MAKEFILE

Table of Content

Introduction

- We need a file called a *makefile* to tell make what to do.
- Makefile tells make how to compile and link a program.

Why Makefile ?

• Make checks timestamps to see what has changed and rebuilds just what we need, without rebuilding other files.

• Manage several source files and compile them quickly with a single command line

What a Rule Looks Like

Makefile contains a set of rules to build

It consists of 3 parts-

Target*: *prerequisites

<tab> recipe

<tab> …

- A **target** can also be the name of an action to carry out, such as ‘clean’
- A ***prerequisite*** is input used to create the target. Dependencies to be checked before creating the target.
- A ***recipe*** is an action that make carries out. A recipe may have more than one command, either on the same line or each on its own line.

[Please note: We need to put a tab character at the beginning of every recipe line!]

For example,

```
clean:
<tab> echo "CLEAN SOURCE FILES"
<tab> rm -rf abc.v xyz.v
```

- Prerequisites or dependencies of a target can be names of other targets.

For example:

```
compile:
        compile-command
sim:
        simulation-command

build_tb: compile
            simulation-command
 [or]
run_tb: compile sim
```

How make Processes a Makefile

- By default, make starts with the first target (not targets whose names start with ‘.’ unless they also contain one or more ‘/’). This is called the *default goal*.
- The command “ make” reads the makefile in the current directory and begins by processing the first rule.

Makefile Rules to split lines

- Makefiles use a “line-based” syntax in which the newline character is special and marks the end of a statement
- For readability, we can break line by adding a backslash (\) character.

Splitting Without Adding Whitespace

To split a line but do *not* want any whitespace added- replace your backslash/newline pairs with the three characters dollar sign **“$”**, backslash “\”, and newline:

```
var: one$\
     word
```

After make removes the backslash/newline, this is equivalent to:

```
var: one$ word
```

Then make will perform variable expansion.

‘$’ refers to a variable with the one-character name “ ” (space) which does not exist and so expands to the empty string

var: oneword

[dollar signs are used to start make variable references, if you really want a dollar sign in a target or prerequisite you must write two of them: **$$]**

Splitting Recipe Lines

Output

```
all :       echo no\
space
             echo no\
             space
             echo one \
             space
             echo one\
              space
```

```
nospace

nospace

one space

one space
```

```
abc:  echo 'hello \
world' ; echo "hello \
 world"
```

```
hello \ world

hello world
```

Note: Two commands separated by semicolon behave much like two separate commands.

Using Variables in Recipes

Output

```
HELLO = 'hello \ world’

all :  echo $(HELLO)
```

```
hello world
```

```
LIST = one two three
all:
		for i in $(LIST); do \
		echo $$i; \
		done
```

```
one
two
three
```

How to Use Variables

- A *variable* is a name defined in a makefile to represent a string of text, called the variable’s *value*.

- A variable name may be any sequence of characters

  Excluding ‘:’, ‘#’, ‘=’, or whitespace.

- Variable names are case-sensitive.

  For eg: ALL, All, all represent different variables.

Two Flavors of Variables

The flavors are distinguished in how they handle the values they are assigned in the makefile, and in how those values are managed when the variable is later used and expanded.

Recursive Assignment	Setting recursively expanded variables.
Simple Assignment	Setting simply expanded variables.

Recursively Expanded Variable Assignment

Variables of this sort are defined by lines using ‘**=**’
If the specified value contains references to other variables, these references are expanded whenever this variable is substituted. When this happens, it is called ***recursive expansion***.

Example:

```
abc = ${pqr}
pqr = ${xyz}
xyz = alphabets

all: echo ${abc}
```

Output

```
alphabets
```

Simply Expanded Variable Assignment

Variables of this sort are defined by lines using ‘:=’ or ‘::=’
The value of a simply expanded variable is scanned once, expanding any references to other variables and functions, when the variable is defined. Once that expansion is complete the value of the variable is never expanded again.
If the value contained variable references the result of the expansion will contain their values *as of the time this variable was defined*. This is called ***simple expansion***.

Example:

```
abc := pqr
pqr := ${abc} xyz
abc := alphabets

all: echo ${pqr}
     echo ${abc}
```

Output

```
pqr xyz
alphabets
```

Appending More Text to Variables

- To add more text to the value of a variable already defined by using ‘**+=**’
- When the variable in question has not been defined before, ‘+=’ acts just like normal ‘=’ and ‘:=’

Example:

```
alphabets := abc
alphabets += xyz


all: echo ${alphabets}
```

Or

```
alphabets = abc
alphabets = ${alphabets} xyz
```

Output

```
abc xyz
```

Defining Multi-Line Variables

Another way to set the value of a variable is to use the define directive define directive is followed on the same line by the name of the variable being defined.

Example:

```
define alphabets
echo abc
echo ${xyz}
endef
```

Equivalent to:

```
alphabets = echo abc; echo ${xyz}
```

Conditional Parts of Makefiles

- A *conditional* directive causes part of a makefile to be obeyed or ignored depending on the values of variables.
- Syntax:

```
conditional-directive
text-if-true
else
text-if-false
endif
```

Example of conditional using three directives: ifeq, else and endif-

```
abc = def
xyz = def

ifeq (${abc},${xyz})
echo “match”
else
echo “no match”
endif
```

ifdef *variable-name*

The ifdef form takes the *name* of a variable as its argument
If the value of that variable has a non-empty value, the *text-if-true* is effective; otherwise, the *text-if-false*, if any, is effective.

Examples:

(A)

```
abc = def
xyz = ${abc}

ifdef ${xyz}
echo “true”
else
echo “false”
endif
```

(B)

```
abc =
xyz = ${abc}
```

(C)

```
xyz =
```

Output: (A)true (B)true (C)false

Difference between := , =

- x = abc
- y = $(x) bar
- x = pqr #The value of y would be pqr bar

- x := abc
- y := $(x) bar
- x := pqr #The value of y would be abc bar

Execute a make command

- First, change directory to the folder where Makefile is present

- Giving the following command, make reads the makefile in the current directory and begins by processing the first rule

```
make
```

- If you have several makefiles, then execute them with command:

```
make [-f filename]
```

- To execute a particular target from a Makefile:

```
make [targetname]
          [Or]
make [-f filename] [targetname]
```