local Preprocessor Arguments for b.cpp, which is what we need to filter out from the compiler invocation.

Therefore the compiler invocation for the version of a.gcm that would be usable in the context of the translation of b.cpp would look like this:

```
g++ -std=c++20 -fmodules-ts \
  -fmodule-mapper=b-compatible/a.module-map \
  -fictional-option-2 \
  -I/path/to/some_header/ \
  -DOPTION_FOR_SOME_HEADER=1 \
  -fmodule-only -c a.cpp
```

The assumption is that those two invocations will not generate ODR violations, but that assumption is the same that we already have in the context of source inclusion today, so they are not a new requirement.

# Built Module Interface Compatibility Identifiers

The previous section established a way to produce a BMI when an usable one cannot be found. In this section we will propose a mechanism to identify whether an existing BMI is compatible with the current context.

The proposal here is that **every built module interface file will have an identifier for the compatibility of the file**. The module metadata shipped with a pre-built library will advertise the list of BMI files that were made available with the binaries. That list will include the compatibility identifier of the BMI alongside the location of the file itself. Any consumer of the library will obtain the identifier for their own build context, and if a matching identifier is found, the build system will know that the BMI can be reused.

## Defining the Compatibility Identifier

The scope of compatibility for consuming an existing built module interface file is defined by the union of the Basic Toolchain Configuration Arguments and the BMI-Sensitive arguments. And it should explicitly exclude Local Preprocessor Arguments. Those arguments are the ones that should be used to define the compatibility identifier.

**The compiler should offer an interface (e.g.: command line option) that will produce to the standard output the identifier as the first line. That command line should accept the regular compiler arguments as if parsing a module interface unit (although without specifying a translation unit). Arguments that are irrelevant to the compatibility of the built module interface output should be ignored.**

**The build system should take the compiler invocation for a translation unit, remove the Local Preprocessor Arguments and the reference to the specific translation unit and invoke the compiler in that mode in order to obtain the compatibility identifier for BMIs produced and consumed by that translation unit.**

A build system would, therefore, only need to invoke the compiler once for each set of arguments to get the identifier that would be used for all translation units using the same arguments, and then search the module metadata shipped with the pre-built library for a BMI with that particular identifier.

If a BMI with that identifier is found, the build system doesn’t need to emit instructions on how to build that BMI again, however if a BMI is not found, then those arguments are concatenated with the Local Preprocessor Arguments advertised by the module in order to produce a local BMI.

When generating install instructions, the build system would use that same identifier when producing the metadata for the modules that are part of the library being distributed.

## Compilers that support multiple inputs

While the author of this paper is not aware of any implementation that supports a number of importing BMIs of different compatibility profiles concurrently, this is a significant optimization, and the author considers that it is justifiable to specify it ahead of time.

**A compiler that supports multiple compatibility for the same context should return the additional identifiers on the same output, following the first line. In other words, the first line returned specifies the compatibility identifier of the BMI that will be produced, the following lines specify alternative BMIs that can be consumed**.

One hypothetical use case for this would be if clang produces its own BMI when parsing the interface module units, but is also capable of parsing module files in the IFC format published by Microsoft.