NMAKE

Definition :

- Microsoft development utility program for keeping a set of separately compiled files current, (AT&T and Lucent also maintain versions of nmake).
- Eliminates unnecessary compilations in large programs.
- Similar to the UNIX "make" command. NMAKE however maintains ***state*** information for future executions.
- Integrated into the Microsoft visual development environment.

Basic Operation :

- Reads a text file ("Makefile" in the current directory) that describes the relationships (dependencies) among all of the files that compose the program under development & the system commands required to recreate (compile, link) program files when changes have occurred.
- Queries the OpSys to determine which files have been altered since the last make (last time program was formed) occurred.
- Executes the commands to reform all files that are dependent upon the altered files.

Web References:

http://msdn.microsoft.com/library/devprods/vs6
/visualc/vcug/_asug_overview.3a_.nmake_reference.htm

http://www.bell-labs.com/project/nmake/tutorial/

Dependencies

- Makefile files are composed of dependency lines followed by indented (tab) commands to **recreate** the files.

Dependency line format

```
target files : prerequisite files
                    recreation command
                    ...      ...      ...
                    recreation command
```

- The "target files" is a blank separated list of files that are dependent upon the prerequisite file list specified after the colon, (the first target files' name must start in column 1).
- The recreation commands are any valid system commands, must be tab indented on consecutive lines immediately following the dependency lists. (Colon delimiting target & prerequisite files is required.)
- NMAKE executes the recreation commands if any of the target files have a date or time stamp that is older than any of the prerequisite files.
- NMAKE scans through source files to locate implicit prerequisites, such as header files in C++ programs.

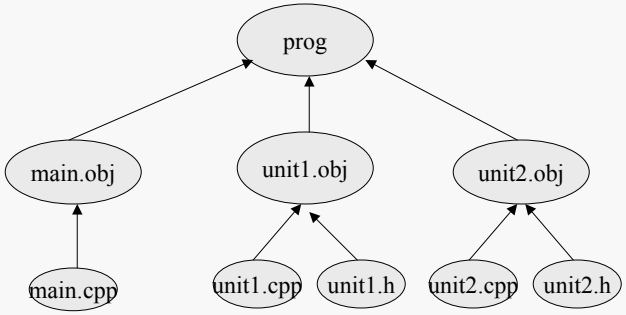
Simple Example


```
prog : main.obj unit1.obj unit2.obj
    cl /Feprog  main.obj unit1.obj unit2.obj
main.obj  : main.cpp
    cl /c main.cpp
unit1.obj : unit1.cpp unit1.h
    cl /c unit1.cpp
unit2.obj : unit2.cpp unit2.h
    cl /c unit2.cpp
```

- The first line gives the dependency of the executable image (prog) upon the object (.obj) files. The command lines calls ***cl*** (MS command line C++ compiler) to perform the linking.
- The following lines gives the object dependencies upon the source, and the ***cl*** commands to recompile the source files.
- If only one source file has been changed nmake will recompile only that file & then re-link the object files.
- Comments follow a sharp (#) on any line.
- A target file may follow make on the command if it is desired to remake only a portion of the system, otherwise make starts with the first dependency line.

Dependency Hierarchy

NMAKE builds an implicit dependency hierarchy for the system:



- The dependency hierarchy is checked by NMAKE to determine if a file is up-to-date when it is used in a prerequisite list.
- If prog must be recreated, it checks the dependency tree to determine if main.obj, unit1.obj & unit2.obj must be reformed first.

Implicit Dependency Rules

- make contains internal (automatic) rules for producing object files (.obj) from C language source files (.cpp).
- Given that the previous source files were cpp files then the makefile could be reduced taking advantage of the internal make rules:

```
prog : main.obj unit1.obj unit2.obj
    cl /Feprog  main.obj unit1.obj unit2.obj
main.obj  : main.cpp
unit1.obj : unit1.cpp unit1.h
unit2.obj : unit2.cpp unit2.h
```

- the commands to call the C compiler are unnecessary, since make can form them itself.
- make can be modified (or taught) the corresponding dependency rules for any language/utility

Variables & Macros

- make allows the user to assign strings to variables.

  syntax: variable = string
- a macro invocation or string usage occurs when the variable is preceded by a $ and enclosed in parenthesis.
- make replaces the variable with the string before executing the command.

Example:

```
SOURCE   = main.cpp unit1.cpp unit2.cpp
HEADERS  = main.h unit1.h unit2.h
OBJECTS = main.obj unit1.obj unit2.obj

prog : $(OBJECTS)
     cl /Feprog $(OBJECTS)

main.obj  : main.cpp
unit1.obj : unit1.cpp unit1.h
unit2.obj : unit2.cpp unit2.h

output : $(SOURCE) $(HEADERS) #print files
    print $(SOURCE)
    print $(HEADERS)
```


Predefined Variables

CC default value is the name of the system C compiler, cl

CFLAGS C compiler options, initially null, sometimes set to -O to optimize compilation

$< list of the prerequisite files that are out-of-date with respect to the target of the current rule.

The previous output dependency could be rewritten:

```
output : $(SOURCE) $(HEADERS) #print files
  print $<      #output altered files.
```

- note that the file output need not exist.
- if make encounters a nonexistent file it automatically executes the associated command sequence, but does not create the file.
- in order to prevent all the files from being printed each time make is executed, a dummy "output" file must be created in order to maintain a time & date stamp for make to check against.

Command Options

- commands can be prefixed with either '@' or '-'.
- when prefixed with '@', the command is executed, but not output to the screen.
- when prefixed with the '-', any error from the command is ignored and make continues execution, normally it stops when an error is returned. Useful when one wishes compilation to continue even though warnings & certain errors occur.

```
SOURCE   = main.cpp unit1.cpp unit2.cpp
HEADERS  = main.h unit1.h unit2.h
OBJECTS = main.obj unit1.obj unit2.obj
CFLAGS   = /FE


prog : $(OBJECTS)
     -cl $(CFLAGS)prog $(OBJECTS)


main.obj  : main.cpp
unit1.obj : unit1.cpp unit1.h
unit2.obj : unit2.cpp unit2.h


output : $(SOURCE) $(HEADERS) #print files
    print $<    #output altered files.


@echo compilation complete
```

The @echo suppresses printing of the echo command itself, but not the command's output.

Suffix & other rules

- a suffix rule describes how a file ending with a particular extension (e.g. .obj) is dependent upon a file with the same prefix, but a different suffix (e.g. .cpp).
- suffix rules allow make's internal implicit dependencies to be altered.

Suffix syntax:

```
.SUFFIXES  :  .ex1 .ex2
        .suffix1  .suffix2  :
                command(s)
```

- the first line adds the extensions to make's suffix.
- the second line specifies the command sequence required to form the "second file.ex2" from the first "file.ex1".

Other options

.IGNORE:

- causes make to ignore the return codes which signal errors from all commands. Equivalent to prefixing all commands with a hypen. Imbedded in the makefile.

NMAKE /N

- displays all commands but does not execute them. Useful for debugging the makefile itself.

NMAKE /F backup

- performs a make upon the specified file instead of the standard Makefile OR makefile.

Makefile Example


```
#Makefile for main.cpp, executable: prog.exe

# Ignore all error return codes
.IGNORE:

#Include .c & .cpp to the Suffix list
.SUFFIXES:  .c .cpp  .o .obj

#Define the CPP compiler & options
CPP  = cl
CFLAGS  = /FE

#Define all files in project
SOURCE   = main.cpp unit1.cpp unit2.cpp
HEADERS  = main.h unit1.h unit2.h
OBJECTS = main.obj unit1.obj unit2.obj


#recompile all object files in the current
directory that have changed

.cpp.obj .c.o :
    $(CPP) $(CFLAGS) $<


prog : $(OBJECTS) #recompile all source files
    $(CPP) $(CFLAGS)prog $(OBJECTS)


output : $(SOURCE) $(HEADERS) #print files
    print $<    #output altered files.


@echo compilation complete
```


Makefile Backup Example

```
#File: backup.mak
#       makefile to perform automatic backup
#       of all source files from current
directory #   to zip drive mounted as Z:

.SUFFIXES: .cpp .c .h .bak

# check if backup needs to be performed
# compare all source file time/date stamps
# to backup.bak time/date stamp
# backup.bak is a dummy 0 length file used to
# maintain the last backup time

#define backup file dependencies

.cpp.bak  .c.bak  .h.bak:

@echo insert the backup zip disk in the Z:
@echo drive in the next 5 secs!
@sleep 5

        copy $< Z:

#copy modified source files

        touch backup.bak       #update dummy
```

- The sleep and touch commands are UNIX utilities that have been ported to Windows by *Cygnus Solutions, a Red Hat company.*
- They are part of the GNU-Win32 package that may be downloaded from http://sources.redhat.com/cygwin/mirrors.html