Make and Autoconf Framework

Vineet Seth

15th Feburary 2006

Outline

Using Implied Rules

t.c

```
#include <stdio.h>
int
print_msg(FILE *file, char *msg)
{
  fprintf(file, msg);
}
int
main(int argc, char **argv)
{
  char str[] = "Hello World\n";

  print_msg(stdout, str);

}
```

C file for testing : t.c

$ make t

Compilation on basis of Implied rule
Good for single file compilation

Using Makefile

- Order of search : GNUmakefile, makefile, Makefile
- Can contain 5 things:
  1. Explicit Rule
  2. Implicit Rule
  3. Variable Definition
  4. Directives
  5. Comments

Makefile Structure

Specifying Rule

target . . . : Prerequisites [; command]

<tab> command

<tab> command

<tab> . . .

- First target is the default Target
- Other target have to be explicitly executed e.g

  $ make clean

Use of Tabs is mandatory

Specifying Variable

Makefile is processed in 2 phases

Variables assignment, can be immediate or Deferred until later This is controlled though use of :

- immediate = deferred
- immediate ?= deferred
- immediate := immediate
- immediate += deferred or immediate

Referring to Variables

Makefile variables are referred by using $(variable) form

Shell variables are referred by using ${variable} form

C code

msg.c

```
#include <stdio.h>
#include <msg.h>
int
print_msg(FILE *file, char *msg) {
   fprintf(file, msg);
}
```

msg.h

```
#ifndef HAVE_MSG_H
#define HAVE_MSG_H
int print_msg(FILE *, char *);
#endif
```

hello.c

```
#include <stdio.h>
#include <msg.h>
int
main(int argc, char **argv) {
  char str[] = "Hello World\n";
  print_msg(stdout, str);
}
```

bye.c

```
#include <stdio.h>
#include <msg.h>
int
main(int argc, char **argv) {
  char str[] = "Bye Bye!!!\n";
  print_msg(stdout, str);
}
```

Makefile forms

Simple Makefile with Implied rules

```
CFLAGS = -I.
files = bye.c hello.c msg.c
objects =  $(patsubst %.c,%.o, $(files))
bins = hello bye
all: $(bins)

hello: hello.o msg.o
bye: bye.o msg.o

$(objects): $(files) msg.h

.PHONY : clean
clean:
        @-rm $(bins) $(objects)
```

1. `Pattern substitution $(patsubst from,to, input)`
2. '-' is for ignoring errors
3. '@' is the silent form

Makefile forms

Simple Makefile with Explicit rules

```
CFLAGS = -I.
files = bye.c hello.c msg.c
objects =  $(patsubst %.c,%.o, $(files))
bins = hello bye

all: $(bins)

hello: hello.o msg.o
	$(CC) $(CFLAGS) -o $@ $^

bye: bye.o msg.o
	$(CC) $(CFLAGS) -o $@ $^

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

.PHONY : clean
clean:
	@-rm $(bins) $(objects)
```

Make specification

- Automatic Variables

Variable	Meaning
$@	All the Targets
$<	First prerequisite
$?	All new prerequisites
$^	All prerequisites

- Default variables
  1. Name of Programs e.g CC, AR, AS, CXX,
  2. Arguments of Programs e.g CFLAGS, LDFLAGS, ASFLAGS, CPPFLAGS, CXXFLAGS
- Exported variables
  1. Auto-exported variables e.g SHELL, MAKEFLAGS
  2. Explicitly exported variables using `export`, `unexport`

Recursive use of Make

For Larger project

1. Makefile exists in Multiple directories
2. Top level Makefile controls the calling of Makefile
3. $(MAKE) -C <subdir> is generally used

```
SUBDIRS := src
all: $(SUBDIRS)

.PHONY : $(SUBDIRS)
$(SUBDIRS):
        $(MAKE) -C $@

.PHONY : clean
clean:
        @for i in $(SUBDIRS); do \
          $(MAKE) -C $${i} clean;  \
        done
```

Methods to calling Make Recursively

- Using .PHONY and $(MAKE) -C
- Using shell script for statement
  - Using the shell variable, `$$`
    First `$` is used to escape from Make

Suffix Rules

- Old fashioned Method

```
.c.o:
        <command>
```

`From .c file to .o file using command`

- Pattern rules

```
%.o: %.c
        <command>
```

`.o file from .c file using command`

- Suffix rules are controlled through .SUFFIXS

  .SUFFIXS .c .o . . .

- Conditional directives `ifeq`, `ifneq`, `ifdef`, `ifndef`

  <conditional directive>
  text-if-true
  else
  text-if-false
  <endif>

- Use of functions e.g
  `patsubst` or `${var:suffix=replacement}`
  1. Functions for String Substitution and Analysis
     `subst`, `strip`, `findstring`, `filter`
  2. Functions for File Name e.g `dir` , `suffix`, `wildcard`
  3. Other functions e.g `foreach` , `origin`, `error`

Make shortcomings

Make power

- Make is easily manage a project
- Make can be used for Compilation, Installation, Making Documents

Make pitfalls

- Make is a very powerful framework and Generalized
- Makefile can be of different type for the same Framework
- Makefile needs to be different for compilation on different system
- As the need for more target grows, complexity of Makefile grows
- Makefiles can become a bottleneck, and a difficult process to Maintain and Modify

Make

Power of Make

- Developer did not want to sacrifice the power of Make
- Solution was to provide a framework over Make, and not to replace Make

By 1992 there were 4 Frameworks

1. Metaconfig - Still used for perl
2. configure - Was used for gcc
3. Imake - for X-Windows
4. Autoconf
   - 1994 Automake was added to Automake framework
   - 1996 Libtool was added
   - 1998 Tool was available for Windows which requires perl also to be installed, + Cygwin For Windows : Supported compilers GNU-gcc<any>, VC++

Autoconf Philosophy

Tasks for Autoconf

- Superior method than writing #ifdef for every system or having different makefile for different system
- Does system wide tests to check for common system parameters: e.g Location, Existence of function and System calls, availability of libraries etc. . .
- Better and easier method to managing a project
- Standardize use of Makefile
- Expandable framework
- Generalized Library framework
- Generalized internationalization Support

Primary tools

- autoconf - Generates configure scripts
- automake - Generates Makefile.in
- autoheader - generates config.h
- aclocal - Useful for adding third party macros
- libtool - General purpose library making tool
- libtoolize - Integrates libtool with Autoconf framework
- autopoint - Gettext framework

Helper tools

- autoreconf - Combined framework
- autoscan - Auto generation of configure scripts
- autoupdate - Updates older configure.in to newer framework
- ifnames - Extracts CPP conditions from given set of files

Autoconf dependencies

Other Tools

1. M4 macro language. This language is more powerful than CPP
   Used on only developer machine
2. Perl and Cygwin shell in case of Windows machine
   Used on only developer machine
3. Shell(sh) script support The output of Autoconf is a shell script, that does not require Autoconf framework to be installed
4. Make support
5. All the tools and Utilities needed to compile the Sources . . .

Example

Using the Earlier example: hello, bye

```
Output Binary Programs:  hello, bye
Source Files:  hello.c, bye.c, msg.c
header Files:  msg.h in directory include
```

Makefile.am

Makefile.am

```
EXTRA_DIST = include
SUBDIRS = src
```

src/Makefile.am

```
INCLUDES = -I$(top_srcdir)/include
bin_PROGRAMS = hello bye
hello_SOURCES = hello.c msg.c
bye_SOURCES = bye.c msg.c
```

Making configure

Run autoscan : Output produced is **configure.scan**

configure.scan

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.57)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([src/bye.c])
AC_CONFIG_HEADER([config.h])
# Checks for programs.

AC_PROG_CC
# Checks for libraries.
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
```

cp configure.scan configure.ac and do changes

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.57)
AC_INIT(hello,0.0.1)
AM_INIT_AUTOMAKE(hello,0.0.1)
AC_CONFIG_SRCDIR([src/bye.c])
#AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CC
AC_PROG_INSTALL
# Checks for libraries.
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
```

Making configure

$ autoreconf -i

autoreconf output

```
configure.ac: installing `./install-sh'
configure.ac: installing `./mkinstalldirs'
configure.ac: installing `./missing'
Makefile.am: installing `./INSTALL'
Makefile.am: required file `./NEWS' not found
Makefile.am: required file `./README' not found
Makefile.am: installing `./COPYING'
Makefile.am: required file `./AUTHORS' not found
Makefile.am: required file `./ChangeLog' not found
src/Makefile.am: installing `./depcomp'
```

Create other file

$ touch **NEWS README AUTHORS ChangeLog**

Running Configure

Using configure

$ ./configure –help

Will present you with a list of options

$ ./configure –prefix=/tmp/hello

$ make && make install && make distcheck

make distcheck will make a distribution by the name ***hello-0.0.1.tar.gz***

```
This is a basic framework
```

Making Libraries

```
$ mkdir lib
$ mv src/msg.c lib
```

Adding Libraries

Modification to configure.ac and Makefile.am

Add : AC_PROG_LIBTOOL
AC_CONFIG_FILES add lib/Makefile

Makefile.am
Add: **lib** to SUBDIRS
src/Makefile.am
Remove : **msg.c**
Add:

```
hello_LDADD = $(top_builddir)/lib/libmsg.la
bye_LDADD = $(top_builddir)/lib/libmsg.la
```

lib/Makefile.am

```
lib_LTLIBRARIES = libmsg.la
libmsg_la_SOURCES = msg.c
INCLUDES = -I$(top_srcdir)/include
```

autoreconf -i –force

./configure

make

make distcheck

Install, Build, Source

Autoconf makes distinction between the Install, Build, Src. So it is possible to run ./configure from a separate directory. Some distribution make it mandatory e.g gcc

$ mkdir test

$ cd test

$ ../hello-0.0.1/configure –prefix=/tmp/hello

$ make

$ make install