# Dum Ka Biryani, Make for each other

Version 1.0
February 2011
GNU Free Documentation License

Shakthi Kannan
shakthimaan@gmail.com
http://www.shakthimaan.com

# Dum Ka Biryani

# 2.1 What a Rule Looks Like

**Makefile**

```
target : prerequisites ...
     recipe
     ...
     ...
```

# Tab before recipe!

# 2.2 A Simple Makefile

## Makefile

```
biryani : masala.o rice.o onion.o curd.o coriander.o meat.o
	cc -o biryani masala.o rice.o onion.o curd.o coriander.o meat.o

masala.o : masala.c masala.h defs.h
	cc -c masala.c

rice.o : rice.c rice.h defs.h
	cc -c rice.c

onion.o : onion.c defs.h
	cc -c onion.c

curd.o : curd.c defs.h
	cc -c curd.c

coriander.o : coriander.c defs.h
	cc -c coriander.c

meat.o : meat.c defs.h
	cc -c meat.c

clean:
	rm biryani \
		masala.o rice.o onion.o curd.o coriander.o meat.o
```

# 2.4 Variables Make Makefiles Simpler

**Makefile**

```
objects = masala.o rice.o onion.o curd.o coriander.o meat.o

biryani : $(objects)
	cc -o biryani $(objects)

masala.o : masala.c masala.h defs.h
	cc -c masala.c

rice.o : rice.c rice.h defs.h
	cc -c rice.c

onion.o : onion.c defs.h
	cc -c onion.c

curd.o : curd.c defs.h
	cc -c curd.c

coriander.o : coriander.c defs.h
	cc -c coriander.c

meat.o : meat.c defs.h
	cc -c meat.c

clean:
	rm biryani $(objects)
```

# 2.5 Letting make Deduce the Recipes

**Makefile**

```
objects = masala.o rice.o onion.o curd.o coriander.o meat.o

biryani : $(objects)
        cc -o biryani $(objects)

masala.o : masala.h defs.h

rice.o : rice.h defs.h

onion.o : defs.h

curd.o : defs.h

coriander.o : defs.h

meat.o : defs.h

clean:
        rm biryani $(objects)
```

# 2.6 Another Style of Makefile

## Makefile

```
objects = masala.o rice.o onion.o curd.o coriander.o meat.o

biryani : $(objects)
	cc -o biryani $(objects)

$(objects) : defs.h

masala.o : masala.h

rice.o : rice.h

clean:
	rm biryani $(objects)
```

# 2.7 Rules for Cleaning the Directory

## Makefile

```
objects = masala.o rice.o onion.o curd.o coriander.o meat.o

biryani : $(objects)
	cc -o biryani $(objects)

$(objects) : defs.h

masala.o : masala.h

rice.o : rice.h

.PHONY : clean
clean:
	-rm biryani $(objects)
```

# 3.1 What Makefiles Contain

* Explicit rule

```
onion.o : onion.c defs.h
        cc -c onion.c
```

* Implicit rule

```
masala.o : masala.h
```

* Variable definition

```
objects = masala.o rice.o onion.o curd.o coriander.o meat.o
```

* Directive

```
ifeq ($(CC), gcc)
        libs=$(libs_for_gcc)
else
        libs=$(normal_libs)
endif
```

* Comments

```
# This is a comment
```

# 3.2 What Name to Give Your Makefile

**Default**

* GNUMakefile
* makefile
* Makefile

# 3.2 What Name to Give Your Makefile

**Default**

* GNUMakefile
* makefile
* Makefile

**$ make**

```
$ make -f myMakefile
$ make --file=myMakefile
```

# 3.3 Including Other Makefiles

* Syntax

  include *filenames...*

* Regex and variables allowed

  include foo *.mk $(bar)

* To ignore Makefile if not found

  -include *filenames...*

# 3.3 Including Other Makefiles

* Syntax

  include *filenames...*

* Regex and variables allowed

  include foo *.mk $(bar)

* To ignore Makefile if not found

  -include *filenames...*

**$ make**

```
$ make -I
$ make --include-dir
```

# 3.6 Overriding Part of Another Makefile

## GNUmakefile

```
meat:
	cc -c vegetables.c

%: force
	@$(MAKE) -f Makefile $@
force: ;
```

# 3.6 Overriding Part of Another Makefile

## GNUmakefile

```
meat:
	cc -c vegetables.c

%: force
	@$(MAKE) -f Makefile $@
force: ;
```

## $ make

```
$ make meat
```

# 3.7 How make Reads a Makefile

**First phase**

* Reads all makefiles
* Reads included makefiles
* Internalize variables, values, implicit and explicit rules
* Constructs dependency graph of targets and pre-requisites
* *immediate*: expansion in phase one

**Second phase**

* Determines which targets need to be re-built
* Invokes the necessary rules
* *deferred*: expansion in phase two

# 3.7 How make Reads a Makefile

**Variable assignment**

* immediate = deferred
* immediate ?= deferred
* immediate := immediate
* immediate += deferred or immediate

  immediate, if variable was previously set as a simple variable (:=) deferred, otherwise

**Second phase**

* Determines which targets need to be re-built
* Invokes the necessary rules
* *deferred*: expansion in phase two

# 3.7 How make Reads a Makefile

## Conditional Directives

* immediate

## Rule Definition

### Makefile

```
target : prerequisites ...
     recipe
     ...
     ...
```

### Makefile

```
immediate : immediate ; deferred
     deferred
```

# 4.2 Rule Syntax

* First target as default, if not specified.
* A rule tells make two things: when the targets are out of date (prerequisites), and how to update them (recipe) when necessary.

## Makefile

```
target : prerequisites ...
     recipe
     ...
     ...
```

## Makefile

```
target : prerequisites ; recipe
     recipe
     ...
     ...
```

# 4.4 Using Wildcard Characters in File Names

**Makefile**

```
clean :
	rm -f *.o
```

**Wildcard expansion does not happen in variable definition**

**Makefile**

```
objects = *.o
```

**Use Wildcard function!**

**Makefile**

```
objects = $(wildcard *.o)
```

# 4.5 Searching Directories for Prerequisites

**VPATH make variable**

VPATH = src:../headers

**vpath Directive**

**Makefile**

```
vpath pattern directories
vpath %.h ../headers
```

**To clear search paths:**

**Makefile**

```
vpath pattern
vpath %.h

vpath
```

# 4.5.4 Writing Recipes with Directory Search

**Automatic variables**

| | |
|---|---|
| $^ | all prerequisites |
| $@ | target |
| $< | first prerequisite |

**Makefile**

```
coriander.o : coriander.c
	cc -c $(CFLAGS) $^ -o $@
```

**Makefile**

```
masala.o : masala.c masala.h defs.h
	cc -c $(CFLAGS) $< -o $@
```

# 4.6 Phony Targets

**Error in submake ignored:**

## Makefile

```
SUBDIRS = masala rice onion

subdirs :
	for dir in $(SUBDIRS); do \
		$(MAKE) -C $$dir; \
	done
```

## Makefile

```
SUBDIRS = masala rice onion

.PHONY : subdirs $(SUBDIRS)
subdirs: $(SUBDIRS)

$(SUBDIRS):
	$(MAKE) -C $@
```

# 4.12 Static Pattern Rules

* Override implicit rule.
* No uncertainity.

## Makefile

```
targets ... : target-pattern: prereq-patterns ...
      recipe
      ...
```

## Makefile

```
objects = curd.o coriander.o masala.o

all: $(objects)

$(objects): %.o: %.c
      $(CC) -c $(CFLAGS) $< -o $@
```

# 5.1 Recipe Syntax

* Follows shell syntax.
* Backslash-newline pairs are preserved and passed to the shell.

## Makefile

```
all :
      @echo In a cooking vessel\
add a layer of semi-cooked Basmati rice
      @echo Add meat on this\
      rice layer
      @echo Add another layer \
            of rice
      @echo Sprinkle the ingredients with water and cook
```

# 5.1 Recipe Syntax

* Follows shell syntax.
* Backslash-newline pairs are preserved and passed to the shell.

## Makefile

```
all :
      @echo In a cooking vessel\
add a layer of semi-cooked Basmati rice
      @echo Add meat on this\
      rice layer
      @echo Add another layer \
            of rice
      @echo Sprinkle the ingredients with water and cook
```

## $ make

```
In a cooking vessel add a layer of semi-cooked Basmati rice
Add meat on this rice layer
Add another layer of rice
Sprinkle the ingredients with water and cook
```

# 5.1.2 Using Variables in Recipes

## Makefile

```
LIST = masala rice onion

all:
	for i in $(LIST); do \
	    echo $$i; \
	done
```

# 5.1.2 Using Variables in Recipes

## Makefile

```
LIST = masala rice onion

all:
	for i in $(LIST); do \
		echo $$i; \
	done
```

## $ make

```
masala
rice
onion
```

# 5.4 Parallel Execution

* -j number, number of recipies to execute in parallel.
* -l number, number (limit) of jobs to run in parallel.

# 5.4 Parallel Execution

* -j number, number of recipies to execute in parallel.
* -l number, number (limit) of jobs to run in parallel.

**$ make**

```
$ make -j
$ make --jobs

$ make -l
$ make --max-load
```

# 5.5 Errors in Recipes

* Ignore errors in a recipe line.

**Makefile**

```
clean :
	-rm -f *.o
```

# 5.7 Recursive Use of make

* Recursive make commands should always use the variable MAKE.

**Makefile**

```
subsystem :
	cd subdir && $(MAKE)
```

**Makefile**

```
subsystem :
	$(MAKE) -C subdir
```

# 5.7.2 Communicating Variables to Sub-make

## Makefile

```
variable = value

export variable ...

unexport variable ...
```

## Makefile

```
export variable = value

export variable := value

export variable += value
```

# 5.8 Defining Canned Recipes

**Makefile**

```
define mix-masala=
@echo "Adding green chillies"
@echo "Adding ginger garlic paste"
@echo "Adding cinnamon, cardamom"
@echo "Adding fried onions"
@echo "Adding coriander and mint leaves"
endef

mix:
        $(mix-masala)
```

# 5.8 Defining Canned Recipes

## Makefile

```
define mix-masala=
@echo "Adding green chillies"
@echo "Adding ginger garlic paste"
@echo "Adding cinnamon, cardamom"
@echo "Adding fried onions"
@echo "Adding coriander and mint leaves"
endef

mix:
        $(mix-masala)
```

## $ make

```
$ make mix
Adding green chillies
Adding ginger garlic paste
Adding cinnamon, cardamom
Adding fried onions
Adding coriander and mint leaves
```

# 6.2 The Two Flavors of Variables

* Recursively Expanded variable

**Makefile**

```
spice = $(chillies)
chillies = $(both)
both = green chillies and red chillie powder

all: ; echo $(spice)
```

# 6.2 The Two Flavors of Variables

* Recursively Expanded variable

**Makefile**

```
spice = $(chillies)
chillies = $(both)
both = green chillies and red chillie powder

all: ; echo $(spice)
```

**$ make**

```
green chillies and red chillie powder
```

# 6.2 The Two Flavors of Variables

* Simply Expanded variable

**Makefile**

```
chillie := red chillie
spice := $(chillie) powder, green chillies
chillie := green chillies

all:
	echo $(spice)
	echo $(chillie)
```

# 6.2 The Two Flavors of Variables

* Simply Expanded variable

**Makefile**

```
chillie := red chillie
spice := $(chillie) powder, green chillies
chillie := green chillies

all:
	echo $(spice)
	echo $(chillie)
```

**$ make**

```
red chillie powder, green chillies
green chillies
```

# 6.2 The Two Flavors of Variables

* Conditional Variable assignment operator

**Makefile**

```
ifeq ($(origin RICE), undefined)
     RICE = Basmati rice
endif
```

**Makefile**

```
RICE ?= Basmati rice
```

# 6.3.1 Substitution References

**Makefile**

```
masala := chillies.o onions.o garlic.o ginger.o
biryani := $(masala:.o=.c)
```

**Makefile**

```
masala := chillies.o onions.o garlic.o ginger.o
biryani := $(masala:%.o=%.c)
```

* 'biryani' is set to 'chillies.c onions.c garlic.c ginger.c'

# 6.3.2 Computed Variable Names

## Makefile

```
greeny = coriander
coriander = green coriander leaves
leaves := $($(greeny))

all:
	@echo $(leaves)
```

# 6.3.2 Computed Variable Names

## Makefile

```
greeny = coriander
coriander = green coriander leaves
leaves := $($(greeny))

all:
	@echo $(leaves)
```

## $ make

```
green coriander leaves
```

# 6.6 Appending More Text to Variables

## Makefile

```
objects = masala.o rice.o onion.o curd.o
...
objects += coriander.o
```

## Makefile

```
CFLAGS = $(includes) -O
...
CFLAGS += -pg
```

# 6.7 The override Directive

## Makefile

```
override variable = value

override variable := value

override variable += more text
```

## Makefile

```
override CFLAGS += -g

override define masala=
more-masala
endef
```

# 6.9 Undefining Variables

## Makefile

```
water := saffron water
leaves = mint leaves

undefine water
undefine leaves
```

## Makefile

```
override undefine CFLAGS
```

# 6.11 Target-specific Variable Values

**Makefile**

```
target ... : variable-assignment
```

**Makefile**

```
prog : CFLAGS = -g
prog : masala.o rice.o onion.o
```

# 6.12 Pattern-specific Variable Values

## Makefile

```
pattern ... : variable-assignment
```

## Makefile

```
%.o : CFLAGS = -O
```

## Makefile

```
%.o: %.c
	$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@

lib/%.o: CFLAGS := -fPIC -g
%.o: CFLAGS := -g

all: rice.o lib/masala.o
```

# 7.1 Example of a Conditional

## Makefile

```
libs_for_gcc = -lgnu

normal_libs =

biryani : $(objects)
ifeq ($(CC), gcc)
     libs=$(libs_for_gcc)
else
     libs=$(normal_libs)
endif
```

# 7.2 Syntax of Conditionals

**Makefile**

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

**Makefile**

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

**Makefile**

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

# 7.2 Syntax of Conditionals

**Four conditional directives:**

* ifeq

  ifeq ($(CC), gcc)

* ifneq

  ifneq ($(STRIP), strip)

* ifdef

  ifdef $(LIBS)

* ifndef

  ifndef $(CFLAGS)

# 8.1 Function Call Syntax

## Makefile

```
$(function arguments)

${function arguments}
```

## Makefile

```
comma := ,
empty :=
space:= $(empty) $(empty)

masala:= chillies onion garlic ginger
list:= $(subst $(space),$(comma),$(masala))
```

**'list' is now 'chillies,onion,garlic,ginger'**

# 8.2 Functions for String Substitution and Analysis

## Makefile

```
$(subst from,to,text)

$(patsubst pattern,replacement,text)

$(strip string)

$(findstring find,in)

$(filter pattern...,text)

$(filter-out pattern...,text)

$(sort list)

$(word n, text)
```

# 8.3 Functions for File Names

## Makefile

```
$(dir names...)

$(notdir names...)

$(suffix names...)

$(basename names...)

$(addsuffix suffix,names...)

$(addprefix prefix,names...)

$(join list1, list2)

$(wildcard pattern)
```

# 8.4 Functions for Conditionals

## Makefile

```
$(if condition,then-part[,else-part])

$(or condition1[,condition2[,condition3...]])

$(and condition1[,condition2[,condition3...]])
```

# 8.5 The foreach Function

## Makefile

```
$(foreach var,list,text)
```

## Makefile

```
find_files = $(wildcard $(dir)/*)

dirs := masala rice leaves

files := $(foreach dir, $(dirs), $(find_files))
```

# 8.6 The call Function

## Makefile

```
$(call variable,param,param,...)
```

## Makefile

```
reverse = $(2) $(1)

make = $(call reverse,cook,mix)
```

**'make' contains 'mix cook'**

# 8.7 The value Function

## Makefile

```
$(value variable)
```

## Makefile

```
LIST = $PATH

all:
	@echo $(LIST)
	@echo $(value LIST)
```

# 8.7 The value Function

## Makefile

```
$(value variable)
```

## Makefile

```
LIST = $PATH

all:
	@echo $(LIST)
	@echo $(value LIST)
```

## $ make

```
ATH
/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin
```

# 8.9 The origin Function

## Makefile

```
$(origin variable)
```

**How the variable was defined:**

* undefined
* default
* environment
* environment override
* file
* command line
* automatic

# 8.10 The flavor Function

**Makefile**

```
$(flavor variable)
```

**Flavor of the variable:**

* undefined
* recursively expanded variable
* simply expanded variable

# 8.11 The shell Function

## Makefile

```
contents := $(shell cat onion.c)
```

## Makefile

```
files := $(shell echo *.c)
```

# References

GNU Make:
http://www.gnu.org/software/make/

GNU Make Manual:
http://www.gnu.org/software/make/manual/

Dum Ka Biryani, Make for each other (sources):
http://www.shakthimaan.com/downloads.html#dum-ka-biryani-make-for-each-other

Symbols in LaTeX and HTML:
http://newton.ex.ac.uk/research/qsystems/people/sque/symbols/