目录

简单的makefile

```
CROSS_COMPILE=/opt/4.5.1/bin/arm-linux-

CC=$(CROSS_COMPILE)gcc
AS=$(CROSS_COMPILE)as
LD=$(CROSS_COMPILE)ld

CFLAGS=-g -Wall
LIBS=-lpthread

all:main

main:main.o gsm_gprs.o socket.o telosb.o wifi.o
    $(CC) $(CFLAGS) $(LIBS) $^ -o $@

main.o: main.c gsm_gprs.h option.h telosb.h
    $(CC) $(CFLAGS) -c $<

gsm_gprs.o:gsm_gprs.c gsm_gprs.h socket.h
    $(CC) $(CFLAGS) -c $<

socket.o:socket.c socket.h option.h
    $(CC) $(CFLAGS) -c $<

telosb.o: telosb.c telosb.h option.h
    $(CC) $(CFLAGS) -c $<

wifi.o: wifi.c wifi.h option.h
    $(CC) $(CFLAGS) -c $<

clean:
    -rm main -f *\.o *\*~ *~
```

makefile赋值

赋值	说明
=	基本的赋值 会在makefile的最后才赋值
:=	覆盖之前的值 会立即赋值
?=	如果没有赋值过就赋值
+=	添加后面的值

.PHONY : clean

伪目标

make命令默认支持的文件名

> make指令如果没有指定具体的makefile文件，就会自动寻找如下的makefile文件

GNUmakefile　makefile　Makefile

include

`makefile`包含

-include

`makefile`包含，当`include`过程中出现错误，不报错继续执行

MAKEFILES

`make`会自动`include`这个环境变量中的值

VPATH

指定`makefile`文件搜寻路径

vpath

`make` 关键词 设置文件搜寻路径

Makefile内建函数

文本处理和分析函数

替换

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

模式替换

> 可用%（只用第一个%有用），如 `$(patsubst %.c,%.o,x.c.c bar.c)`，结果 `‘x.c.o bar.o’`

```
$(patsubst pattern,replacement,text)
```

去掉文本两端空格，以及把2个和2个以上的空格换成一个

```
$(strip string)
```

查找

```
$(findstring find,in)
```

过滤

> 只保留pattern部分

```
$(filter pattern…,text)
```

过滤掉

> 不保留pattern部分

```
$(filter-out pattern…,text)
```

排序

```
$(sort list)
```

取字符串

```
$(word n,text)
```

取字符串列表

> 第s(start)个到第e（end ）个

```
$(wordlist s,e,text)
# $(words text) Returns the number of words in text. Thus, the last word of text
is $(word $(words text),text).
```

取第一个

```
$(firstword names…)
```

取最后一个

```
$(lastword names…)
```

文件名处理函数

取目录

```
$(dir names…)
```

取文件

> 但并不完全正确，注意观察，因为这个原理是已斜杠“/”为标识符的，如果文件名中包含斜杠，则返回的文件名就有误

```
$(notdir names…)
```

取文件后缀

```
$(suffix names…)
```

取文件名

> 包括前面的目录部分，如`$(basename src/foo.c src-1.0/bar hacks)`， 结果为`src/foo src-1.0/bar hacks`

```
$(basename names…)
```

添加后缀

> example ：`$(addprefix src/,foo bar)`， produces the result ‘src/foo src/bar’

```
$(addsuffix suffix,names…)
```

连接函数

> example: ‘$(join a b,.c .o)’ produces ‘a.c b.o’.

```
$(join list1,list2)
```

通配符函数

> 表示可以使用正则表达式的符号。The argument pattern is a file name pattern, typically containing wildcard characters (as in shell file name patterns). The result of wildcard is a space-separated list of the names of existing files that match the pattern

```
$(wildcard pattern)
```

真实路径

```
$(realpath names…)
```

绝对路径

```
$(abspath names…)
```

foreach函数

`$(foreach var,list,text)`相当于for循环函数，不过最终这里返回的是text的值，这个值是循环得到的一个list，如

```
find_files = $(wildcard $(dir)/*) #“=”等号是延时加载（deferred）
dirs := a b c d
files := $(foreach dir,$(dirs),$(find_files))
```

即

```
files := $(wildcard a/* b/* c/* d/*)
```

if函数

```
ifeq (arg1, arg2)
ifneq (arg1, arg2)
ifdef variable-name
ifndef variable-name
```

call函数

```
$(call VARIABLE,PARAM,PARAM,...)
```

“call”函数是唯一一个可以创建定制化参数函数的引用函数。使用这个函数可以 实现对用户自己定义函数引用。我们可以将一个变量定义为一个复杂的表达式，用“call” 函数根据不同的参数对它进行展开来获得不同的结果。

如：`reverse = $(2) $(1) foo = $(call reverse,a,b)` foo will contain ‘b a

value函数

```
$(value variable)
```

The result of this function is a string containing the value of variable, without any expansion occurring. For example, in this makefile:

```
FOO = $PATH
all:
    @echo $(FOO)
    @echo $(value FOO)
```

The first output line would be ATH, since the `$P` would be expanded as a make variable, while the second output line would be the current value of your $PATH environment variable, since the value function avoided the expansion.

realpath

```
$(realpath ../../)
```

获取绝对路径

wildcard

根据通配符获取列表

```
src = $(wildcard *.c)
```

origin函数

```
$(origin variable)
```

获取变量的属性值，如下几个

1. undefined 变量“VARIABLE”没有被定义。
2. default 变量“VARIABLE”是一个默认定义（内嵌变量）。如“CC”、“MAKE”、“RM”等变 量。如果在 Makefile 中重新定义这些变量，函数返回值将相应发生变化
3. environment 变量“VARIABLE”是一个系统环境变量，并且 make 没有使用命令行选项“-e” （Makefile 中不存在同名的变量定义，此变量没有被替代）。
4. environment override 变量“VARIABLE”是一个系统环境变量，并且 make 使用了命令行选项“-e”。Makefile 中存在一个同名的变量定义，使用“make -e”时环境变量值替代了文 件中的变量定义。
5. file 变量“VARIABLE”在某一个 makefile 文件中定义。
6. command line 变量“VARIABLE”在命令行中定义。
7. override 变量“VARIABLE”在 makefile 文件中定义并使用“override”指示符声明。
8. automatic 变量“VARIABLE”是自动化变量。

shell函数

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

make LOG以及控制函数

```
$(info text)    #打印log
$(warning text) #和 error 一样，但是 产生致命错误退出
$(error text)   #产生致命错误，并提示“text”信息给用户，并退出 make 的执行
```