Document Number: p1184r2
Date: 2020-07-10
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 and their interface source file s, and for both named modules and header-units to their Compiled Module Interface name. 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, or responsible for creating said objects. This interface allows a compiler to be agnostic about the mapping, and sharing of resources across several compilations.

1 Background

Compiling a module interface unit is expected to generate an intermediate file of no particular specified form. This intermediate form, a CMI, is read when processing import declarations (and module implementation units). The intent here is to speed compilation, and although the standard 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 CMI. Similarly when processing:

```
import foo;
```

the compiler needs to know where `foo`'s CMI was placed.

More complex cases arise. In the second case, what if `foo`'s CMI 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]

We also have header-units, which are named from the header-files from whence they come. They may be imported:

1 Nothing here precludes a CMI 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 CMIs.

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

As header units export macros it is now longer to preprocess the source in isolation from reading in CMIs. This essentially makes it difficult for the dependency graph to be accurately determined up front, before any module compilation. Preprocessing requires reading in the macros from each legacy import before continuing. P1857:‘Modules Dependency Discovery’ documents various heuristics can be used to determine the dependency graph without full preprocessing.

A further issue is that #include directives may be translatated to import-declarations of the specified file as a header-unit. Determining whether to perform this translation is complex.

Thus we reduce to three questions:

- When exporting a module, where should the CMI be placed?
- When importing a module, where should the CMI 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 CMI 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 CMI, the fixed mapping was used to write into the current directory. Options were added to manipulate the search path.

When searching for a CMI 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/CMI-location tuples, and has more global visibility of the system. Compilations are simply concerned with processing a source file.

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.

The system model is of a build system directly, or indirectly, spawning compilation jobs. Those compilations connect to a server controlled by, or part of, the build system. The build system and compilations already have a mechanism for bulk data transfer, with a common naming scheme (I.e. the file system).

3.1 What it is Not

This is not a general-purpose compile server. The intent is that there is a one to one correspondence between build invocations and servers. A server instance is live only as long as the build it serves. Only the compilations of that particular build connect to that build's server invocation. The means by which that can be checked or enforced depends on the means of connection. For instance, if connection is by anonymous pipes, there can be no other connections. But that requires the pipe file descriptors to be propagated correctly from the build system to the compiler. Experience with GNUMake's jobserver shows that is brittle[2]

It is not designed for a hostile environment. A shared filesystem is already presumed. The build system is invoked by the user, and therefore has (or should enforce) the access rights of that user. Obviously, if the build system runs as a different user, there is already a privilege escalation possibility, should it contain bugs. If connection is via named unix local sockets (`AF_UNIX`), the file system can enforce connection rights. Of course that restricts connections to the local system. A distributing build system will need something different, for instance IP sockets. Clearly these should not be exposed to the external internet, and connection origins should be validated. The port number is not fixed, and could

2 GNUMake needs to know the rule being invoked is subjob-like, requiring users to prefix a '+', if heuristics fail. If a sub job itself is a wrapper, it needs to ensure those file descriptors are passed on to the process(es) it invokes.

be build-specific. As described in Section 4.6, compilers may identify themselves via an identifier generated in an implementation-specific manner. It is not a cryptographically strong cookie. Merely a mechanism by which a build system can identify the target producing possibly new dependencies.

It is not a bulk transfer mechanism. As mentioned above, the compilations of a particular build will already have a bulk transfer mechanism to (a) get the sources to compile and (b) deliver the compiled object files to the linker. In common with header files, multiple compilations can use the same CMI file for importing. Further at least 2 module implementations use `mmap` to read the CMI, thus sharing physical memory pages, for concurrent compilations on a single system. Were CMIs delivered via this protocol, there would be a per-import bandwidth requirement, and per-import memory use.

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 CMIs) 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.

The compiler initiates connection and queries, the server responds. Every request has a response. It is line based and consists of whitespace-separated words. Messages consisting of no words are ignored.

Protocol completeness is not claimed. The version described in this r2 paper is incompatible with that of the r1 paper, but fortunately the initial handshake is sufficiently similar to notice the incompatibility and provide a diagnostic.

4.1 Environment

The mechanism by which the compiler connects to the build system is not specified.

As mentioned in Section 3, it is intended that the build system launches compilations, and can thereby inform them of how to connect. Also that they have a common view of a shared filesystem, and the build system knows the compiler's working directory.

As the server and the client share a file system (or a mechanism to transfer bulk data using common names), it is presumed that they share a common pathname convention. What that convention is, is currently unspecified. While URIs provide a well-defined resource naming scheme, they are too capable for the needs here, and might encourage cross-site interaction going against the restrictions outlined in Section 3.1.

4.2 Encoding Layer

The encoding layer has seen the greatest change in this version. The ability to use tools such as `netcat` to probe an implementation remains.

Each message consists of a line of Unicode text encoded as UTF8[3] octets, ending with a newline (0xa). The newline may be preceded by a continuation marker (see below). The message itself consists of an arbitrary number of words, separated by one or more space (0x20) or tab (0x9) characters. Words need not be quoted, if they only contain:

- Upper or lower case ASCII letters [0x41-05a] or [0x61-0x7a]
- Digits [0x30-0x39]
- Plus (0x2b)
- Minus (0x2d)
- Underbar (0x5f)
- Percent (0x25)
- Slash (0x2f)
- Dot (0x2e)

Thus many pathnames do not require quoting. Words containing characters in the ranges [0x0-0x20] and above 0x7e must be quoted. Other words should be quoted.

Quoting is delimited by tick (0x27) characters. Within a quoted word, backslash (0x5c) is used as an escape mechanism. Within a quoted word, characters [0x-0x1f], 0x27, 0x5c & 0x7f must be escaped. Other characters may be escaped, but need not be. The following escapes are available:

- \' (0x5c,0x27) a tick (0x27)
- \\ (0x5c,0x5c) a backslash (0x5c)
- \n (0x5c,0x6e) a new line (0xa)
- \t (0x5c,0x64) a tab (0x9)
- \<hex>{1,2} (0x5c,[0x30-0x39,0x61-0x66]{1,2} hex encoding

Any octet may be encoded by the hex encoding, which explicitly uses lower-case characters, and may be one or two hexadecimal digits. The shorter form may be used only when one digit is sufficient, and the next (unescaped) character could not be mistaken for one. Unicode characters that are encoded in multiple octets do not need escaping, and presumed correct. Thus any octets in the range [0x80-0xff] are simply passed in quoted words. Escaping is at the octet level, not the unicode character level.

The \n, \t, \\ and \' encodings are mnemonically helpful to human readers of messages. Note that these encodings need not be used, the more generic hex encoding may be used in place.

3 See below for non-utf8 filesystems.

Quoting behaviour may be initiated and terminated mid-word, there is no requirement that it encompass a whole word.

4.2.1 Batching

Messages may be batched into a single block. Continuation is indicated by ending a message with an unquoted space, semicolon (0x20,0x3b) immediately before the newline character. This indicates another message follows in the block. In this manner the message buffering layer does not need to lex or parse block contents to determine whether additional data should be waited for. If the last character received was a newline, and it wasn't immediately preceded by a semicolon, more characters are to be expected.

The responses to a batched set of requests must also be batched. A batch of one message is equivalent to an unbatched message.

In GCC's case, all top-level named-module imports are deferred until the end of preprocessing, and requested as a block at that point. C++ parsing commences once that block request has been responded to.[4] The compiler's memory footprint will essentially be that of a preprocessor when waiting, as no ASTs of the program will have been constructed, and nothing more than macro tables of imported header-units will have been read.

4.3 Pathnames

As mentioned above, the build system is in control of the compilations, and will have provided them with at least the source pathname to compile. Commonly the build system and compilation will share a common filesystem, and the former will know the latter's working directory. Thus pathnames are naturally meaningfully transferred from the server to the client.

Therefore pathames specify location in the compiler's view of the filesystem.

In a heterogeneous distributed build, the build system will still have knowledge of the compiler's filesystem but may need to apply extra-protocol pathname translation.

4.3.1 Non-UTF8 Filesystems

Some filesystems use a different encoding for their pathnames, (eg UTF16le). It would be awkward to reencode such pathnames to and from UTF8. Fortunately, in this version it is known what words of a message are filenames. The encoding layer is sufficient to pass through other character encodings, as it does not presume the UTF8 encoding for characters outside the ASCII set is valid UTF8 – the right escapes will be inserted to ensure end-to-end fidelity. Thus pathnames could use the system's character encoding (as mentioned in Section 4.1, they have a common pathnaming scheme).

4 As you will infer, GCC preprocesses the entire token stream before C++ parsing.

There is also the complexity of header-unit names. These are derived from the pathname of the header-file they derive from, and hence the filesystem's character encoding.

I am insufficiently versed in such subtlety of character encodings to suggest anything more than 'experimentation needed'.

4.4 Module Names

There are two kinds of module names – those for named modules and those for header-units. These are distinguished by ensuring the latter unambiguously look like pathnames. Further, header-unit names must be resolved by the compiler to the header-file they correspond to.[5]

Header-names are either absolute pathnames (using the local system convention), or the are forced to be relative paths by ensuring they begin with '`./`' (or whatever the local system convention is for the current directory). Thus the named module '`bob`' will have a module name of '`bob`', but the header unit "`bob`" will have the name '`./bob`', if it was found in the current directory. If it was found in the `foo` subdirectory, its name is still prefixed with '`./`' as '`./foo/bob`'.

4.5 Errors

If a request is malformed, or fails for some reason an error response is given. The format is

```
ERROR $msg
```

The `$msg` is a human readable word. Error codes are not used, as that proved awkward. Generally errors result in catastrophic failure to compile, and `$msg` can be presented to the user .

It is unspecified whether such message text has been localized at the server side, or should be localized by the compiler.[6]

4.6 Handshake

The first request is a handshake:

```
HELLO $ver $agent [$ident]
```

The protocol version number, `$ver`, is currently 1. The `$agent` identifies the compiler. I expect different compilers to require and produce incompatible CMIs. The final item, `$ident`, is optional and intended to identify the particular compilation. The mechanism by which the compiler knows what `$ident` to provide is unspecified, however it is recommended that the source file name be used as a default.

5 The "" or <> quoting of the header-unit has been resolved at this point. To do otherwise would require the compiler and server have a common include path, and, for "", or the common include_next extension require the indication of the file containing the import and/or its location within the include path.

6 Experimentation with error codes showed them to be more awkward than useful. Perhaps an error code could be added as an optional component of the error response.

The response is either:

```
HELLO $ver $agent
```

to indicate successful handshake. Again $ver is currently 1. The $agent word identifies the build system to the compiler. If connection fails the response is an ERROR.

It is currently unspecified how mismatched version numbers should be handled. A likely behaviour is that communication is restricted to the lowest version's set.

A successful handshake places the system in the connected state, where other requests can be made. There is no requirement for the handshake to be in a separate block to other requests – although, if the handshake fails those subsequent requests will each deliver an ERROR response.

A common use case is to batch the handshake with a module repository query:

```
HELLO 1 GCC quux.cc ;
MODULE-REPO
```

and expecting a response similar to:

```
HELLO 1 gcc ;
MODULE-REPO gcm.cache
```

4.7 Module Repository

The concept of a repository of module CMIs is supported. All CMI pathnames are relative to this repository. The compiler issues a:

```
MODULE-REPO
```

request. The response will be:

```
MODULE-REPO [$repo]
```

where $repo is the optional path to the repository.

The default repository is '.', the compiler's working directory.

4.8 CMI Mappings

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

```
MODULE-IMPORT $module
MODULE-EXPORT $module
```

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

```
MODULE-CMI $cmi
```

or an ERROR.

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

```
MODULE-DONE
```

which expects an 'OK' 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's CMI before the interface itself has completed compilation. If compilation of the module interface fails, the compiler need not explicitly inform the build system. It can simply exit (with some kind of error code).

Clearly, these requests implicitly give the server dependency information between the source being compiled and the module being imported or exported.

4.9 Include Translation

When processing an include directive, it is necessary to know whether to textually include the header or translate to an import declaration. The query is:

```
MODULE-INCLUDE $header
```

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

```
INCLUDE-TEXT
```

to textually include it,

```
INCLUDE-IMPORT
```

to import as a module, or:

MODULE-CMI $cmi

to explicitly indicate the $cmi to import. If the INCLUDE-IMPORT response is received, a MODULE-IMPORT request will need to be issued to determine the CMI name.

As mentioned in Section 4.4 the header name has already been resolved to a file, using the compiler's include path. If the include path contains absolute pathnames, the queried header names could be absolute.

4.10 Sample Implementation

This is currently implemented as a C++11 library, libcody.[7] This is used in GCC, replacing the horrible hack I previously had. In addition to supporting inter-process communication, libcody provides an in-process mechanism, and that is now GCC's default (rather than fork a subprocess).

GCC continues to provide a default server program, but using `libcody`. It is no longer spawned by default.

The library provides connection helpers, but does not specify any particular syntax for that. 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 pair of file descriptors (or a single bi-directional file descriptor)
- a local domain unix socket,[8] 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.

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.

If no `-fmodule-mapper` option is given, the environment variable `CXX_MODULE_MAPPER` is queried for a value that, if present, matches `-fmodule-mapper`'s argument form.

4.11 Future Directions

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

It may be profitable for the server to resolve include names, rather than have each compilation discover that anew. The (often negative) file system look ups within a complex build can accumulate to a noticeable extent.

7 github.com/urnathan/libcody

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

4.12 LTO

Link Time Optimization is another situation where by a compiler would like to communicate with a build system. LTO is initiated at the linker stage and often performs a pre-link, followed by a set of concurrent compilations. Both Clang's thin-LTO, and GCC's LTO operate in this manner. Those concurrent compilations are jobs the build system could manage, as it is aware of the overall level of concurrency desired.

In GCC's case, it currently performs this step by writing a Makefile and then spawning make with a user-specified concurrency level. It does attempt to use GNUMake's jobserver protocol, but that is only available if the original invoking make recognized the link step as possibly needing it. Make's only knows that if (a) make is explicitly spawned by that build rule, or (b) the build rule is prefixed by '+'. Users often forget the '+'.

A Google Summer of Code student is working on this issue.

4.13 Make

I have not updated the GNU Make proof-of-concept I described in p1602 'Make Me A Module'.

5 Revision History

R0 Presented San Diego'18

R1 Amended some terms to avoid unintended assumptions.

R2 Redesign for more general application.