Author: Olga Arkhipova
Paper number: P2536R0
Date: 2022-02-02
Intended audience: SG15

Distributing C++ Module Libraries with dependencies json files.

This paper proposes a different approach to addressing the scenarios described in P2473R1-Distributing C++ Module Libraries, which can be described as follows:

1. Party A builds Library A, which contains C++ modules, and they want to distribute them as BMIs. The modules depend on other modules which are not included in the Library A package.
2. Party B wants to use Library A – it uses a compatible compiler and libraries, so it can use prebuilt libraries and modules.
3. Static analysis (or other tools that don't understand the BMI format used in the build) need to find the module source and the compilation options that were used to build it when working with Party B's sources/build. There are two scenarios here:
   a. Rebuild a module after main build succeeds (i.e. can use additional info produced by the Party B build)
   b. Rebuild a module before main build happens (intellisense scenarios in the Visual Studio IDE)

Let's start from #2 & 3 and determine what information is required to be able to work with pre-built modules there, and then we'll see what is required from Library A and how this information can be potentially found.

What information is required to be able to work with pre-built modules?

To be able **to use ModuleA in Party B's build (#2),** the following information is required:

- The location of ModuleA BMI, matching current build configuration
- The BMI locations of all direct and indirect ModuleA dependencies

To be able **to recompile/reparse a ModuleA interface (#3)** using a different compiler, the following information is required:

- ModuleA's interface source location on PartyB's machine
- The location of all direct and indirect include files on PartyB's machine
- The direct and indirect module dependencies (with the same info about the source, etc.)
- The machine independent (non-location) compilation options for the current build configuration.

Before we talk about how a build (#2, #3.b) can find ModuleA's BMI, let's talk about scenarios #3.a – rebuilding a module after the main build already ran and succeeded.

Rebuilding a module using dependency info produced by the main build (#3.a)

As the compiler has to "know" the locations of all BMIs it is using in order to be able to compile a TU with modules, it can easily write this information (together with other dependencies like includes) in some format, preferably easily parse-able, like json, similar to the currently produced by MSVC for /sourceDependencies.
Note that information about all source dependencies is needed to support incremental build, so it is not specific to the scenario discussed in this paper.
So for PartyB's source.cpp, which looks like this:
Sourse.cpp:

```
import ModuleA; // LibraryA module, depends on ModuleA:PartitionA and ModuleC from
another library
import ModuleB; // Party B's module
…
```

The source dependencies json will look like this:

```
{
  "Version": "1.1",
  "Data": {
    "Source": "C:\\PartyB\\sources\\source.cpp",
    "Includes" = […],
    "ImportedModules": [
      {
        "Name": "ModuleA",
        "BMI": "C:/Path/To/LibraryA/x64/SpecialConfig/ModuleA.ixx.ifc"
      },
      {
        "Name": "ModuleA:PartitionA",
        "BMI": " C:/Path/To/LibraryA/x64/SpecialConfig/ModuleA-PartitionA.ixx.ifc"
      },
      {
        "Name": "ModuleC",
        "BMI": " C:/Path/To/LibraryC/x64/Release/ModuleC.ixx.ifc"
      },
      {
        "Name": "ModuleB",
        "BMI": "C:/Path/To/Outputs/x64/MyConfig/ModuleB.ixx.ifc"
      },
    ],
    "ImportedHeaderUnits": []
  }
```

}

As ModuleB.ixx.ifc was built as part of PartyB's build, the source-dependencies file should be found in the same directory as the ifc and use the same file name. So C:/Path/To/Outputs/x64/MyConfig/ModuleB.ixx.ifc.json exists and contains the following info:

```
{
  "Version": "1.1",
  "Data": {
    "Source": "C:\\PartyB\\sources\\ModuleB.ixx",
    "ProvidedModule": "ModuleB",
    "Includes" = […],
    "ImportedModules": […]
    "ImportedHeaderUnits": […]
  }
}
```

If we can find similar json files near C:/Path/To/LibraryA/x64/SpecialConfig/ModuleA.ixx.ifc and other libraries' ifcs (pointing to the source locations on PartyB's machine), **we have the sources and module names for all modules that were used in the compilation** of source.cpp. We also know the exact locations of all dependencies.
So, to be able **to rebuild any module (including ModuleA) after the main build has succeeded (#3.a)**, besides the source-dependency.json files we just need the command line or possibly just the "non-location" options (#defines, etc) as we already know the exact locations of all dependencies.

LibraryA Package with dependencies json (#1)

The package structure for a library might be arbitrary. The only requirements are:

1. Package should contain module sources
2. Package should contain dependency json files (<bmi name>.d.json) near each BMI.

So for our LibraryA, which contains ModuleA, which imports ModuleA:PartitionA as well as ModuleC (from another library), the package structure might look like

```
LibraryA
  Sources
    ModuleA.ixx
    ModuleA-PartitionA.ixx
  x64
    SpecialConfig
      ModuleA.ifc
      ModuleA.ifc.d.json
      ModuleA-PartitionA.ifc
      ModuleA-PartitionA.ifc.d.json
      LibraryA.lib
  x64
```

Release
…

Where x64/SpecialConfig/ModuleA.ifc.d.json might look like:

```
{
  "Version": "1.1",
  "Data": {
    "Source": "../../sources/ModuleA.ixx", ← relative path to module source from this file location
    "ProvidedModule": "ModuleA",
    “Options”=[…], ← configuration specific, location agnostic  compilation options.
    "ImportedModules": [
      {
        "Name": "ModuleA:PartitionA",
        "BMI": ". /ModuleA-PartitionA.ixx.ifc" ← this package dependency
      }
      {
        "Name": "ModuleC",
        "BMI": ”” ← the dependency outside of this package
      }
    ]
  }
}
```

If Library A’s build has produced the dependencies json, it can be easily and programmatically transformed into the file shown above.

Using ModuleA in Party B’s build (#2) and rebuilding it without using build info (#3.b)

To be able to use ModuleA in Party B’s build its BMI location (matching a particular build configuration, like “<root>/LibraryA/x64/SpecialConfig”) must be “externally” given to the build. This can be done either by the user or by the package manager, or by putting the BMI in the location already used by the build. The BMI locations can be specified either as a directory on the module search path, or as an explicit location in the form of [module name]=[BMI full path].

When a module search path is used for compilation, the BMI naming should match the compiler expectations of how module name is “encoded” in the BMI name. For instance, ‘:’ (used in module partitions) is not allowed symbol to use as in the file name on Windows, so it must be either substituted by a different symbol (say, ‘-‘ or ‘+’ which is valid in files, but invalid in c++ types) or a convention in the folder structure should be used.

This might work for main Party B’s build (as it is using the same or compatible build tools as Library A’s build), but might not work for the scenarios like #3.b – rebuilding a module using a different compiler and maybe even on a different OS.

Even if all compilers agree on some BMI name convention (which will have to work for all possible file systems), requiring to "encode" module name in the BMI file name requires the user to update the BMI name/location build options each time the module name is changed in the sources. It puts the burden on the user and also adds complexity to the IDE's refactoring operations.

But with .d.json near all BMI files, even if only a module search path (and not explicit BMI locations) was defined for the main build, we can find all .d.json files on this path. Thus, we can read all of them and create a "prebuilt modules map" in the form of [module name]=[BMI full path] (or even [module name]={all info from d.json} if we want). This information is sufficient to resolve all module dependencies in the PartyB's sources, as well as for all library modules if all required libraries are present on the machine. It also provides all information that is needed to rebuild a module with a different compiler without requiring any "encoding" of the module name in the BMI or module interface file names.

Proposal Summary

The list of all files used in a TU compilation is required for incremental build support. Most of the compilers are currently capable of producing the dependency files with #include info and (optionally) source location.

Module's BMIs are dependencies similar to #include files. The compiler has to "know" their exact locations to be able to compile a TU and should be able to list them as dependencies.

I propose the compilers should be able to produce dependency info in JSON format (so it is easily parse-able and modify-able) with additional information about:

- module name (when compiling module interface sources)
- compilation options (possibly including the compiler's internal defines and options)
- module dependencies

The module dependencies information will be somewhat similar to the one proposed for source scanning in P1689R4 (i.e. produced before source compilation), but with the addition of the BMI and source locations.

I also propose that all libraries that ship prebuilt modules, include:

- Module interface sources.
- Dependency json files (.d.json) near each BMI.

As BMIs are configuration specific, the dependencies and compilation options in the .d.json files, are also configuration specific. There is no ambiguity about which configuration to select during build.
As .d.json files will contain full or relative paths of the module interface source, as well as all its dependency names (and possibly locations), there is no need to "encode" module name in the file system.

If we have the dependency json file described above near each BMI, we have enough information to support all module's rebuild scenarios.

In this paper, I used a json format similar to the one currently produced by MSVC for /sourceDependencies, but the actual field names and additional content is to-be-agreed-upon.

References

P2473R1 Distributing C++ Module Libraries
P1689R4 Format for describing dependencies of source files
cl.exe /sourceDependencies format