Document number P1788R3
Date 2019-10-29
Reply-to Olga Arkhipova
olgaark@microsoft.com
Audience 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

Reusing existing BMIs as much as possible is quite important for many performance critical scenarios, especially:

- User interactive scenarios in IDEs (IntelliSense, refactoring, browsing, etc.) which compile/parse the sources as user is changing them.
- 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 which 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.

It would be very beneficial from IDE perspective to have common/standard BMI format for all compilers, but even if everybody agrees, it will not happen right away. Thus, when IDE/Code analyzers/other tools “see” a source file which is using a BMI they cannot read, they need to be able to rebuild it to a different format to process the source file. **Here we are talking only about finding and rebuilding a module interface source (to be able to read type declarations there), not rebuilding module implementation, etc**.

Module recipe: what is needed to be able to build a module

All locations below are full or relative to working directory paths on the local machine where BMI is being used.

- Module interface source file location
- Defines
- Include directories (all #inlcude files in module interface source should be found)
- Referenced BMI locations (BMIs for all imports in module interface source and all BMIs they reference (i.e. references of references) should be found)
- environment variables
- working directory
- The compiler “ID” (name/version)
- Other compiler specific command line switches used to produce the BMI

Rebuilding locally built modules

If the original BMI has been produced **locally or on a machine with identical file layout** the "module recipe" can be easily built from the command line and build environment. This will cover the following scenarios:

- Local build
- Local build of the installed source libraries
- Distributed build with identical file layout on all machines.

All this information can be added to a BMI or a BMI's satellite file in a commonly used format (say, json) during module build. For instance, MyModule.bmi.json in the same directory as MyModule.bmi.
The satellite file has the following advantages:

- It can be created not only by compiler, but by a build or packaging systems.
- It does not require a special compiler to read.

The disadvantages are:

- It is a separate file, which can be lost or get out of sync with the BMI.
- BMI most likely will contain at least some of this info (to be able to tell if the BMI is compatible with the compiled source) so having a satellite means that this info will be duplicated

Rebuilding installed modules

If the original BMI was produced **on a different machine with a different file layout** and was installed as a part of a package/library**,** the original build info is not enough to produce a "module recipe" for the local machine and additional package manifest (P1767) information is required.
To be able to create local "module recipe" satellite file for the BMIs during package install, the package manifest should contain the following info (all locations are relative paths to the package install dir):

- Package references (other packages this package depends on)
- Package include directories

For all its modules

- Module interface source file location
- Include directories (within this package, additional to "Package includes directory")

The following locations are configuration/architecture/targetsOS specific:

- Module BMI location
- Referenced BMI locations (within this package)

Combining this information with the similar information from the referenced packages and the "built-in module recipe" we should be able to create a "local module recipe" during/after package install.
All paths are full paths created from relative paths defined in the package manifest and package install dir:

- Module interface source file location
  from the package manifest

- Defines
  from original “built in module recipe”

- Include directories
  From the package and referenced packages manifests:
  <module include dirs> + <package include dirs> + <referenced packages include dirs>

- Referenced BMI locations
  From the “built-in module recipe” we can get BMI names
  Those names should be found among this package and referenced packages BMIs
  <BMIs from this package> + <BMIs from referenced packages>

- environment variables
  none (or the same as in the “built-in module recipe”?)

- working directory
  none

- The compiler “ID” (name/version)
  from original “built in module recipe”

- Other compiler specific command line switches used to produce the BMI
  from original “built in module recipe”

The system/toolsets specific search paths for includes and BMIs might be not included in the “module recipe” and can be expected to be provided by a build system.

Should BMI contain “module recipe”?

Though BMI can contain only the info from the local build which creates it, it still looks preferable to have it there so the info is not easily lost when, say, a BMI file is copied during build to some other folder. Otherwise, all build systems will have to be aware of the satellite recipe file.
If the “module recipe” satellite file is present (like in the installed packages/libraries), it should be used by a build system instead or the info in the BMI.
From this perspective, if all BMIs can always contain a “header” with a “module recipe” in the same commonly used format as the satellite file would use (say, json), it would be the most convenient.

Recommendation to library vendors:

- Always ship module interface source (but can ship BMIs too).
- Provide additional info (the format is TBD, part of packaging discussions P1767) 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:

- Include “module recipe” data (from build command line and environment) into BMI. The data should be in a commonly used format and should not require a special compiler to extract/read it.
- Provide a way to check if particular build options are compatible with the BMI build options.

Recommendation to package system vendors

- Produce “module recipe” for all BMIs in the packages during install

Recommendation to build system vendors

- Produce satellite “module recipe” files for all built BMIs before it is available in the BMI itself.

References

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