mkmf: f90/cpp dependency analysis and makefile generation

fortran links

- gfortran: gnu fortran compiler [http://gcc.gnu.org/wiki/gfortran]
- g95 [http://g95.sourceforge.net]
- photran [http://www.eclipse.org/photran]
- fortran resources: knoble [http://www.personal.psu.edu/faculty/h/d/hdk/fortran.html]
- clrc fortran tool survey [http://www.sesp.cse.clrc.ac.uk/publications/tools_report_2005/tools_report_html.html]
- fortran compiler comparisons (from polyhedron) [http://www.polyhedron.com/compare0html]

`mkmf` is a tool written in perl5 that will construct a makefile from distributed source. A single executable program is the typical result, but I dare say it is extensible to a makefile for any purpose at all.

Author: v. balaji (balaji@princeton.edu).

Users with access to `cobweb` should probably go to this GFDL-internal version of the mkmf manual [http://cobweb.gfdl.noaa.gov/~vb/mkmf.html].

Features of `mkmf` include:

- It understands dependencies in f90 (`modules` and `use`), the fortran `include` statement, and the cpp `#include` statement in any type of source;
- There are no restrictions on filenames, module names, etc.;
- It supports the concept of overlays (where source is maintained in layers of directories with a defined precedence);
- It can keep track of changes to `cpp` flags, and knows when to recompile affected source (i.e, files containing `#ifdefs` that have been changed since the last invocation);
- It will run on any unix platform that has perl version 5 installed;
- It is free, and released under GPL. External users can download the source here [http://www.gfdl.noaa.gov/~vb/mkmf/mkmf]. Current public revision is 14.0.

`mkmf` is pronounced *make-make-file* or *make-m-f* or even *McMuff* (Paul Kushner's suggestion).

mkmf user guide

The calling syntax is:

```
mkmf [-a abspath] [-c cppdefs] [-o otherflags] [-d] [-f] [-m makefile] [-p program] [-t template] [-v] [-x] [args]
```

- `-a abspath` attaches the `abspath` at the *front* of all *relative* paths to sourcefiles. If `-a` is not specified, the current working directory is the `abspath`.
- `cppdefs` is a list of `cpp #defines` to be passed to the source files: affected object files will be selectively removed if there has been a change in this state;
- `otherflags` is a list of compiler directives to be passed to the source files: it is rather similar to `cppdefs` except that these can be any flags. Also, by fortran convention, `cpp` is only invoked on `.F` and `.F90` files; `otherflags` apply to all source files (`.f` and `.f90` as well).
- `-d` is a debug flag to `mkmf` (much more verbose than `-v`, but probably of use only if you are modifying `mkmf` itself);
- `-f` is a formatting flag to restrict lines in the makefile to 256 characters. This was introduced in response to a customer who wanted to edit his makefiles using `vi`). Lines longer than that will use continuation lines as needed;
- `makefile` is the name of the makefile written (default `Makefile`);
- `template` is a file containing a list of make macros or commands written to the beginning of the makefile;
- `program` is the name of the final target (default `a.out`). A recent change in the behaviour of `mkmf` is that if `program` has the file extension `.a`, it is understood to be a library. The command to create it is `$(AR) $(ARFLAGS)` instead of `$(LD) $(LDFLAGS)`.
- `-v` is a verbosity flag to `mkmf`;
- `-x` executes the makefile immediately;
- `args` are a list of directories and files to be searched for targets and dependencies.

Makefile structure:

A *sourcefile* is any file with a source file suffix (currently `.F, .F90, .c, .f. .f90`). An *includefile* is any file with an include file suffix (currently `.H, .fh, .h, .h90, .inc`). A valid sourcefile can also be an includefile.

Each sourcefile in the list is presumed to produce an object file with the same basename and a `.o` extension in the current working directory. If more than one sourcefile in the list would produce identically-named object files, only the first is used and the rest are discarded. This permits the use of overlays: if `dir3` contained the basic source code, `dir2` contained bugfixes, and `dir1` contained mods for a particular run, `mkmf dir1 dir2 dir3` would create a makefile for correct compilation. Please note that precedence *descends* from left to right. This is the conventional order used by compilers when searching for libraries, includes, etc: left to right along the command line, with the first match invalidating all subsequent ones. See the Examples section for a closer look at precedence rules.

a closer look at precedence rules.

The makefile currently runs `$(FC)` on fortran files and `$(CC)` on C files. Flags to the compiler can be set in `$(FFLAGS)` or `$(CFLAGS)`. The final loader step executes `$(LD)`. Flags to the loader can be set in `$(LDFLAGS)`. Preprocessor flags are used by `.F`, `.F90` and `.c` files, and can be set in `$(CPPFLAGS)`. These macros have a default meaning on most systems, and can be modified in the template file. The predefined macros can be discovered by running `make -p`.

In addition, the macro `$(CPPDEFS)` is applied to the preprocessor. This can contain the `cpp #defines` which may change from run to run. `cpp` options that do not change between compilations should be placed in `$(CPPFLAGS)`.

Includefiles are recursively searched for embedded includes.

For `emacs` users, the make target `TAGS` is always provided. This creates a TAGS file in the current working directory with a cross-reference table linking all the sourcefiles. If you don't know about emacs tags, please consult the emacs help files! It is an incredibly useful feature.

The default action for non-existent files is to `touch` them (i.e create null files of that name) in the current working directory.

All the object files are linked to a single executable. It is therefore desirable that there be a single main program source among the arguments to `mkmf`, otherwise, the loader is likely to complain.

Treatment of [args]:

The argument list `args` is treated sequentially from left to right. Arguments can be of three kinds:

- If an argument is a sourcefile, it is added to the list of sourcefiles.
- If an argument is a directory, all the sourcefiles in that directory are added to the list of sourcefiles.
- If an argument is a regular file, it is presumed to contain a list of sourcefiles. Any line not containing a sourcefile is discarded. If the line contains more than one word, the last word on the line should be the sourcefile name, and the rest of the line is a file-specific compilation command. This may be used, for instance, to provide compiler flags specific to a single file in the sourcefile list:

```
a.f90
b.f90
f90 -Oaggress c.f90
```

This will add `a.f90, b.f90` and `c.f90` to the sourcefile list. The first two files will be compiled using the generic command `$(FC) $(FFLAGS)`. But when the make requires `c.f90` to be compiled, it will be compiled with `f90 -Oaggress`.

The directory `abspath` (specified by `-a`), or the current working directory, is always the first (and top-precedence) argument, even if `args` is not supplied.

Treatment of `[-c cppdefs]`:

The argument `cppdefs` is treated as follows. `cppdefs` should contain a comprehensive list of the `cpp #defines` to be preprocessed. This list is compared against the current "state", maintained in the file `.cppdefs` in the current working directory. If there are any changes to this state, `mkmf` will remove all object files affected by this change, so that the subsequent `make` will recompile those files. Previous versions of `mkmf` attempted to `touch` the relevant source, an operation that was only possible with the right permissions. The current version works even with read-only source.

The file `.cppdefs` is created if it does not exist. If you wish to edit it by hand (don't!) it merely contains a list of the `cpp` flags separated by blanks, in a single record, with no newline at the end.

`cppdefs` also sets the `make` macro `CPPDEFS`. If this was set in a template file and also in the `-c` flag to `mkmf`, the value in `-c` takes precedence. Typically, you should set only `CPPFLAGS` in the template file, and `CPPDEFS` via `mkmf -c`.

Treatment of includefiles:

Include files are often specified without an explicit path, e.g

```
#include "config.h"
```

or:

```
#include <config.h>
```

By convention, the first form will take a file of that name found in the *same* directory as the source, before looking at include directories specified by `-I`. The second form ignores the local directory and only uses `-I`.

local directory, and only uses -I.

`mkmf` does not currently distinguish between the two forms of `include`. It first attempts to locate the includefile in the same directory as the source file. If it is not found there, it looks in the directories listed as arguments, maintaining the same left-to-right precedence as described above.

This follows the behaviour of most f90 compilers: includefiles inherit the path to the source, or else follow the order of include directories specified from left to right on the `f90` command line, with the `-I` flags *descending* in precedence from left to right. It's possible there are compilers that violate the rule: if you come across any, please bring that to my attention.

If you have includefiles in a directory `dir` other than those listed above, you can specify it yourself by including `-Idir` in `$(FFLAGS)` in your template file. Includepaths in the template file take precedence over those generated by `mkmf`. (I suggest using `FFLAGS` for this rather than `CPPFLAGS` because fortran `includes` can occur even in source requiring no preprocessing).

Examples:

- The template file for the SGI MIPSpro compiler contains:

```
FC = f90
LD = f90
CPPFLAGS = -macro_expand
FFLAGS = -d8 -64 -i4 -r8 -mips4 -O3
LDFLAGS = -64 -mips4 $(LIBS)
LIST = -listing
```

The meaning of the various flags may be divined by reading the manual. A line defining the `make` macro LIBS, e.g:

```
LIBS = -lmpi
```

may be added anywhere in the template to have it added to the link command line.

- This example illustrates the effective use of `mkmf's` precedence rules. Let the current working directory contain a file named `path_names` containing the lines:

```
updates/a.f90
updates/b.f90
```

The directory `/home/src/base` contains the files:

```
a.f90
b.f90
c.f90
```

Typing

```
mkmf path_names /home/src/base
```

produces the following `Makefile`:

```
# Makefile created by mkmf $Id: mkmf.html,v 4.11 2001/05/28 03:23:50 vb Exp $


.DEFAULT:
        -touch $@
all: a.out
c.o: /home/src/base/c.f90
        $(FC) $(FFLAGS) -c      /home/src/base/c.f90
a.o: updates/a.f90
        $(FC) $(FFLAGS) -c      updates/a.f90
b.o: updates/b.f90
        $(FC) $(FFLAGS) -c      updates/b.f90
./c.f90: /home/src/base/c.f90
        cp /home/src/base/c.f90 .
./a.f90: updates/a.f90
        cp updates/a.f90 .
./b.f90: updates/b.f90
        cp updates/b.f90 .
SRC = /home/src/base/c.f90 updates/a.f90 updates/b.f90
OBJ = c.o a.o b.o
OFF = /home/src/base/c.f90 updates/a.f90 updates/b.f90
clean: neat
        -rm -f .cppdefs $(OBJ) a.out
neat:
        -rm -f $(TMPFILES)
localize: $(OFF)
        cp $(OFF) .
TAGS: $(SRC)
        etags $(SRC)
tags: $(SRC)
        ctags $(SRC)
```

```
a.out: $(OBJ)
        $(LD) $(OBJ) -o a.out $(LDFLAGS)
```

Note that when files of the same name recur in the target list, the files in the `updates` directory (specified in `path_names`) are used rather than those in the base source repository `/home/src/base`.

Assume that now you want to test some changes to `c.f90`. You don't want to make changes to the base source repository itself prior to testing; so you make yourself a local copy.

```
make ./c.f90
```

You didn't even need to know where `c.f90` originally was.

Now you can make changes to your local copy `./c.f90`. To compile using your changed copy, type:

```
mkmf path_names /home/src/base
make
```

The new Makefile looks like this:

```
# Makefile created by mkmf $Id: mkmf.html,v 4.11 2001/05/28 03:23:50 vb Exp $


.DEFAULT:
        -touch $@
all: a.out
c.o: c.f90
        $(FC) $(FFLAGS) -c      c.f90
a.o: updates/a.f90
        $(FC) $(FFLAGS) -c      updates/a.f90
b.o: updates/b.f90
        $(FC) $(FFLAGS) -c      updates/b.f90
./a.f90: updates/a.f90
        cp updates/a.f90 .
./b.f90: updates/b.f90
        cp updates/b.f90 .
SRC = c.f90 updates/a.f90 updates/b.f90
OBJ = c.o a.o b.o
OFF = updates/a.f90 updates/b.f90
clean: neat
        -rm -f .cppdefs $(OBJ) a.out
neat:
        -rm -f $(TMPFILES)
localize: $(OFF)
        cp $(OFF) .
TAGS: $(SRC)
        etags $(SRC)
tags: $(SRC)
        ctags $(SRC)
a.out: $(OBJ)
        $(LD) $(OBJ) -o a.out $(LDFLAGS)
```

Note that you are now using your local copy of `c.f90` for the compile, since the files in the current working directory always take precedence. To revert to using the base copy, just remove the local copy and run `mkmf` again.

- This illustrates the use of `mkmf -c`:

```
mkmf -c "-Dcppflag -Dcppflag2=2 -Dflag3=string ..."
```

will set `CPPDEFS` to this value, and also save this state in the file `.cppdefs`. If the argument to `-c` is changed in a subsequent call:

```
mkmf -c "-Dcppflag -Dcppflag2=3 -Dflag3=string ..."
```

`mkmf` will scan the source list for sourcefiles that make references to `cppflag2`, and the corresponding object files will be removed.

Caveats:

- In F90, the module name must occur on the same source line as the `module` or `use` keyword. That is to say, if your code contained:

```
use &
   this_module
```

it would confuse `mkmf`. Similarly, a fortran `include` statement must not be split across lines.

- Two `use` statements on the same line is not currently recognized, that is:

```
use module1; use module2
```

is to be avoided.

- I currently provide a default action for files listed as dependencies but not found: in this case, I `touch` the file, creating a null file of that name in the current directory. I am willing to debate the wisdom of this, if you are disturbed. But it is currently the least annoying way I've found to take care of a situation when cpp `#includes` buried within obsolete `ifdefs` ask for files that don't exist:

```
#ifdef obsolete
#include "nonexistent.h"
#endif
```

- If the formatting flag `-f` is used, long lines will be broken up at intervals of 256 characters. This can lead to problems if individual paths are longer than 256 characters.

TODO:

- An option to write a dependency graph, perhaps in HTML.

Please address all inquires to Author: v. balaji (balaji@princeton.edu).

---

created by v. balaji (balaji@princeton.edu) in emacs [http://www.gnu.org/software/emacs/] using the emacs-muse [http://www.mwolson.org/projects/MuseMode.html] mode.
last modified: 10 March 2011
this page visited: times