Document Number: P2581R0
Date: 2022-04-25
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Specifying the Interoperability of Binary Module Interface Files

Abstract

This paper specifies a mechanism to allow build systems to identify if a binary 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 binary module interface file.

Introduction

Binary Module Interface files are an implementation aspect on how modules are reused across different invocations of the compiler without the need for a full reparsing. 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, binary module interface files in those artifacts are going to be reusable by a compiler, and in other cases they won't.

The purpose of this paper is to establish a mechanism to allow for the clear identification of whether a given binary module interface file will be usable within a given build context without the need to actually read those files and without the need to invoke the compiler for each module.

Metadata driven discovery

The current consensus on SG15 is that module discovery should be driven by metadata. The subject of how that metadata will be found or the format for that metadata is still being debated. However, it is already clear that a C++ module library will need to identify that it provides modules, and it needs to be able to offer binary module interface files for those as an optimization for the cases where they can be used.

The end result is that a build system should be able to identify where the modules are and if there are candidate binary module interface files to be used. However, **we still don't have a mechanism to predict whether that binary module interface file is compatible with the compiler as used in the current context or not**.

Document Number: P2581R0
Date: 2022-04-25
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Binary Module Interface Format Identifiers

The proposal here is that **every Binary Module Interface file will have an identifier for the precise format of the file**. That identifier will then be presented in the metadata alongside the location of the file itself. This will also allow a library to be shipped with more than one binary module interface file for the same module, providing for different formats. It will also allow a build system to decide whether a binary module interface file is compatible without having to read the file at all.

Defining the Format Identifier

The scope of compatibility for consuming a binary module interface file can be limited by various factors, such as:

- The specific version and build of the compiler;
- The language standard;
- Specific ISA/ABI options;

Some of those are defined in terms of the arguments being passed to the compiler. Note, however, that this doesn't include all compiler options passed when parsing a module interface unit.

The format identifier, therefore, should be built only from the things that affect whether or not a particular binary module interface could be used in this translation.

Specifically, compilers may use a hashing mechanism to identify incoherency problems when several modules with transitive dependencies are used. For instance, if module "a" is built with one set of options when imported by module "b" and another set of options when imported by module "c", having modules "b" and "c" imported by a third translation unit would result in One-Definition-Rule violations.

It is important to notice that the format identifier is not the same as hashing used for those coherency checks, as those will, in general, depend on the specific options being used when parsing that particular interface unit. **It is a goal of this paper to avoid the need to invoke the compiler for each binary module interface in order to decide if it can be used or not**.

Another way to reason about this problem is that we want the format identifier to be only as specific as what would be needed to produce an equivalent binary module interface file once we used the additional instructions provided with the metadata for that module.

The implicit result of those two goals is that libraries that ship module interface units need to be limited in what kinds of instructions they are allowed to give to a system producing their own binary module interfaces for other modules. Particularly, **the metadata for a module should not specify any flag that may change the format identifier**. Resolving incoherencies on those flags will have to be managed at a higher level between the build systems and the package management systems being used, before modules are even considered[1].

Obtaining the format identifier in a given context

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, and should process them in a way that the output corresponds to where this module interface would be usable. Arguments that are irrelevant to the format of the binary module interface output should be ignored.

The build system should obtain that identifier after the toolchain configuration is set for a given target, however without including additional preprocessor definitions or include directories that are specific to that target and that will be specified in the module metadata. Again, the semantics are that by combining the baseline arguments, plus the preprocessor definitions and include directories specific to that target, you would be able to produce an equivalent binary module interface file.

Compilers that support multiple input formats

While the author of this paper is not aware of any implementation that supports a number of formats 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 input formats 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 format that will be produced, the following lines specify alternative formats that can be consumed.

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

[1] In a lot of cases, this is managed at a distribution level where a given ABI and ISA are selected as the one used in the system (for instance, RedHat Enterprise Linux switches from the pre-C++11 ABI on RHEL7 to the C++11 ABI on RHEL8), and any incoherency on those is considered a bug. In other cases, the package manager is aware of all those and the dependency solver needs to exclude any incompatible package from the resolution (the spack package manager being a notable example).