Document Number: P3286R0
Date: 2024-06-15
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Module Metadata Format for Distribution with Pre-Built Libraries

Abstract

This paper specifies the format for pre-built libraries to advertise the metadata about the C++ Modules being provided, with the information required to perform the translation of the Importable Units into Built Module Interface files. The format complements the work from P2701R0[1] and P2577R2[2] with the concrete format to be used when advertising modules in the distribution of pre-built libraries. This will complement the Modules Ecosystem TR with the format to be used by tooling implementers.

1. Context

This paper is a continuation of the work from P2701R0 and P2577R2, this paper is just specifying the format for the metadata file. More specifically, the need for this format became concrete as the libc++ standard library is now shipping experimental support for the std and std.compat modules, and with that build systems need to be able to produce the Built Module Interface (BMI) files for the modules provided by the standard library.

The discussion happened initially in the libc++ pull request[3], which then spawned a thread in the SG15 mailing list, where an initial format was proposed[4]. This paper, therefore, formalizes that proposal.

It should be noted that this format will also be used for the distribution of pre-built libraries in general, not just the standard library. Future integration with the evolving package management ecosystem is something that we should look out for.

2. Assumptions

The same assumptions documented in P1689R5[5] also apply to the format being proposed here.

---

[1] RUOSO, Daniel. Translating Linker Input Files to Module Metadata Files, 2022. https://wg21.link/P2701R0
[2] RUOSO, Daniel. C++ Modules Discovery in Prebuilt Library Releases, 2022. https://wg21.link/P2577R2
[3] https://github.com/llvm/llvm-project/pull/75741
[4] https://lists.isocpp.org/sg15/2023/12/2240.php
[5] BOECKEL, Ben. KING, Brad. Format for describing dependencies of source files. 2022. https://wg21.link/p1689r5

3. Implementation Experience

This has been released as an experimental feature in libc++. There has been early experimentation in CMake to add the ability to generate the BMI for those modules. Experimental `import std` support landed for clang 18.0.2+ in CMake for the upcoming 3.30 release using this format.

4. Requirements

The requirements for this format were mostly laid out in P2577R2 and P2701R0. Further requirements were gathered in discussions on how libc++ would describe the standard modules for build systems. For simplicity, those requirements will be summarized here:

- A build system should have a way to identify which modules are provided by a pre-built library.
- Locating the metadata file:
    - For the Standard Library:
        - The build system should be able to query the toolchain (either the compiler or relevant packaging tools) for the location of that metadata file.
    - Other Libraries:
        - In the absence of stronger package management, in environments where that is viable, the build system may infer the location of the metadata based on link-line fragments (P2701R0).
        - If package management is present, that information can be gathered in implementation-defined ways.
    - The path to the metadata file should be related to the input files that are given to the linker. The expectation is that different builds of the library may have different metadata files.
- The contents of the metadata must include:
    - The “logical name”[6] name of the importable unit being provided.
    - The path to the primary source file for the importable unit.
    - Any additional include paths required to translate that particular importable unit.
    - Any compiler definitions required to translate that particular importable unit.
    - Whether the module is a module provided by the standard library or not, since those module names are reserved.
- The contents of the metadata may include:
    - The “logical name” of importable units that are a dependency of that translation unit.
    - Vendor-specific attributes

[6] The concept of “logical name” is specified in BOECKEL, Ben. KING, Brad. Format for describing dependencies of source files. 2022. https://wg21.link/p1689r5

5. Format

The file will be encoded in JSON[7] and the data model is described in this paper as a JSON Schema[8]. As it happens for P1689R5, the format will also require that file paths must be constrained to valid utf-8 sequences[9].

5.1. Schema

For the information provided by the format, the following JSON Schema[10] may be used.

```javascript
{
    "$schema": "",
    "$id": "http://example.com/root.json",
    "type": "object",
    "title": "WG21 SG15 C++ Module Metatadata Format",
    "definitions": {
      "vendor": {
        "$id": "#vendor",
        "type": "object",
        "description": "vendor-specific information. The key is the name of the
vendor and the value is implementation defined.",
        "patternProperties": {
            "^.+$": {
                "type": "object",
                "description": "implementation-defined data for the vendor
using that identifier"
            }
        }
      },
      "datablock": {
        "$id": "#datablock",
        "type": "object",
        "description": "A filepath",
        "minLength": 1
      },
```

[7] The JSON Data Interchange Syntax. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.
[8] Austin Wright and Henry Andrews. JSON Schema: A Media Type for Describing JSON Documents. https://tools.ietf.org/html/draft-handrews-json-schema-01.
[9] In the 2024 Tokyo meeting, there was general consensus that we need to work with the Unicode Study Group to figure out the appropriate mechanisms to refer to files where the names are representable as utf-8 sequences.
[10] Austin Wright and Henry Andrews. JSON Schema: A Media Type for Describing JSON Documents. https://tools.ietf.org/html/draft-handrews-json-schema-01.

```
      "preprocessor-define": {
        "$ird": "#preprocessor-define",
        "type": "object",
        "description": "a definition to be set in the preprocessor",
        "required": [
          "name"
        ],
        "properties": {
          "name": {
            "type": "string",
            "description": "the name of the token to be defined in the
preprocessor"
          },
          "value": {
            "type": "string",
            "description": "the value to be set. If not present it is
equivalent to -DFOO in gcc and clang",
            "default": null
          },
          "undef": {
            "type": "boolean",
            "default": false,
            "description": "If set, instructs the preprocessor to make that
value undefined. Equivalent to -UFOO in gcc and clang. Incompatible with using
a value at the same time."
          },
          "vendor": {
            "$ref": "#/definitions/vendor"
          }
        }
      },
      "local-arguments": {
        "$id": "#local-arguments",
        "type": "object",
        "description": "Local arguments to be used when translating the module
unit",
        "properties": {
          "include-directories": {
            "type": "array",
            "description": "An array of paths that need to be appended to the
compilation include search path, same semantics as appending -I in gcc and
clang.",
            "items": {
              "$ref": "#/definitions/datablock"
```

```
            }
          },
          "system-include-directories": {
            "type": "array",
            "description": "An array of paths that need to be appended to the
compilation include path as system locations, same semantics as appending
-isystem in gcc and clang.",
            "items": {
              "$ref": "#/definitions/datablock"
            }
          },
          "definitions": {
            "type": "array",
            "description": "An array of definitions for the preprocessor.",
            "items": {
              "$ref": "#/definitions/preprocessor-define"
            }
          },
          "vendor": {
            "$ref": "#/definitions/vendor"
          }
        }
      },
      "module": {
        "$id": "#module",
        "type": "object",
        "description": "Metadata about a module provided by the library",
        "required": [
          "logical-name",
          "source-path"
        ],
        "properties": {
          "logical-name": {
            "$ref": "#/definitions/datablock"
          },
          "is-interface": {
            "type": "boolean",
            "description": "True if this is an interface unit (primary or
interface partition), false if it's an internal partition.",
            "default": true
          },
          "source-path": {
            "$ref": "#/definitions/datablock"
          },
```

```
          "is-std-library": {
            "type": "boolean",
            "description": "Whether this module is part of the standard
library, and therefore allowed to use the reserved names",
            "default": false
          },
          "local-arguments": {
            "$ref": "#/definitions/local-arguments",
            "default": {}
          },
          "vendor": {
            "$ref": "#/definitions/vendor"
          }
        }
      }
    },
    "required": [
      "version"
    ],
    "properties": {
      "version": {
        "$id": "#version",
        "type": "integer",
        "description": "The version of the output specification"
      },
      "revision": {
        "$id": "#revision",
        "type": "integer",
        "description": "The revision of the output specification",
        "default": 0
      },
      "modules": {
        "$id": "#rules",
        "type": "array",
        "title": "rules",
        "default": [],
        "items": {
          "$ref": "#/definitions/module"
        }
      }
    }
  }
```

5.2. Examples

The following example represents what could be used for declaring modules that are part of the standard library.

```JavaScript
{
  "version": 1,
  "revision": 1,
  "modules": [
    {
      "logical-name": "std",
      "source-path": "modules/std.cppm",
      "is-std-library": true
    },
    {
      "logical-name": "std.compat",
      "source-path": "modules/std.compat.cppm"
      "is-std-library": true
    },
    {
      "logical-name": "std:someinterfacepartition",
      "source-path": "modules/std-someinterfacepartition.cppm"
      "is-std-library": true
    }
  ]
}
```

The following example represents modules provided by an arbitrary other library with additional preprocessor requirements.

```JavaScript
{
  "version": 1,
  "revision": 1,
  "modules": [
    {
      "logical-name": "foo",
      "source-path": "modules/foo.cppm",
      "local-arguments": {
        "definitions": [
          {
            "name": "FOO_CONFIG_VALUE",
            "value": 42
```

```
            }
          ]
        }
      }
    ]
  }
```

5.3. Resolving relative paths

The build system will get the path to this file by either asking the toolchain or an underlying package manager for it. The path provided to this file should be used as-is, without any additional symbolic link resolution.

Any file or directory referenced by the metadata file in relative form should be considered relative to the path provided, any relative path in the metadata file will be resolved based on the path provided by the toolchain or package manager.

6. Versioning

This format follows the same model defined in P1689R5.