

1 Introduction

- It a build system.
 - i) Automate the compilation and linking of source files into *executables*.
 - ii) Recompilation on only the changed portion of source code and the portion dependant on the changed code.
 - iii) Make maintaining the build system easier by avoiding redundant code in build system.
- Used by any compiled language.
 - C
 - C++
 - Fortran
 - Latex

1.1 Printing

- To print Text, one can use any of the following:

```
$(info This is an info text)
$(warning This is an warning text)
$(error This is an error text)
```

- Printing an `$(error)` will make the execution stop.

1.2 Variables

- Variables in MakeFile are defines as

```
VARNAME = VALUE
```

- They can be used with the help of dollar symbol (\$).

```
PIVALUE = 3.14
$(info Checking the Variable $(PIVALUE))
```

1.3 Build Rule

- Every rule starts with a target (which is going to be built) followed by colon (:) followed by the dependencies the target require.
- The next line may or may not contain commands to execute to get the dependencies.
- The dependencies may be
 - Files
 - Other Build Rules

1.3.1 Build rule depending on files

```
my_rule1 : source.c test.c
gcc -c source.c -o source.o
```

- In this case, the *my_rule1* build rule will run only when *source.c* and *test.c* are available.
- Even if *test.c* is not used in the command, since it is present as dependencies, *test.c* must be available for *my_rule* to be executed.

1.3.2 Build rule depending on other build rules

```
my_rule1 : my_rule2

my_rule2 : my_rule3
gcc source.o -o source.bin

my_rule3 : source.c
gcc -c source.c -o source.o
```

- In this case, *my_rule1* depends on *my_rule2* which in turn depends on *my_rule3* which in turn depends on *source.c*.
- Here, only *my_rule2* and *my_rule3* has commands and those executed, when their corresponding dependencies are met.

1.4 Build Rule with *wildcards*

- Here, *wildcards* are denoted by %.
- % denotes to any text.

```
%.o : %.c
```

- The above build rule indicates, to build anytext.o, then the dependency is the same-text.c.
- This can be used to build any *.o.

1.4.1 Make special variables

- Used inside the command below the build rule.
- @ takes the left hand side of build rule.
- ^ takes the right hand side of build rule.

```
source.o : source.c
gcc -c $^ -o $@
```

- The above lines are same as:

```
source.o : source.c
gcc -c source.c -o source.o
```

- These special variables along with wildcards can be used to create a generic building of *.o files from *.c files.

```
%.o : %.c
gcc -c $^ -o $@
```

- But, they can run independently without a target, hence

```
OBJFILES = test.o source.o

all : $(OBJFILES)

%.o : %.c
gcc -c $^ -o $@
```

- Above, for every object files needed by *all* build rule, the corresponding object files are created using *%.o* build rule.

1.5 Building

- Builing is done with *make*.
- This will run the first build rule and not *all* build rule.
- Only if the dependencies changes, the build rules changes and this is propagated to other build rules which uses the dependencies.
- Generarllly, MakeFile doesn't look for program oriented dependencies,like header files.
- @ symbol can be used to not to print the command.

2 Simple MakeFile Example

A sample example MakeFile used for building code inside *./Simple_MakeFileSystem* is shown below:

```
1  # A Simple MakeFile to build C programs
2
3  # C files to be compiled
4  CFILES = prog.c mod1.c mod2.C
5  OBJECTS = prog.o mod1.o mod2.o
6  EXEC = prog
7
8
9  # Compiler to use ; Can be changed for cross compilation
10 CC = gcc
11 # Include directories to be used
12 INCDIRS = ./Include
13 # C Flags to be given for compilation
14 CFLAGS = -Wall -Wextra -I $(INCDIRS)
15
16 # make command will run the first build rule, so generally the first one is all
17 all : $(EXEC)
18
19 # prog is the final executable to be created which isdepended on the object
20 → files to be available
21 $(EXEC) : $(OBJECTS)
22     $(CC) $^ -o $@
```

```

23 %.o : %.c
24     $(CC) $(CFLAGS) -c $^ -o $@
25
26 # The @ symbol means while running 'make clean', the command won't be shown
27 clean:
28     @rm -rf $(OBJECTS) $(EXEC)
29
30
31 # added test to run the program
32 test : all
33     $(info Running the program ...)
34     $(info )
35     ./$$(EXEC)

```

2.1 Disadvantages

- I) The *make* command recompiles the code where there is a change in .c files as only those are the depended for creating .o files.
 - But, when only the header content changes the command doesn't compile.
- II) Also, the include directory is limited to one.
- III) In this, we have to give the .c and .o files.

3 Solving 2.1 - I)

- The particular problem, of header file not being seen can be solved by generating files (.d) that encodes make rule for .h dependencies.
- This can be done by adding the flags to the GCC compiler

```
-MP -MD
```

- Now, this will create .d files for the .c files which has depend over header files so as to be used by make build system.
- We also need to clean the .d files.
- Now, the new MakeFile is shown below:

```

1  # A Simple MakeFile to build C programs
2
3  # C files to be compiled
4  CFILES = prog.c mod1.c mod2.C
5  OBJFILES = prog.o mod1.o mod2.o
6  DEFILES = prog.d mod1.d mod2.d
7  EXEC = prog
8
9
10 # Compiler to use ; Can be changed for cross compilation
11 CC = gcc
12 # Include directories to be used
13 INC_DIRS = ./Include
14 # C Flags to be given for compilation
15 CFLAGS = -Wall -Wextra -I $(INC_DIRS) -MD -MP
16

```

```

17  # make command will run the first build rule, so generally the first one is all
18  all : $(EXEC)
19
20  # prog is the final executable to be created which is depended on the object
→  files to be available
21  $(EXEC) : $(OBJFILES)
22      $(CC) $^ -o $@
23
24  %.o : %.c
25      $(CC) $(CFLAGS) -c $^ -o $@
26
27  # The @ symbol means while running 'make clean', the command won't be shown
28  clean:
29      @rm -rf $(OBJFILES) $(DEPFILES)
30
31  # The @ symbol means while running 'make clean all', the command won't be shown
32  cleanall: clean
33      @rm -rf $(EXEC)
34
35  # added test to run the program
36  test : all
37      $(info Running the program ...)
38      $(info )
39      ./$$(EXEC)

```

4 Solving 2.1 - II)

- += is used to append values to a variable.
- To solve the issue of multiple include directories, we use foreach loop whose syntax is:

```
$(foreach var, range, cmds)
```

- An example to print numbers from 1 to 4 using foreach loop:

```
range = 1 2 3 4
$(foreach i, $(range), $(info $(i)))
```

- Using the above information, now the make file can be made to have multiple include directories:

```

1  # A Simple MakeFile to build C programs
2
3  # C files to be compiled
4  CFILES = prog.c mod1.c mod2.c
5  OBJFILES = prog.o mod1.o mod2.o
6  DEPFILES = prog.d mod1.d mod2.d
7  EXEC = prog
8
9
10 # Compiler to use ; Can be changed for cross compilation
11 CC = gcc
12 # Include directories to be used
13 INC_DIRS += .
14 INC_DIRS += ./Include

```

```

15  # C Flags to be given for compilation - (adding foreach for iterating through
    ↪  each include directory
16  CFLAGS = -Wall -Wextra $(foreach includedir, $(INCDIRS), -I $(includedir)) -MD
    ↪  -MP
17
18  # make command will run the first build rule, so generally the first one is all
19  all : $(EXEC)
20
21  # prog is the final executable to be created which is depended on the object
    ↪  files to be available
22  $(EXEC) : $(OBJFILES)
23      $(CC) $^ -o $@
24
25  %.o : %.c
26      $(CC) $(CFLAGS) -c $^ -o $@
27
28  # The @ symbol means while running 'make clean', the command won't be shown
29  clean:
30      @rm -rf $(OBJFILES) $(DEPFILES)
31
32  # The @ symbol means while running 'make clean all', the command won't be shown
33  cleanall: clean
34      @rm -rf $(EXEC)
35
36  # added test to run the program
37  test : all
38      $(info Running the program ...)
39      $(info )
40      ./$$(EXEC)

```

5 Solving 2.1 - III)

- The problem of finding .c files in the directories can be solved initializing the C directories to look for.
- Then, using foreach loop and wildcards, the .c files can be found using

```

CDIRS += .
CDIRS += ./src

CFILES = $(foreach cdir, $(CDIRS), $(wildcard $(cdir)/*.c))

$(info $(CFILES))

```

- This will produce the output as:

```
./prog.c ./src/mod2.c ./src/mod1.c
```

- The object files to be created for each of the .c files in the same location so we use patsubst to get the .o files.

```

OBJFILES = $(patsubst %.c, %.o, $(CFILES))

$(info $(OBJFILES))

```

- This will produce the output as:

```
./prog.o ./src/mod2.o ./src/mod1.o
```

- Now, the complete Make file will all the Disadvantages solved is given below:

```

1  # A Simple MakeFile to build C programs
2
3
4  # Include directories to be used
5  INCDIRS += .
6  INCDIRS += ./Include
7
8  # C directories
9  CDIRS += .
10 CDIRS += ./src
11
12 # C files to be compiled is found by using wildcard to get the c files
13 CFILES = $(foreach cdir, $(CDIRS), $(wildcard $(cdir)/*.c))
14 # o files are obtained by substituting .c with .o using pathsubst in CFILES
15 OBJFILES = $(patsubst %.c, %.o, $(CFILES))
16 # d files are obtained by substituting .c with .d using pathsubst in CFILES
17 DEPFILES = $(patsubst %.c, %.d, $(CFILES))
18 EXEC = prog
19
20
21 # Compiler to use ; Can be changed for cross compilation
22 CC = gcc
23 # C Flags to be given for compilation - (adding foreach for iterating through
24 → each include directory
25 CFLAGS = -Wall -Wextra $(foreach includedir, $(INCDIRS), -I $(includedir)) -MD
26 → -MP
27
28 # make command will run the first build rule, so generally the first one is all
29 all : $(EXEC)
30
31 # prog is the final executable to be created which isdepended on the object
32 → files to be available
33 $(EXEC) : $(OBJFILES)
34     $(CC) $^ -o $@
35
36 %.o : %.c
37     $(CC) $(CFLAGS) -c $^ -o $@
38
39 # The @ symbol means while running 'make clean', the command won't be shown
40 clean:
41     @rm -rf $(OBJFILES) $(DEPFILES)
42
43 # The @ symbol means while running 'make clean all', the command won't be shown
44 cleanall: clean
45     @rm -rf $(EXEC)

```

```
44     # added test to run the program
45     test : all
46         $(info Running the program ...)
47         $(info )
48         ./${EXEC}
```
