Document Number: p1184r1
Date: 2018-11-12
To: SC22/WG21 SG15
Reply to: Nathan Sidwell
nathan@acm.org / nathans@fb.com

A Module Mapper

Nathan Sidwell

The modules-ts specifies no particular mapping between module names (dotted identifier sequences or legacy header unit names), their interface source file and their Binary Module Interface. This leads to toolchains developing their own schemes. This paper describes an interface implemented as a serial protocol by which compilation tools may interrogate an entity encoding this mapping. This interface allows a compiler to be agnostic about the mapping.

1 Background

Compiling a module interface unit is expected to generate an intermediate file of no particular specified form. This intermediate form, a BMI, is read when processing import declarations (and module implementation units). The intent here is to speed compilation, and although the TS does not mandate this approach, known compiler implementations are taking this direction. Static analysis tools may do something different though.

Thus when compiling:

```
export module foo;
```

The compiler needs to determine where to write `foo`'s BMI. Similarly when processing:

```
import foo;
```

The compiler needs to know where `foo`'s BMI was placed.

More complex cases arise. In the second case, what if `foo`'s BMI has not yet been generated? Is that a case that should be handled, or must we require build systems to have predetermined a dependency graph and build it in the correct order?[1]

With the addition of (some parts of) ATOM, we now also have legacy header units. These are (sufficiently modular) header files compiled in an implementation-defined legacy header mode. They may be imported using a new kind of module name:

1 Nothing here precludes a BMI being multiply built for different importers. This increases overall work, but may reduce the need for dependency analysis. Of course each such build must produce equivalent BMIs.

```
import "foo.h";
import <baz.h>;
```

As such legacy header units export macros visible it is now longer to preprocess the source in isolation from reading in BMIs. This essentially makes it impossible for the dependency graph to be determined up front, before any module compilation. Preprocessing requires reading in the macro tables at each legacy import before continuing.

A further issue is that during legacy header compilation, one may encounter:

```
#include "foo.h"
```

That of course textually includes the body of `foo.h` into the current compilation. But if `foo.h` itself is a legacy header we can end up with two instances in our compilation. Apart from inviting ODR violations, it is inefficient – why parse `foo.h` twice? Thus the introduction of include directive translation, where the above may be rewritten to:

```
import "foo.h";
```

Determining whether to perform this translation complex.

Thus we reduce to three questions:

- When exporting a module, where should the BMI be placed?
- When importing a module, where should the BMI be found?
- When including a header, should it be translated to an import?

2 Experimentation

The GCC modules implementation began with a fixed mapping of module name to BMI filename, and a search path to look for them. This answered the loading question at the cost of forcing a particular naming scheme. When producing a BMI, the fixed mapping was used to write into the current directory. Options were added to manipulate the search path.

When searching for a BMI failed, the compiler spawned a user-provided wrapper program. That was tasked with making sure a repeated search would succeed (or return a failure). Options were added to control the wrapper program.

These met initial modest needs, but failed with the first customer, Boris Kolpackov, who wanted to have an arbitrary mapping and per-compilation control of the output file. Options were added to control mapping files and output names.

The addition of an include translation scheme indicated the implementation was on a path to a myriad of options, each a special case.

Conversations with Richard Smith & David Blaikie moved towards providing a distinct component in the compiler to handle these questions. In particular having some way of finding the dependency graph during compilation, because of the above mentioned interdependence of preprocessing and legacy header unit compilation. Fundamentally, the questions can be too complex to be solvable by a block of data given to the compiler before starting.

Initially a plugin was considered, but that could mean different plugins for each compiler/build system combination. In considering how a plugin might work, a client/server architecture suggested itself.

3 Client/Server

The idea of a client-server scheme has the build system providing a server, and compilations may interact with that as clients. The build system acts as a cache of module-name/BMI-location tuples, and has more global visibility of the system. Compilations are simply concerned with processing a source file. *This is not a general-purpose compile server.*

If adopted by multiple compilers, it would provide a uniform way in which module-aware build systems could interact with them.

A default scheme will be needed, and one is provided by a default server. Whether the defaults are correct, or whether it would be better implementing that directly in the compiler is an open question. Having it separated out does allow experimentation by non-compiler experts. The compiler itself is now agnostic about mappings.

4 The Protocol

The protocol is a simple text-based query/response scheme. It is intended for use on systems sharing, or duplicating, a file system, and large objects (such as source or BMIs) are accessed via that. Connections are expected to be local, there is no encryption layer, or DOS defense. In order to reduce round trips, a batching mechanism is employed, which can take advantage of some ATOM features.

The compiler initiates connection and queries, the server responds. It is line based and consists of space-separated words, where the final item on a line may contain embedded whitespace.

Protocol completeness is not claimed.

4.1 Handshake

The first query is a handshake:

```
HELLO $ver $kind $ident
```

The protocol version number, $ver, is currently 0. The tool $kind is ‘GCC’, and I expect other tools to uniquely identify themselves. The current server implementation ignores this field as it has no need for

it. The final item, $ident, is a user-provided identity given to the compiler invocation. If none was given the source pathname is used.

The response is either:

```
OK $ver $repopath
```

to indicate successful handshake. Again $ver is currently 0, responding with it will allow systems to use a highest common denominator protocol, should this be extended. The $repopath value is a file system location against which all non-absolute BMI names are to be interpreted. If connection fails the response is:

```
ERROR $msg
```

Where $msg is user-meaningful text.

4.2 BMI Mappings

Two queries determine the name of a BMI given a module name:

```
IMPORT $module
EXPORT $module
```

The specified module needs to be imported, or is being exported. A module implementation unit issues an IMPORT for the implemented module. The response is either:

```
OK $bmipath
```

or

```
ERROR $msg
```

When exporting a module, the completion of the export is via:

```
DONE $module
```

There is no response. Note that the compilation may not have completed the object-file generation of the interface unit. This permits a build system to launch compilations depending on this module before the interface itself has completed compilation.

Clearly, this query gives the server dependency information between the source being compiled and the module being imported or exported.

4.3 Include Translation

When processing an include directive during legacy header compilation (and possibly other compilations), it is necessary to know whether to textually include the header or translate to an import declaration. The query is:

```
INCLUDE $header
```

where $header is the to-be included header. The response is one of:

```
INCLUDE
```

to textually include it, or

```
IMPORT [$module]
```

to import as a module. The latter can provide a module name to import, otherwise it is the header name. Although not implemented, it may be that the query should also contain the location of the source containing the `#include` directive, and possibly other information. Likewise, perhaps the `INCLUDE` response could contain the resolved pathname to include – allowing the build system to act as a cache of header file lookups. A third response is postulated:

```
SEARCH
```

where the intent is to have the compiler resolve the header location and retry the query with a full path. This saving the build system from duplicating header search path algorithm. It seems fragile to force that algorithm to be in two places.

4.4 Batched Queries

To avoid multiple round trips, when processing an ATOM preamble in particular, requests may be batched using a '`+`' prefix on each non-final line, and either a '`-`' or omitted prefix on the final line (which may be otherwise empty).

Responses to a batched set of requests must be in order, and might or might not be batched themselves. If they are batched, the batch must contain the complete set of responses – there can be no split-batch responses.

4.5 Sample Implementation

This is currently implemented in the GCC modules compiler, which requires you Build Your Own Compiler. The options are documented in the GCC manual, but are subject to change due to the experimental nature of this development.

The `-fmodule-mapper=$val` option controls the mapper, allowing it to be invoked in one of the following ways:

- a file of tuples, or
- a local domain unix socket,[2] or
- an ipv6 domain socket & port, or
- a program to spawn using stdin/stdout communication.

The tuple file is the least flexible scheme, requiring all potential questions to be answered before starting compilations. The default is to spawn a provided program, with in-built mapping capability. It is expected that the ipv6 socket case will *not* make the server socket world-accessible.

A ident may be provided, which is used when initiating connection, and allows the build server to distinguish between compilations. It defaults to the source file name.

One may inspect the communication by invoking the default mapper in verbose server mode communicating over a port:

```
cxx-mapper -n :12345
g++ -fmodules-ts -fmodule-mapper=:12345
```

4.5.1 Attack Vectors

This protocol is not intended for an unprotected environment. One could layer it on top of a secure protocol, but that would be a surprising use case. Transfer of BMIs themselves is through the file system.

As an arbitrary hostname is an option, connections could be remote, but that would also be a surprising use. It is provided to permit distributed builds within a local network. One might expect the build system to not make that server accessible externally. Nor would one expect it to need to defend against attacks from within the organization.

Likewise the distinguishing ident token is not a cryptographic cookie. It is merely an arbitrary non-secret text string – a name – so that multiple compilations may interact with the same build system.

4.6 Future Directions

This protocol has been developed on a compilation system. Static analysis systems may want source rather than BMI locations. Experimentation might suggest such a use should be by a modal setting or new request kinds.

2 Other OS's could provide their own variant of local sockets.

A new way of avoiding building module BMI's in order to determine dependencies might be a reverse mapping from header unit names to the source or header file from whence they are generated. That source could be preprocessed to extract just the macro definitions, which are then used to continue the dependency preprocessing.

Non-ASCII names have not been considered in detail. Perhaps a UTF8 encoding will suffice.

5 Revision History

R0 Presented San Diego'18

R1 Amended some terms to avoid unintended assumptions.