Document Number: P1884r0
Date: 2020-01-07
Audience: SC22/WG21 SG2
Reply to: Nathan Sidwell nathan@acm.org

Private Module Fragment: An Inconsistent Boundary

Nathan Sidwell

The Private Module Fragment (PMF) was described as a simple mechanism to provide single file modules, but restricted definition reachability. Implementation experience is not known. There are different requirements for the placement of inline function definitions, depending their linkage. These differences appear to either be unintentional, or serve no benefit to programmers, and incur implementation difficulty and/or user surprise.

1 Background

The Private Module Fragment was introduced by P1242r0 'Single-file modules with the Atom semantic properties rule'. It provides a mechanism by which a single source file may continue to hide completeness of types (for instance), from importers of the module. Prior to the ATOM semantics rule, this could be achieved via not decorating definitions with '`export`', but that was shown to be brittle, particularly when code refactoring.

The syntax breaks a module's primary interface into three sections:

```
module;  // Global Module Fragment (optional)
#include <xtensor.hpp>

export module foo; // module interface
export struct incomplete;

module :private; // Private Module Partition (optional)
struct incomplete {…};
```

The contents are described in P1242r1 as a module interface partition followed by an implementation partition. Its discussion of importation uses earlier terminology where, what is now simplify referred to as the primary module interface, is termed the primary module interface partition:

> When the primary module interface partition is imported (either via import into a module implementation partition, or via implicit import into a module implementation unit), its corresponding inline module implementation partition, if present, is also imported.

There is no discussion of importing into user code, but that is also intended. This paragraph is affected by subsequent EWG discussion below.

1.1 NB Comments

Two NB comments concerning the PMF were filed. The first a call for an exemplar:

GB079

> The private-module-fragment is mentioned many times in the standard but its usage its obscure; as is in the papers. Nobody in the BSI knows its intended usage.
> Please provide an example.
>
> Proposed change:
> Example added

The second, filed by the author, was rather more invasive

US036

> The semantics of the private module fragment (PMF) are underspecified. It appears well formed to declare an entity in the interface purview and define it in the PMF. How does this interact with inlining, instantiation and internal linkage definitions? The intent of the PMF is as-if it is a separate module implementation unit, but we do not define the boundary between the interface purview and the PMF as a translation unit boundary. Implementations may defer instantiation (of function definitions) to the end of translation. Similarly internal linkage and inline functions (that are ODR-used) must be defined at the end of translation. Perhaps emission of Compiled Module Interface (CMI) should be deferred to the end of translation – and not at the beginning of the PMF. It will therefore observe entities declared or defined in the PMF. This may create implementation difficulty to preserve the semantics of the PMF being as-if a separate translation unit. Alternatively, the boundary could be specified as a new kind of ‘end of translation’ point. This seems a dramatic change.
>
> Proposed change:
> Remove the private module fragment. It’s semantics are too ill-defined, and I do not believe there is sufficient experience to define them at this point.

SG2 discussed these comments at the Belfast’19 meeting.

- It was agreed to address GB079 with an example & cross references in Clause 10. (https://github.com/cplusplus/nbballot/issues/78)

Prior to the Belfast’19 meeting, the author discovered the WP did specify many of the issues raised by US036. However a reader would need to look at approximately half a dozen otherwise-unconnected places in the WP to discover this.

- It was agreed to reject US036, but extend the response for GB079 to include a paragraph referencing the places the PMF affects semantics.
  (https://github.com/cplusplus/nbballot/issues/35).

Core wording was provided in P1971R0. That wording was discovered to contain an error. The wording was corrected editorially and applied to the WP shortly after the Belfast'19 meeting.

1.2 Inconsistency

The core of the issue is the following wording from the WP (emphasis added):

> An **exported** inline function or variable shall be defined in the translation unit containing its exported declaration, outside the *private-module-fragment* (if any). [dcl.inline]/7

In particular, this restriction does not apply to module-linkage inline functions or variables.

A frequent error this author makes, when reasoning about the PMF is considering other module units of the module. This is already addressed by:

> A *private-module-fragment* shall appear only in a primary module interface unit (10.1). A module unit with a private-module-fragment shall be the only module unit of its module; no diagnostic is required. [module.frag.private]/1

That was a change from P1242 made at the Kona'19 meeting, discussed below. The difficulties discussed here do not concern such disallowed module units.

1.3 Inline

C++17 requires inline functions and variables to be defined in every TU that they are used:

> An inline function or variable shall be defined in every translation unit in which it is odr-used outside of a discarded statement. N4700 [basic.def.odr]/4

The WP adjusts this to:

> A definition of an inline function or variable shall be reachable in every translation unit in which it is odr-used outside of a discarded statement. [basic.def.odr]/10

The inline specifier description, repeats this, with the extra specificity that it is reachability at the end of that translation unit:

> If an inline function or variable is odr-used in a translation unit, a definition of it shall be reachable from the end of that translation unit, … [dcl.inline]/6

The new property of reachability is defined as:

> A declaration *D* is *reachable* if, for any point *P* in the instantiation context (10.6),

- *D* appears prior to *P* in the same translation unit, or

- *D* is not discarded (10.4), appears in a translation unit that is reachable from *P*, and either does not appear within a *private-module-fragment* or appears in a *private-module-fragment* of the module containing *P*. [module.reach]/3

(Outside of instantiations, 'instantiation context' [module.context] is the source code point.)

Part of the inline specification is:

> The inline specifier indicates to the implementation that inline substitution of the function body at the point of call is to be preferred to the usual function call mechanism. [dcl.inline]/2

The requirements on inline function definitions are such that inlining can be achieved with translation-unit-at-a-time compilation – whole-program or link-time compilation is not necessary.[1] The inline function definition can appear after the call, so sequential function-at-a-time compilation is insufficient.[2]

This paper is predicated on retaining the compilation model where inlining occurs within a single TU to create object files that are combined by a simple linker.

1.4 Origins of PMF Restrictions

A set of PMF requirements make the point immediately before the PMF a kind of end of translation. For instance:

> During the implicit instantiation of a template whose point of instantiation is specified as that of an enclosing specialization (13.7.4.1), the instantiation context is the union of the instantiation context of the enclosing specialization and, if the template is defined in a module interface unit of a module *M* and the point of instantiation is not in a module interface unit of *M*, the point at the end of the *top-level-declaration-seq* of the primary module interface unit of *M* (prior to the *private-module-fragment*, if any). [module.context]/3

This is driven by implementability concerns. Namely whether implementations needed defer writing out a Compiler Module Interface[3] (CMI) to the end of translation, after the PMF had been parsed.

At the Kona'19 meeting, this was discussed in both CWG (http://wiki.edg.com/bin/view/Wg21kona2019/CoreWorkingGroup, as part of D1103R3 Merging Modules) and EWG (http://wiki.edg.com/bin/view/Wg21kona2019/P1242R0-EWG) sessions.

1 The standard does not specify a unit-at-a-time compilation mode, but this is a common implementation technique.
2 The author recalls that GCC required inline function definitions before the point of the call until sometime in the 2000's, when it moved to a whole-TU compilation model.
3 The standard does not specify a CMI, but this is a common implementation technique. This paper presumes such a technique.

The EWG discussion is not captured. The author's recollection is that the compelling example was that of an exported class declaration with a PMF-located definition:

```
export module M;
export class X;
…
module :private;
class X { … };
…
```

If there are no other module units, no instantiations in the interface purview could depend on the completeness of X[4], then nothing in the PMF needs serializing. Therefore the CMI could be written before the PMF was parsed, and that satisfied the implementability concerns. Unfortunately inline function definitions were not considered.

The poll results are:.

> Any module which contains "module :private"; is a single file module:

SF	F	N	A	SA
4	9	5	0	0

> Motion passes

> The private module fragment behaves as a module implementation unit (not a module partition)

SF	F	N	A	SA
3	6	6	0	1

> Motion doesn't pass

The first poll's intent was to reduce the problem space. If there are no other module units, the interaction between the primary interface (& its PMF) and any other units is mooted. The intent of P1242 is for single-file modules, so permitting additional module units would seem to be at cross-purposes.

The reasons for the second poll not passing, despite the seeming favourability, are not mentioned.

It is noted that terminology was not changed to refer to it as a *private-module-partition*, in spite of the deliberate choosing of a partition-like syntax with a leading '`:`'.

4 It would be an error for the same instantiation to appear both before and after X's definition.

1.5 Implementations

The author knows of no implementations supporting the PMF. GCC, Clang, MSVC++ and EDG all plan to support it at a future date.[5] As such we should proceed with caution. Restrictions that are put in place now could be relaxed later.

2 Discussion

Here is an example module interface:

```
export module M;

inline void frob_m (int);
export inline void frob_e (int);

export inline void widget (int x) // #1
{
   frob_m (x);  // #2
   frob_e (x);  // #3
}

export template<typename T> void bodgit (T x) // #4
{
  frob_m (x); // #5
  frob_e (x); // #6
}

module :private;

void frob_m (int) {}
void frob_e (int) {}
```

The WP makes this ill formed because '`from_e`' is an exported inline function who's definition is not before the PMF ([dcl.inline]/7). Disregard that ill-formedness when considering unrelated aspects of the example.

The templated calls #5 & #6 are dependent, and therefore resolved at instantiation time. Other overloads might be visible at that point, and the ones declared here not selected by overload resolution. If the example is modified slightly, perhaps the two functions here are not found by phase 1 lookup at all, but could be reached by ADL at phase 2.

5 Conversations with the respective compiler implementors.

2.1 Reachability & Inline

A careless reading of [basic.def.odr]/10 may lead one to believe an inline function definition must appear before the non-discarded ODR-use – as [module.reach]/2 specifies reachable definitions within a TU as being those before the instantiation point. But [basic.def.odr]/10 merely states the definition must be reachable in the TU. [dcl.inline]/6 further states the definition must be reachable from at the end of translation.

As mentioned above, the earlier C++17 rules are intended to allow inlining with per-TU compilation. The author believes the reformulation via reachability, to accommodate modules, should also have the same goal. To do otherwise is a significant change in the compilation model, which we should consider carefully as a separate issue.

2.2 Linkage Inconsistency

As mentioned above, it is only exported inline functions that must be defined before the PMF. This appears to be an effort to permit the compilation model described in Section 1.3, where the CMI can be written before the PMF has been parsed.

If P1242 envisioned writing the CMI at the end of the TU (and the PMF be an imported implementation partition), there would be no requirement that the exported inline function body be provided before the PMF. For instance, as separate source files, the following is (almost) permitted:

```
export module foo:interface;
inline void frob_m ();
export inline void frob_e (); // #1

module foo:implementation;
import :interface;
void frob_m () {}
void frob_e () {}

module foo;
export import :interface;
import :implementation;
```

Because `frob_e` is declared inline at #1, but no definition is provided, that TU is ill-formed. Such a restriction would seem useful in the primary interface unit – because nothing else could provide a definition that would be reachable from user code. But it seems overly restrictive in a module partition – another partition can easily provide the definition. Provided it was imported (directly or indirectly) into any ODR-using source all would be well.

While P1242 (also) disallows the equivalent for `frob_e` as a single-file module, it does permit it for `frob_m`. However, the implementation constraints of inlining `frob_e` and `frob_m` are the same.

Their definitions need to be available to the TU containing the ODR-use. As the example at the start of Section 2 shows, both functions can be called from code emitted by user source (calling `widget` or instantiating `bodgit`).

2.3 PMF Restrictions

The Kona'19 EWG discussion concludes that the PMF should be considered a module implementation partition. There are at least two aspects about this that have somewhat surprising semantics:

1. The PMF is imported into the interface.
2. The PMF sees the of the interface purview.

Named partitions may be imported into the primary module interface, module implementation units or other implementation or interface partitions. Such importation must happen in the importation sequence immediately following the module declaration. This makes the partition contents reachable from the primary's purview. That is not possible with a PMF, as it succeeds the interface purview.

For implementations, each partition produces its own CMI, which is read into a TU when the partition is imported. If the partition is imported into the primary interface, the partition's CMI will be transitively imported into any TU that imports the primary.[6] If the primary's CMI is written before parsing the PMF, this is not possible.

Thus, if it is to be considered imported, it is an exceptional import.

Entities declared in the primary interface are not reachable from named partitions as named partitions do not import the primary interface. This is unlike implementation units, which implicitly import the primary interface. However, the PMF does effectively import the primary interface – there is no emptying of the reachable symbol table at the PMF boundary. Unlike implementation units, it shares the primary's internal-linkage entities and GMF.

Again, the PMF is an exceptional implementation partition.

The author's recollection is that the results were consistent with permitting implementations to write the CMI at the point the PMF declaration was encountered, before any of the PMF contents had been parsed. The contents of the PMF do not need to be written to a CMI – they only affect generated object code. This is true for type, but as mentioned in Section 1.4, inline function definitions were not discussed.

2.4 PMF Exceptionalism

We have four causes of exceptionalism:

6 GCC's implementation combines the partition CMI directly into the primary's CMI, which means the partition CMIs are only needed when compiling that module's components. GCC does not (yet) implement the complete reachability semantics.

1. An imported partition that is not reachable from the primary's purview.
2. A partition that implicitly imports its primary interface.
3. A partition that observes the internal linkage entities of its primary interface.
4. A partition that observes the GMF of its primary interface.

The first can be removed by not considering the PMF as imported.

The second is only an exception because the PMF is considered a partition. If it were considered an implementation unit, primary interface importation would be the usual case.

The third and fourth are a consequence of being concatenated into a single source file. It would be a rather drastic change to the compilation model to reset symbol tables in the middle of a translation unit.

3 Proposal

3.1 PMF is an Implementation Unit

As mentioned above, the PMF is made more consistent with other components of the module system by considering it to be an implementation unit, and not an implementation partition:

- The private-module-fragment is semantically an implementation unit, with the addition of sharing the primary interface's global-module-fragment and its internal-linkage entities.

This provides a simpler model to reason with. Except for the module-linkage inline case, this is effectively how the PMF behaves.

3.2 Event Horizon

The boundary between the primary interface and the PMF should be made much harder – it is in effect an event horizon beyond which the primary interface cannot observe.

The WP already includes wording to make this so in many cases. [basic.def.odr]/10 and/or [dcl.inline]/7 should be altered such that the inline definitions must be available:

- At the end of the TU, augmented by,
- For calls within a module interface purview, before any private-module-fragment (if any)

Were the PMF considered an implementation unit, rather than a partition, the above questions are mooted:

- Implementation units never produce a CMI, importers never see their contents.
- Implementation units implicitly import their primary interface.

The only oddity would be that the PMF shares the primary's GMF & internal entities. This is consistent with the event-horizon metaphor though – information continues to flow in one direction past the horizon.

This is the defect the author would like to resolve.

3.3 Inline Function Definitions

Rather than require exported inline functions to be defined before the end of their declaring interface (primary or partition), require the definition to be reachable:

- Before the end of the primary interface, outside of any PMF, and
- Before the end of a module partition in which they are ODR-used

Similarly tighten the requirements of module-linkage inline functions, to be the same, with the possible proviso that if the definition is not reachable at the end of the primary interface before the PMF, then it is ill-formed for them to be ODR-used from outside the module. I.e. should they be selected by 2nd-phase ADL, the program is ill-formed in a similar manner to which is is expected P1498 determines for internal-linkage functions.

This is arguably not a bug fix, but additional flexibility, refer to Section 1.5.

4 Acknowledgments

Mathias Stearn noticed the GB079 response error.

5 Revision History

R0 First version