Building C++ 20 Modules in Visual Studio

This document describes how C++ 20 named modules are produced and consumed during build of VC projects

Short MSBuild reference

Project – a file containing info about sources and various build options used by build logic to construct command lines for various tools. The build logic provides defaults for most of the options, but the user can change any option. This includes the locations and file names of all outputs, such as obj, ifc, lib, dll, exe, etc.

Configuration – 'Configuration Name' – 'Platform Name' pair. 'Configuration Name' is an arbitrary user defined string. 'Platform Name' – an arbitrary string defined by VS or its extensions which provide build logic for a particular target OS/platform/technology. An example would be:

> Platform=x64 (Windows Desktop – x64)
> Platform=x64_GDK (Xbox – x64)

A project can use configuration conditions for all properties it defines.

Project Configuration – a project that has been evaluated with particular configuration settings. The build always happens for a particular project configuration and cannot "see" other project configurations.

P2P – project to project reference, defined in the referencing project, used by msbuild for project build order. Referencing project can get information from referenced projects (or rather their configurations used in the build)

For instance, Project A is referencing Project B and Project C.
The build of Project A will start only after Project B and Project C builds are completed. Projects B and C are not dependent on each other and can be built in parallel.
The Project A's build can get information from the Project B and Project C, such as locations of BMIs and libs produced by them.

Modules production and consumption within one project

Suppose we have the following sources in one project:

MyStaticLibrary1 project:

> ModuleA.ixx:
>
> ```
> export ModuleA;
> ```

```
import ModuleB;
```

ModuleB.ixx:

```
export ModuleB;
```

source.cpp

```
import ModuleA;
…
```

User identifies module interfaces by using .ixx file extension or by setting “CompileAs” option to “Module Interface”.

The build **automatically** does the following:

1. “Scans” all module interface files and (optionally) other sources - calls cl.exe with special switches to produce module dependencies json similar to described in P1689R4. The currently used switch and json format is described here: /sourceDependencies:directives (List module and header unit dependencies) | Microsoft Docs

   ModuleA.ixx -> ModuleA.ixx.module.json
   ModuleB.ixx -> ModuleB.ixx.module.json

   ModuleA.ixx.module.json looks like

   ```
   {
      "Version":"1.1",
      "Data":{
         "Source":"Path\\To\\ModuleA.ixx",
         "ProvidedModule":"ModuleA",
         "ImportedModules":[
            "ModuleB"
         ]
      }
   }
   ```

2. Reads all “scan” json files and creates module dependency graph:

   ModuleA node -> ModuleB node

   A node in the graph contains module name as well as “base” command line (without module references) to build an ifc.

   For instance, for ModuleA the “base” command line for Debug x64 configuration would look like

   cl.exe /c /D Debug /interface /ifcOutput x64\Debug\ModuleA.ixx.ifc /Fo x64\Debug\ ModuleA.ixx.obj ModuleA.ixx

3. Builds modules (and other scanned files) in the dependency order and adds necessary module **references** to their command lines in [ModuleName]=[ifc location] form.
   It also tells the compiler to produce *.d.json* for all built .ifcs (with the same file name and location). The d.json format is described here: [/sourceDependencies (Report source-level dependencies) | Microsoft Docs]. We'll discuss the usage of .d.json file later in "Consumption of modules build by referenced projects".

   ModuleB:
   ```
   cl.exe /c /interface
   /D Debug
   /ifcOutput x64\Debug\ModuleB.ixx.ifc
   /sourceDependenices x64\Debug\ModuleB.ixx.ifc.d.json
   /Fo x64\Debug\ModuleB.ixx.obj
   ModuleB.ixx
   ```

   ModuleA:
   ```
   cl.exe /c /interface
   /D Debug
   /ifcOutput x64\Debug\ModuleA.ixx.ifc
   /sourceDependenices x64\Debug\ModuleA.ixx.ifc.d.json
   /Fo x64\Debug\ModuleA.ixx.obj
   /reference ModuleB=x64\Debug\ModuleB.ixx.ifc
   ModuleA.ixx
   ```

4. Builds the rest of the sources passing all built modules as /reference
   source.cpp:
   ```
   cl.exe /c
   /D Debug
   /Fo x64\Debug\source.obj
   /reference ModuleB=x64\Debug\ModuleB.ixx.ifc
   /reference ModuleA=x64\Debug\ModuleA.ixx.ifc
   source.cpp, …
   ```

5. Libs all objs:
   x64\Debug\MyStaticLibrary1.lib:
   ```
   x64\Debug\ModuleB.ixx.obj
   x64\Debug\ModuleA.ixx.obj
   x64\Debug\source.obj
   …
   ```

Consumption of modules build by referenced projects

Suppose we have another static library project which wants to use ModuleA as well as an exe project which wants to use that static library:

MyApp references **MyStaticLibrary2** which references **MyStaticLibrary1**

MyStaticLibrary2 project

references **MyStaticLibrary1** project and contains

ModuleC.ixx

export ModuleC;

import ModuleA;

…

As MyStaticLibrary project references MyStaticLibrary1 project, its build happens after MyStaticLibrary1 build is complete and does the following:

1. Gets all ifc locations produced by MyStaticLibrary1 though P2P.
   Full\Path\To\x64\Debug\ModuleA.ixx.ifc
   Full\Path\To\x64\Debug\ModuleB.ixx.ifc
2. Reads d.json files for all ifcs and creates “module map” for them (currently just in memory)
   ModuleA= Full\Path\To\x64\Debug\ModuleA.ixx.ifc
   ModuleB= Full\Path\To\x64\Debug\ModuleB.ixx.ifc
3. Scans ModuleC.ixx -> ModuleC.ixx.module.json
4. Reads “scan” json(s) and build the dependency graph:

   ModuleC node ->ModuleA

5. Builds modules in the dependency order adding all “module map” modules as references (in addition to module dependencies built by this project, but we have only one module here).
   Module A is not a part of this project and its ifc already exists in “module map” – no build is needed.
   ModuleC:
   cl.exe /c /interface
   /D Debug
   /ifcOutput x64\Debug\ModuleC.ixx.ifc
   /sourceDependenices x64\Debug\ModuleC.ixx.ifc.d.json
   /Fo x64\Debug\ModuleC.ixx.obj
   /reference ModuleB=Full\Path\To\x64\Debug\ModuleB.ixx.ifc
   /reference ModuleA=Full\Path\To\x64\Debug\ModuleA.ixx.ifc
   ModuleC.ixx

   x64\Debug\ModuleC.ixx.ifc.d.json will contain the exported module name, as well as all direct and indirect modules that were used in its compilation:

```
{
    "Version": "1.1",
    "Data": {
        "Source": "Full\\Path\\To\\ModuleC.ixx",
        "ProvidedModule": "ModuleC",
        "ImportedModules": [
            {
                "Name": "ModuleA",
                "BMI": "Full\\Path\\To\\x64\\Debug\\ModuleA.ixx.ifc"
```

```
            },
            {
                "Name": "ModuleB",
                "BMI": "
Full\\Path\\To\\x64\\Debug\\ModuleB.ixx.ifc"
            },
        ],
        …
    }
}
```

6. Builds the rest of the source with additional module references for ModuleA, ModuleB and ModuleC
7. Lib the objs.

MyApp project

References **MyStaticLibrary2** and contains

main.cpp

import ModuleC;

…

As MyApp project references MyStaticLibrary2 project, its build happens after MyStaticLibrary2 (and MyStaticlibrary1) build is complete.

The MyApp build does the following:

1. Gets ifc locations produced by direct project references (i.e. just MyStaticLibrary2) though P2P.

   Full\Path\To\x64\Debug\ModuleC.ixx.ifc

2. Reads d.json files for all ifcs (**Full\Path\To\x64\Debug\ModuleC.ixx.ifc.d.json** in our case) and creates "module map" for them. Note that because the .d.json file contains all direct and indirect module dependencies for the corresponding .ifc, the build does not need to traverse the indirect project dependencies, i.e. get info from MyStaticLibrary1 project. The "module map" will still contain all necessary modules:

   ModuleA= Full\Path\To\x64\Debug\ModuleA.ixx.ifc
   ModuleB= Full\Path\To\x64\Debug\ModuleB.ixx.ifc
   ModuleC= Full\Path\To\x64\Debug\ModuleC.ixx.ifc

3. Compiles sources adding all "module map" modules as references:

   main.cpp:

   cl.exe /c
   /D Debug
   /Fo x64\Debug\main.obj
   /reference ModuleA=Full\Path\To\x64\Debug\ModuleA.ixx.ifc
   /reference ModuleB=Full\Path\To\x64\Debug\ModuleB.ixx.ifc
   /reference ModuleC=Full\Path\To\x64\Debug\ModuleC.ixx.ifc
   main.cpp, …

4. Links the exe. By default, the build will automatically include all direct and indirect static libraries built by reference projects thus getting all used module objs.

   Link
   x64\Debug\main.obj
   Full\Path\To\x64\Debug\MyStaticLibrary1.lib
   Full\Path\To\x64\Debug\MyStaticLibrary2.lib
   …

To be able to define a module in a dll and use it in other binaries all module type/methods should be dll exported. The dll exports are good for c style APIs, but not fully supported for c++ types, which makes this scenario quite limited.

Note that all references and outputs (ifc, lib, .module.json and .d.json files) are configuration specific. The build for a different configuration can produce drastically different set of these outputs as module content and dependencies can be different.

Building modules for C++ intellisense

The user expectation in VS is that c++ intellisense should be available regardless of whether the code has been built before or not.
To fulfill this expectation, when Solution/codebase is opened in the VS the following actions are performed:

1. The designtime components get information from the project system about sources and its compilation options (“base” command lines constructed from project properties)
2. All sources that are identified as module interfaces are “scanned” – cl.exe is called to produce dependency json files for the currently active configuration.
3. The scan json files are read and the full modules dependency graph (which allows ambiguities when several modules with the same name exist in the codebase) is created. The dependency graph node contains pointers to the module interface source and its command line
4. When a C++ file that imports a module is opened in the editor, the correspondent module is found in the dependency graph and the ifc is built for it as well as all its dependences in the dependency order.

Currently, for a number of reasons which might or might not go away when ifc format is stabilized, C++ intellisense does not use modules created during build even when they exist – it always builds them on its own (and to different locations than the “read” build).
So with the externally supplied modules it is critical that we are able to find module source as well as compilation options, module dependencies & includes and be able to re-compile them.