Document number	P1788R1
Date	2019-06-16
Reply-to	Olga Arkhipova olgaark@microsoft.com
Audience	EWG (Evolution), SG15 (Tooling)

Reuse of the built modules (BMI)

Olga Arkhipova
Microsoft

Is BMI just like a precompiled header?

Currently, built modules format is compiler type and version specific and the content depends on build options used for module sources compilation. This makes built modules to be quite like the precompiled headers, which have to be produced during build and can only be reused by the subsequent incremental builds where the same build tools and build options are used.

On the other hand, unlike precompiled headers, several BMIs, and in random order, can be used when building a cpp file. This opens a possibility for library vendors to ship BMIs together with module sources and implementation static libraries, so builds which use those static libraries can use BMIs as well, without rebuilding them from scratch, maximizing the compilation throughput.

Another difference from precompiled headers is that it looks possible to use the same BMI in a wider range of compilation options: due to module encapsulation it should be easier to decide if a source command line is compatible with the module command line when it is not matching it exactly.

Importance of BMI reuse

The more BMI can be reused, the better the performance of any build becomes. This is especially important for many performance-critical scenarios, such as:

- “Cold” IDE scenarios (IntelliSense, refactoring, browsing, etc.), which need to compile/parse the sources and need all used modules (and their dependencies) to be compiled as well.
- Distributed build – modules with long dependencies chains reduce parallelization. BMI reuse can greatly affect build performance.

Can reuse BMI or not?

To be able to successfully reuse an existing BMI we need to have a quick and robust way to tell if it is compatible with the given tools and build options or not. Having all build systems to figure out all compatible and incompatible build options on their own or just pass this responsibility to the user seem suboptimal. It would be more efficient if compiler vendors provide a way to check BMI compatibility with a set of build options or at least documentation for this.

When BMI cannot be reused

Often static analysis tools and IDE components use their own code parsers/compilers, which imitate "real" build compilers, but are optimized for specific work. For instance:

- Visual Studio and VS Code support not only MSVC, but also clang and gcc. VS is using EDG compiler as Intellisense engine, which currently supports MSVC, Clang and gcc modes. To be able to work for module-using code, EDG will need to be able to somehow use modules already built by MSVC, clang and gcc. Alternatively, VS should be able to rebuild them to the format EDG would understand.
  Visual Studio also uses "tag" code parser (not compiler), which will also need to "see" types defined in BMIs produces by all compilers.
- Coverity (static source code analyzer) supports many c++ compilers (and many versions of them) and uses its own parser to analyze the code. It is not feasible for it to support all BMI formats. It needs to "see" all modules' source code and their build options to be able to work.

When tools cannot use a BMI directly, they need to be able to find the all module sources (TU) and build options to extract the necessary information from the source or rebuild it using different tools.

As a build/project system might not include all module sources for all used BMIs, the source and its build options (a module "recipe") needs to be stored in the BMI itself or in a separate satellite (or easily found by other means) file.

The module "recipe" should include all necessary info to be able to rebuild a particular BMI, specifically:

- Module source file path
- The compiler "ID" (name/version)
- The command line used to produce the BMI.
- Other build options: environment variables, working directory, etc.

When a BMI is produced on the same (or identical) machine, its “recipe” can be used for rebuilds “as is” even if it contains full paths for, say, include directories, dependent modules or a module source file.

But for the BMIs built on a machine with different folders layout (as a part of a library or a distributed build) the original paths need to be modified for the rebuild to be able to find the module sources and dependencies on the current machine.

This requires some external info of a BMI/library/package installation and a way to find it. This and is being discussed in P1767 (C++ packaging).

Recommendation to library vendors:

- Always ship module sources (but can ship BMIs too).
- Provide additional info (the format is TBD, part of packaging discussions) about how to build a module from its source on the machine where this library is installed, especially if shipping BMIs.

Recommendation to compiler vendors:

- Provide a way to check if particular build options (used for a source which is importing a module) are compatible with the BMI build options.

  To be able to do this the BMI itself or another easy to find file (like, say, a file with the same name but different extension in the same directory as BMI) needs to contain enough information about the command line and macros used in the module TU.
- Store the original build “recipe” in the BMI (or a satellite file) and provide a way to retrieve it.

References

[P1103] Richard Smith. Merging Modules.
[P1441] Rene Rivera. Are modules fast?
[P1767] Richard Smith. Packaging C++ Modules.