Document Number: p1602r0
Date: 2019-03-01
To: SC22/WG21 EWG
Reply to: Nathan Sidwell nathan@acm.org
Re: p1184 A Module Mapper

Make Me A Module

Nathan Sidwell,

P1184 presented a protocol by which a compiler and build system could communicate, allowing precise module importing requirements to be determined without a prescan. This paper describes adding a mapper to GNU Make.

1 Background

As described in P1184, there is a tension between compilers and build systems regarding dependencies between modules. A module cannot be imported before its interface has been compiled, but discovering the dependency graph requires more sophisticated prescanning than that needed for header files. In general, it is not possible to determine a precise dependency graph without:

1. Additional meta-data, or
2. Converting header imports to `#include` directives during a preprocessing prescan, or
3. Invoking additional compilations during a compilation

A conservative approach of presuming all import declarations are reachable, may simplify a prescan at the expense of dealing with deferring compilation failure reporting until it is known the compilation is a true dependency.

P1184's protocol allows option 3 by creating a back channel from the compiler to the build system. But of course, requires the build system to provide a server.

1.1 GNU Make

GNU Make is an implementation of the Unix Make program, which was originally implemented by Stuart Feldman in 1976 (https://en.wikipedia.org/wiki/Make_(software)). Make reads a *Makefile*, which contains *rules* that define dependencies between *targets* and *prerequisites*, and *commands* describing what to execute if the target is out of date. Generally these commands will cause the target to be rebuilt. An example might be:

```
foo.o : foo.cc
	g++ -c foo.cc
```

This rule specifies that `foo.o` depends on `foo.cc` and `g++` should be invoked if it is out of date.

Make also supports variables using a shell-like `$` prefix. Because of history, multi-character names must be `()` or `{}` delimited. These are also a hook by which internal functions are invoked. Some variables are automatically created from various components of the rule being executed

Make proceeds by constructing a dependency graph starting at the ultimate set of targets, and rebuilding any target that is older than any of its prerequisites. Ordering within the Makefile is largely unimportant.

GNU Make consists of roughly 36,000 lines of C89. It implements several extensions to the original implementation, some of which are made use of with this extension.

2 Supporting Modules

Early in the development of GCC's module implementation I considered how C++ Modules might be integrated into a Make-based build. The early, now removed, concept of a helper program would, by default, look for a Makefile and attempt to invoke that to build dependent modules. This was unsatisfactory for several reasons – racy parallelism, interfering with a superior Make's prerequisite correctness. With p1184 I thought there was no hope of working with Make, and stopped considering how it might be done.

More recently I realized that it would be possible to treat module prerequisites as auto-generated dependencies, and with suitable syntax allow users to override defaults. But I had no previous experience with Make's source code and did not know how practical such changes might be. I chose GNU Make, as it is commonly used to build GNU software, and is the build system I am familiar with, as a user.

In augmenting GNU Make there are essentially three modifications needed:

1. Adjusting job control to avoid deadlock
2. Injecting new dependencies into the graph
3. Serving mapper requests

2.1 Mapper Requests

The mapper implements a server interface, accepting new connections on a unix-local (`AF_UNIX`) socket. Requests are served using a `select` system call, waiting on file descriptors with something to read, or a child signal.

A new connection adds a new client object, which is then communicated with. Clients that are not blocked on a module dependency may send requests, which are buffered until complete. Once complete the set of requests is processed. If the requests are immediately satisfiable (no new target

needs building), responses are immediately returned. Otherwise the client naturally blocks while the required targets are built.

The `select` system call is inserted into GNU Make's wait-for-child routine, discussed below.

2.2 Job Control

GNU Make's job control is rather baroque, because of its history. Originally, it would only permit a single live job, and could therefore be tightly integrated with the dependency graph walk. Currently GNU Make supports the following job control variants:

1. One job
2. Unbounded number of jobs
3. Up to N (>1) jobs
4. Job spawning under control of a *jobserver*
5. Job spawning up to a specified load average

Because of this history, GNU Make does not have a single point in its main loop where it either spawns or waits for jobs. The closest it has is case 5, which is at a single point in this loop. Other cases are handled at the point it wishes to spawn a new job, using a common wait-for-child routine. If at the job limit, this waits for at least one job completion, and depending on OS characteristics, polls for any other completed jobs.

Rather than handle all cases I extend cases 3 & 4. Alter case 1 to be handled as case 3. Cases 2 & 5 do not need adjusting. All that is required here is accounting for stalled jobs, and not count them in the current count of (live) jobs.

A jobserver is simply an accounting mechanism that allows recursive Make use to operate in parallel using a global accounting of live jobs.

2.3 Dependency Injection

When the mapper discovers a new BMI is needed, it needs to inject a new dependency into the graph. This new dependency cannot be added to the job that is stalled – because one of Make's invariants is that a job is only started once its prerequisites are ready. Instead a new global target is injected, marking it as a module-relevant one. If the target is already waiting for prerequisites, or currently running, it is still inserted as a new global target (duplicates do not cause harm).

The main loop will pick up the new target and proceed to handle it just as if it was specified on the command line. Because of Make's structure these new targets have to queued to an intermediate list, which is spliced into the global goals at an appropriate point.

When a module-relevant target completes, the mapper is informed and updates all clients waiting for that job. This might cause a response to the client, unblocking it, in which case the job accounting is adjusted (clients can be waiting on several imports).

2.4 Error Handling

If a circular module dependencies are discovered, waiting clients are informed of the error, and will then terminate. Similarly, if a module build fails, waiters are informed and will themselves fail.

If `make` terminates because of an unrelated failure, again waiting clients are informed.

2.5 Rule Syntax

GNU Make provides implicit rules, and uses a pattern matching syntax to describe them. This is how a known target suffix can be generated from a known prerequisite suffix. For instance, the default compilation might be given by:[1]

```
%.o : %.cc ; $(COMPILE.cc) $(OUTPUT_OPTION) $<
```

Here, '`%`' indicates a pattern match and marks the stem. If a file with a `.o` suffix is a target, Make sees if a `.cc` suffix variant can be found, and if so will treat it as a dependency. If the rule is active the command will invoke the C++ compiler.

The trick is to treat module names as-if they are files with a particular suffix.

```
%.c++m : $(CXX_MODULE_PREFIX)%.gcm ;
```

Here an artificial `%.c++m`[2] target depends on a `%.gcm` BMI file, possibly in a user-specified directory. The command is empty. Thus a request to satisfy an for an '`import bob;`' declaration results in searching for a `$(CXX_MODULE_PREFIX)bob.gcm` file via a `bob.c++m` target. A further pattern rule:

```
$(CXX_MODULE_PREFIX)%.gcm :| %.o ;
```

completes the graph. This indicates the BMI is dependent on the object file, but uses an order-only dependency, marked by the `|` character. This is needed because the BMI is actually emitted before the object file, and therefore might be seen as older. All we need to show is that if the object file needs regenerating, then the BMI depends on that happening. But the BMI being older than the object file is not in itself sufficient reason.

At that point the earlier rule to generate an object file triggers.

---

1 Ignoring the `.SUFFIXES` approach, which makes this even terser.

2 Chosen to be an unlikely real suffix, but still mnemonic.

For an export request, all that is required is locating the rule, but not making it a new target in the graph. The BMI prerequisite name is immediately reported back to the client.

The mapper needs to mark the intermediary `.gcm` as to be kept due to secondary expansion, and the `.c++m` file is marked a phony target to save a file system check.

Notice that this default scheme extends the basename requirement of sources and object files to the module name too. Of course the user can override this with explicit rules, just as an explicit rule can permit decoupling an object file base name from its source file name.

To deal with header units, additional default rules are provided:

```
"%".c++m : $(CXX_MODULE_PREFIX)%.gcmu ;
<%>.c++m : $(CXX_MODULE_PREFIX)%.gcms ;
```

These match on quoted or angled header names, and show them dependent on slightly different BMIs. The associated build rules are:

```
$(CXX_MODULE_PREFIX)%.gcmu : %
   $(COMPILE.cc) $(call CXX_MODULE_HEADER,"$*") $<
$(CXX_MODULE_PREFIX)%.gcms : %
   $(COMPILE.cc) $(call CXX_MODULE_HEADER,<$*>) $<
```

The behavior with quoted header names is made simpler by Make's lack of a robust quoting scheme. Of course this will fail on header names containing interesting characters, but as Make already works with header names, these should not be present in existing Make-based builds.

2.6 Examples

As an example, here is a Makefile for 3 module units `u.cc`, `m.cc` & `n.cc` and a header unit `"leg.h"`:

```
# Request local socket mapper
CXX_MODULE_MAPPER := =

# Non-installed build compiler
GCC_DIR := $(HOME)/egcs/modules/obj/x86_64/gcc
CXX = $(GCC_DIR)/xg++ -B $(GCC_DIR)/
CXXFLAGS += -fmodules-ts  -MD

OBJS := u.o m.o n.o

a.out : $(OBJS)
  g++[3] -o $@ $^

# mark as a header unit
"leg.h".c++m :
```

3 Use the installed compiler for linking.

The `CXX_MODULE_MAPPER` assignment requests a local socket server. As the name is otherwise unspecified, a default in `/tmp` is constructed. This value is passed to jobs in their environments. The three object files are named, and the usual link line to generate a program provided. In this case the only special piece of user annotation needed is to mark `"leg.h"` as a header unit by naming it as a target. We do not have to provide prerequisites as Make will automatically find them from the default pattern rule when none are explicitly specified.

In this particular set of sources `u.cc` imports modules ‘`m`’ and ‘`n`’, and includes `"leg.h"`. Source `m.cc` also imports module ‘`n`’.

A clean build, with debug output produces the following annotated output:

```
nathans@zathras:26>obj/x86_64/make -f test.make –debug=p
/home/nathans/egcs/modules/obj/x86_64/gcc/xg++ \
  -B /home/nathans/egcs/modules/obj/x86_64/gcc/ \
  -fmodules-ts  -MD -c -o u.o u.cc

# we start building u.cc, which connects:
mapper:1 connected
mapper:1 processing 'HELLO 0 GCC 0x7184a0'
mapper:1 sending 'HELLO 0 GNUMake '

# and queries about translating the #include directive:
mapper:1 processing 'INCLUDE "leg.h"'
mapper:1 sending 'IMPORT '

# queries where to find the specified header unit’s BMI:
mapper:1 processing 'IMPORT "leg.h"'

# make then starts another job, but because I haven’t prioritized the
# new dependency, picks something else from its build graph:
/home/nathans/egcs/modules/obj/x86_64/gcc/xg++ \
  -B /home/nathans/egcs/modules/obj/x86_64/gcc/ \
  -fmodules-ts  -MD -fdump-lang-module   -c -o m.o m.cc

# which connects and then indicates it’ll export m,
# but needs an import of n:
mapper:2 connected
mapper:2 processing 'HELLO 0 GCC 0x718540'
mapper:2 sending 'HELLO 0 GNUMake '
mapper:2 processing '+EXPORT m'
mapper:2 processing '+IMPORT n'
mapper:2 processing '-'

# the m.cc job blocks, and another prerequisite is chosen:
/home/nathans/egcs/modules/obj/x86_64/gcc/xg++ \
  -B /home/nathans/egcs/modules/obj/x86_64/gcc/ \
  -fmodules-ts  -MD -fdump-lang-module   -c -o n.o n.cc

# connects, exports ‘n’ and completes:
mapper:3 connected
mapper:3 processing 'HELLO 0 GCC 0x7185e0'
```

```
mapper:3 sending 'HELLO 0 GNUMake '
mapper:3 processing '+EXPORT n'
mapper:3 processing '-'

# response to export query:
mapper:3 sending 'OK n.gcm'

# it is done:
mapper:3 processing 'DONE n'

# inform the m.cc job it is exporting m.gcm and reads n.gcm:
mapper:2 sending '+OK m.gcm'
mapper:2 sending '+OK n.gcm'
mapper:2 sending ''

# the n.cc job terminates:
mapper:3 destroyed

# the m.cc job writes its BMI:
mapper:2 processing 'DONE m'

# and terminates:
mapper:2 destroyed

# finally leg.h's compilation starts:
/home/nathans/egcs/modules/obj/x86_64/gcc/xg++ \
  -B /home/nathans/egcs/modules/obj/x86_64/gcc/ \
  -fmodules-ts  -MD -fdump-lang-module   -c \
  -fmodule-header='"leg.h"' leg.h

# it connects and informs of an export:
mapper:4 connected
mapper:4 processing 'HELLO 0 GCC 0x720ff0'
mapper:4 sending 'HELLO 0 GNUMake '
mapper:4 processing '+EXPORT "leg.h"'
mapper:4 processing '-'

# and is informed of the output BMI, which it completes:
mapper:4 sending 'OK leg.h.gcmu'
mapper:4 processing 'DONE "leg.h"'

# which allows us to finally inform the u.cc job of that BMI:
mapper:1 sending 'OK leg.h.gcmu'

mapper:4 destroyed

# then the u.cc requests its two imports:
mapper:1 processing '+IMPORT m'
mapper:1 processing '+IMPORT n'
mapper:1 processing '-'

# which are immediately available:
mapper:1 sending '+OK m.gcm'
```

```
mapper:1 sending '+OK n.gcm'
mapper:1 sending ''

# and finally we're done:
mapper:1 destroyed

# and the final link can proceed:
g++ -o a.out u.o m.o n.o
```

2.7 Patches

I have cloned the GNUMake repository (https://git.savannah.gnu.org/git/make.git) to Github, adding a module-hack branch to develop on.

Clone and build with:

```
git clone git://github.com/urnathan/make.git
cd make
git branch module-hack
./bootstrap
./configure
make
```

Copy the resulting `make` some where, possibly renaming it to avoid confusing.

```
cp make ~/somewhere/local/bin/make-module
```

My build of choice is x86_64-linux-gnu, YMMV.

The new `cxx-mapper.c` source file is 900 lines, and the entire diffstat is:

```
src/cxx-mapper.c | 900 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
src/debug.h      |   1 +
src/dep.h        |   7 +
src/file.c       |   2 +
src/filedef.h    |   1 +
src/implicit.c   |   2 +-
src/job.c        |  53 +++++--
src/job.h        |  13 ++
src/main.c       |  32 +++-
src/makeint.h    |  18 +++
src/posixos.c    |  54 +++++--
src/remake.c     |  41 ++++-
src/rule.c       |   4 +-
src/variable.c   |  22 ++-
14 files changed, 1097 insertions(+), 53 deletions(-)
```

2.8 GCC Dependency Generation

GCC's dependency generation has been augmented to emit module import prerequisites. With `-MD` or `-MMD`, dependencies of the form:

```
u.o: "leg.h".c++m m.c++m n.c++m
CXX_IMPORTS += "leg.h".c++m m.c++m n.c++m
```

are included in the output.

2.9 Future

This is a proof-of-concept hack. I hope to have time to clean it up and make it more robust. I do not know whether GNU Make maintainers would accept it, perhaps via extending GNU Make's plugin scheme.

While P1184 also described an IPv6 interface, I have not implemented that. Unlike the sample server in GCC's source I have not used `epoll`. Perhaps if Make's main loop was restructured this would be a possibility. It is also desirable to have Make spawn one of the immediately dependent subjobs in order to minimize wall time in a parallel build.

It would be possible to specify header unit aliasing with rules of the form:

```
"header".c++m: "alias".c++m
```

however, this is not currently implemented.

To provide header unit mappings, Make's include mechanism could be employed to read additional data. Although Make's `VPATH` mechanism looks attractive to implement include directory searching, it does not have the complete capabilities needed, and reimplementing the compiler's search algorithm risks behaviour divergence. Also header searching is known to be system-call intensive and extensively optimized in preprocessor implementations. P1184 noted the possibility of a feedback response asking the compiler for more information.

Ben Boeckel is investigating extending GCC's dependency output to emit a JSON format.

3 Acknowledgements

Thanks to GNU Make maintainer Paul Smith for orienting me in GNU Make's source code.

4 Revision History

R0 First version, post presentation at Kona '19 meeting.