基础2-U-boot主Makefile分析

主Makefile位于uboot源码的根目录下，其内容主要结构为：
1. 确定版本号及主机信息
2. 实现静默编译功能
3. 设置各种路径
4. 设置编译工具链
5. 设置规则
6. 设置与cpu相关的伪目标

需要注意的是，结构顺序不代表代码执行顺序，关于代码的执行顺序以及推荐阅读顺序请移步[U-boot配置及编译阶段流程宏观分析]

1.确定版本号及主机信息

```
                $(VERSION) $(PATCHLEVEL) $(SUBLEVEL)$(EXTRAVERSION)
              $(obj)include/version_autogenerated.h
```

- 这四个变量的含义依次为：主版本号、次版本号、再次版本号、附加的版本信息（值为空，可以给用户使用）
- U-Boot_VERSION这个变量的值为真正的uboot版本号，易知其值为1.3.4
- VERSION_FILE变量的值是一个.h文件，注意=是并行赋值的意思，类似于Verilog语言中的<=。要注意:=（可以理解为串行赋值）和=的区别。$(obj)这个变量是在后面定义并赋值的，其值是编译输出路径
- version_autogenerated.h这个文件是在make之后自动生成的，文件内容是一条宏，这条宏给将其他.c文件提供uboot的版本号

```
HOSTARCH := $(shell uname -m | \
    sed -e s/i.86/i386/ \
        -e s/sun4u/sparc64/ \
        -e s/arm.*/arm/ \
        -e s/sa110/arm/ \
        -e s/powerpc/ppc/ \
        -e s/ppc64/ppc/ \
        -e s/macppc/ppc/)
```

- makefile的函数调用与变量调用很类似，格式是$(function arguments)，其实上面一大段的意思是变量HOSTARCH的值是一个函数的返回值
- shell是makefile中的一个函数，$(shell XXX)会被解析成执行shell命令XXX；此处是执行了一条 uname -m |sed -e ……，符号 \是makefile的换行符
- 其中，|是shell语法中的管道结构，例如：XXX | YYY ，表达式XXX 的输出将作为表达式YYY的输入，YYY的输出才是整句表达式的输出
- uname -m 指令将输出负责编译的主机cpu架构，比如ixx86；sed -e是替换命令，比如把ixx86替换为i386
- 由此可见这个HOSTARCH变量的值将得到负责编译的主机cpu架构。大部分情况下我们得到的都是i386

```
HOSTOS := $(shell uname -s | tr '[:upper:]''[:lower:]' | \
        sed -e 's/\(cygwin\).*/cygwin/')
```

- 这个HOSTOS变量和上一句HOSTARCH变量的原理类似，管道第一部分uname -s会得到负责编译的主机的OS，比如Linux
- 管道第二部分是将大写转换成小写
- 管道第三部分的意思是如果前面一个部分得到了cygwin系统，则格式要转换一下。不必深究，因为cygwin基本没人用……
- 由此可见这个HOSTOS变量的值将得到负责编译的主机操作系统，大部分情况下我们得到的都是linux

```
export  HOSTARCH HOSTOS
```

- 导出上面两个变量到全局，使其为环境变量，让其他的文件也可以使用架构和系统信息

2.实现静默编译功能

```
# Allow for silent builds
ifeq (,$(findstring s,$(MAKEFLAGS)))
XECHO = echo
else
XECHO = :
endif
```

- 这整段是为了实现make的静默选项功能，其中，findstring一个函数，$(findstrings, $(MAKEFLAGS))功能是从$(MAKEFLAGS)中找出字符‘s’
- $(MAKEFLAGS)是make的flag（选项），如果在控制台中输入make -s，则$(MAKEFLAGS)的值为‘s’

- 如果$(findstring $(MAKEFLAGS))没找到‘s’，这个表达式的值为空，则ifeq（）为真，即make时无需静默
- 无需静默的实现方法是令变量XECHO值为关键字echo，因为makefile中每次需要打印都会使用XECHO，而不是直接使用echo本身
- 静默make的实现方法是令变量XECHO值为空，当makefile需要打印时会调用XECHO，由于其为空，故无法打印make信息
- 注：编译工具链的信息是永远打印的，和make的静默选项无关

3.设置各种路径

- 这里开始阐述了makefile所支持的“单独外部路径编译”，它的原理和keil把.o、.l、.bin、.a等中间文件放在单独的文件夹内的原理相同，都是为了使源码更简洁明了，防止被中间文件污染；以及为了同时维护多个配置编译方式
- 若要使用单独外部路径编译，有两种方法，方法一：例如在控制台中输入 make O=/tmp/build ，将输出文件夹路径作为参数
- 方法二：导出环境变量，即export BUILD_DIR=/tmp/build
- 注：方法一的优先级高，会覆盖方法二。而且方法一必须每次输入make时都要输入参数（不论是make clean还是make config 的时候），格式如make O=/tmp/build disclean

```
ifdef O
ifeq ("$(origin O)", "command line")
BUILD_DIR := $(O)
endif
endif

ifneq ($(BUILD_DIR),)
saved-output := $(BUILD_DIR)
```

- 上面这段是方法一的具体实现，如果make时输入参数 O=/tmp/build，那么makefile会认为定义了变量O，于是乎这段代码会开始执行
- 其中，$(origin O)中origin是函数名，$(origin O)的功能是返回变量O的来源；由此可知，如果O的来源是控制台命令，则变量BUILD_DIR的值就是变量O的值
- 然后进行一个判断，如果变量BUILD_DIR不为空，则变量saved-output的值为BUILD_DIR，即saved-output的值也是输出文件夹路径

```
# Attempt to create a output directory.
$(shell [ -d ${BUILD_DIR} ] || mkdir -p ${BUILD_DIR})
```

- 上面这句是shell语法中简写的if表达式，其作用是当输出文件夹路径不存在时就创建它
- 其中包含两个表达式，表达式1||表达式2，当表达式1为真时，表达式2不会被解释器执行，因为总结果一定为真，（尽管总的结果没有意义）。唯有表达式1为假时，解释器才会去执行表达式2，这样就能用逻辑表达式来实现if语句的功能了
- 表达式1中，方括号是固定用法，是为了突出里面表达式的作用是判断语句。-p是shell中判断路径是否存在的符号，如果路径存在则为表达式1为真；如果路径不存在，表达式1为假，解释器会执行表达式2，即创建输出文件夹路径。

```
# Verify if it was successful.
BUILD_DIR := $(shell cd $(BUILD_DIR) && /bin/pwd)
$(if $(BUILD_DIR),,$(error output directory "$(saved-output)" does not exist))
endif # ifneq ($(BUILD_DIR),)
```

- 这整个一段功能是确保输出文件夹路径创建成功，&&的功能是连续执行两句语句，当cd $(BUILD_DIR)执行完后，执行/bin/pwd，即打印当前路径；
- $(if xxx，yyy，zzz）是makefile的判断函数，如果xxx为真，则执行yyy并返回值，否则执行zzz并返回值，由此可知如果未成功创建BUILD_DIR，就会输出错误打印信息；
- 最后一句应该是被注释掉了……

```
OBJTREE     := $(if $(BUILD_DIR),$(BUILD_DIR),$(CURDIR))
SRCTREE     := $(CURDIR)
TOPDIR      := $(SRCTREE)
LNDIR       := $(OBJTREE)
export  TOPDIR SRCTREE OBJTREE
```

- 这段代码是设置并导出了很多和路径有关的环境变量
- 如果BUILD_DIR不为空，OBJTREE（放产生的.o文件）的值就为BUILD_DIR，如果为空，则OBJTREE的值就为CURDIR（即current direction，当前源码所在的目录）
- SRCTREE的值为当前源码所在目录，TOPDIR的值为SRCTREE的值即当前源码所在目录，LNDIR（应该是放链接产生的文件）的值和OBJTREE相同

```
MKCONFIG    := $(SRCTREE)/mkconfig
export MKCONFIG
```

- 将变量MKCONFIG的值设置为当前源码目录下的mkconfig文件，并将其导出为环境变量,这个shell脚本文件是正式make之前的配置脚本，十分重要

```
ifneq ($(OBJTREE),$(SRCTREE))
REMOTE_BUILD    := 1
export REMOTE_BUILD
endif
```

- 判断OBJTREE的值和SRCTREE的值是否不相等，即判断是否使用了“单独外部路径编译”，如果使用了这个功能，则REMOTE_BUILD值为1，并将其导出至全局

```
# $(obj) and (src) are defined in config.mk but here in main Makefile
# we also need them before config.mk is included which is the casefor
# some targets like unconfig, clean, clobber, distclean, etc.
ifneq ($(OBJTREE),$(SRCTREE))
obj := $(OBJTREE)/
src := $(SRCTREE)/
else
obj :=
```

```
src :=
endif
export obj src
```

- 这一段的意思非常简单，判断OBJTREE的值和SRCTREE的值是否不相等，即判断是否使用了“单独外部路径编译”功能，如果使用了这个功能， 则将obj的值赋为OBJTREE的值，将src的值赋为SRCTREE的值；反之，值都为空。最后将他们全部导出至全局

4.设置编译工具链（大部分在config.mk内）

```
ifeq ($(ARCH),powerpc)
ARCH = ppc
endif

ifeq ($(obj)include/config.mk,$(wildcard $(obj)include/config.mk))
```

- 开头三句是powerpc架构的名称转换。
- 最后一句$(wildcard xxx) 参数xxx是一个文件名格式(可使用通配符)，这个函数的返回值是一列和格式匹配且真实存在的文件的名称。但是这句应该没什么意义，因为连endif都没有……

```
# load ARCH, BOARD, and CPU configuration

include$(obj)include/config.mk
export  ARCH CPU BOARD VENDOR SOC
```

- config.mk这个文件其实uboot源码中是不存在的，它是由配置过程中由mkconfig这个脚本创建的， 也就是make之前的一步——make x210_sd_config（这个目标在2600多行），它会去执行根目录下mkconfig这个脚本，脚本中将创建include/config.mk并将参数填充进去。
- config.mk的内容分别定义了ARCH CPU BOARD VENDOR SOC这几个变量的值，通过把这个.mk文件include进来，其内容将在本makefile中原地展开
- 本Makefile得到这几个变量值后再将它们导出到环境变量

```
ifndef CROSS_COMPILE
ifeq ($(HOSTARCH),$(ARCH))
CROSS_COMPILE =
else
ifeq ($(ARCH),ppc)
CROSS_COMPILE = ppc_8xx-
endif
ifeq ($(ARCH),arm)
#CROSS_COMPILE = arm-linux-#CROSS_COMPILE = /usr/local/arm/4.4.1-eabi-cortex-a8/usr/bin/arm-linux-#CROSS_COMPILE = /usr/local/arm/4.2.2-eabi/usr/bin/arm-linux-
CROSS_COMPILE = /usr/local/arm/arm-2009q3/bin/arm-none-linux-gnueabi-
endif
ifeq ($(ARCH),i386)
CROSS_COMPILE = i386-linux-
endif
ifeq ($(ARCH),mips)
CROSS_COMPILE = mips_4KC-
endif
#......中间略去

endif   # sparc
endif   # HOSTARCH,ARCH
endif   # CROSS_COMPILE

export  CROSS_COMPILE
```

- 这上面一整段是配置交叉编译工具链的前缀，即arm-none-linux-gnueabi- ，一旦确定了前缀，加上后缀就能定义在编译过程中用到的各种工具，如ar、gcc等；
- 前缀的选择是通过前面获得的变量ARCH（目标cpu的架构）来判断的

```
# load other configuration

include$(TOPDIR)/config.mk
```

- 这里包含了一个源码目录下的config.mk文件，和之前的那个x210_sd_config生成的include/config.mk不同，这个.mk文件是源码自带的，对编译属性和链接属性进行了很多设置
- 这个位于根目录的config.mk被include进来，将在原地展开，但是由于代码量较大，故其注释写在了config.mk里面，请移步 [U-boot根目录下的config.mk详尽分析]

5.设置规则

```
OBJS  = cpu/$(CPU)/start.o
ifeq ($(CPU),i386)
OBJS += cpu/$(CPU)/start16.o
OBJS += cpu/$(CPU)/reset.o
endif
ifeq ($(CPU),ppc4xx)
OBJS += cpu/$(CPU)/resetvec.o
endif
```

```
ifeq ($(CPU),mpc85xx)
OBJS += cpu/$(CPU)/resetvec.o
endif

OBJS := $(addprefix $(obj),$(OBJS))
```

- 这里将众多.o文件赋给了OBJS变量，这个变量会成为后面目标u-boot的依赖
- 其中用到了多次+=赋值符号，此赋值符号的含义其实是在变量原本的值后面在续上新的值，就是额外添加的意思

```
LIBS  = lib_generic/libgeneric.a
LIBS += $(shell if [ -f board/$(VENDOR)/common/Makefile ]; then echo \
    "board/$(VENDOR)/common/lib$(VENDOR).a"; fi)
LIBS += cpu/$(CPU)/lib$(CPU).a
ifdef SOC
LIBS += cpu/$(CPU)/$(SOC)/lib$(SOC).a
endif
ifeq ($(CPU),ixp)
LIBS += cpu/ixp/npe/libnpe.a
endif
LIBS += lib_$(ARCH)/lib$(ARCH).a
LIBS += fs/cramfs/libcramfs.a fs/fat/libfat.a fs/fdos/libfdos.a fs/jffs2/libjffs2.a \
    fs/reiserfs/libreiserfs.a fs/ext2/libext2fs.a
LIBS += net/libnet.a
LIBS += disk/libdisk.a
LIBS += drivers/bios_emulator/libatibiosemu.a
#......中间略去
ifeq ($(CPU),mpc83xx)
LIBS += drivers/qe/qe.a
endif
ifeq ($(CPU),mpc85xx)
LIBS += drivers/qe/qe.a
endif
LIBS += drivers/rtc/librtc.a
#......中间略去
LIBS += post/libpost.a

LIBS := $(addprefix $(obj),$(LIBS))
.PHONY : $(LIBS) $(VERSION_FILE)

LIBBOARD = board/$(BOARDDIR)/lib$(BOARD).a
LIBBOARD := $(addprefix $(obj),$(LIBBOARD))

# Add GCC lib
PLATFORM_LIBS += -L $(shell dirname `$(CC)$(CFLAGS) -print-libgcc-file-name`) -lgcc
```

- 将众多.a库文件赋给了LIBS变量，以及将和板子有关的.a文件赋给了LIBBOARD变量，这两个变量会成为后面目标u-boot的依赖

```
SUBDIRS = tools \
      examples \
      api_examples

.PHONY : $(SUBDIRS)

ifeq ($(CONFIG_NAND_U_BOOT),y)
NAND_SPL = nand_spl
U_BOOT_NAND = $(obj)u-boot-nand.bin
endif

ifeq ($(CONFIG_ONENAND_U_BOOT),y)
ONENAND_IPL = onenand_bl1
U_BOOT_ONENAND = $(obj)u-boot-onenand.bin
endif

__OBJS := $(subst $(obj),,$(OBJS))
__LIBS := $(subst $(obj),,$(LIBS)) $(subst $(obj),,$(LIBBOARD))
```

- 根据autoconf.mk中nand和onenand的CONFIG做了一个判断，是否使用相应的.bin文件

```
ALL += $(obj)u-boot.srec $(obj)u-boot.bin $(obj)System.map $(U_BOOT_NAND) $(U_BOOT_ONENAND) $(obj)u-boot.
dis
ifeq ($(ARCH),blackfin)
ALL += $(obj)u-boot.ldr
endif

all:$(ALL)
```

- 本段出现了顶层makefile的第一条规则（即目标-依赖-操作），故默认情况下将以all这个变量作为最终目标。但是由于本条规则没有任何操作， 所以一旦把依赖（也就是$(ALL)）实现了，整个makefile文件的最终目标就达成了
- 可以认为，本makefile的终极目标其实是$(ALL)所代表的那些文件

```
$(obj)u-boot.hex:$(obj)u-boot
        $(OBJCOPY) ${OBJCFLAGS} -O ihex $<$@$(obj)u-boot.srec:$(obj)u-boot
        $(OBJCOPY) ${OBJCFLAGS} -O srec $<$@$(obj)u-boot.bin:$(obj)u-boot
        $(OBJCOPY) ${OBJCFLAGS} -O binary $<$@$(obj)u-boot.ldr:$(obj)u-boot
        $(LDR) -T $(CONFIG_BFIN_CPU) -f -c $@$<$(LDR_FLAGS)
```

```
$(obj)u-boot.ldr.hex:$(obj)u-boot.ldr
        $(OBJCOPY) ${OBJCFLAGS} -O ihex $<$@ -I binary

$(obj)u-boot.ldr.srec:$(obj)u-boot.ldr
        $(OBJCOPY) ${OBJCFLAGS} -O srec $<$@ -I binary

$(obj)u-boot.img:$(obj)u-boot.bin
        ./tools/mkimage -A $(ARCH) -T firmware -C none \
        -a $(TEXT_BASE) -e 0 \
        -n $(shell sed -n -e 's/.*U_BOOT_VERSION//p'$(VERSION_FILE) | \
            sed -e 's/"[     ]*$$/ for $(BOARD) board"/') \
        -d $<$@$(obj)u-boot.sha1:$(obj)u-boot.bin


#             后面还有一大堆繁琐但不是很重要的代码，就不贴了
```

- 这些都是为了产生最终文件的规则

```
$(obj)include/autoconf.mk.dep:$(obj)include/config.h include/common.h
    @$(XECHO) Generating $@ ; \
    set -e ; \
    : Generate the dependancies ; \
    $(CC) -x c -DDO_DEPS_ONLY -M $(HOST_CFLAGS) $(CPPFLAGS) \
        -MQ $(obj)include/autoconf.mk include/common.h > $@$(obj)include/autoconf.mk:$(obj)include/config
.h
    @$(XECHO) Generating $@ ; \
    set -e ; \
    : Extract the config macros ; \
    $(CPP) $(CFLAGS) -DDO_DEPS_ONLY -dM include/common.h | \
        sed -n -f tools/scripts/define2mk.sed > $@

sinclude $(obj)include/autoconf.mk.dep
```

- 本段的功能是根据include/configs/x210_sd.h来生成autoconf.mk，makefile利用这些autoconf.mk中的变量来指导编译过程的走向（条件编译）

6.设置与cpu相关的伪目标

```
#由于这些代码都与cpu本身有关，有2000多行，且功能重复，故这里挑选我们板子上的s5pv210为例子来分析，这行代码大概位于2600多行。

x210_sd_config :    unconfig
    @$(MKCONFIG) $(@:_config=) arm s5pc11x x210 samsung s5pc110@echo"TEXT_BASE = 0xc3e00000" > $(obj)boar
d/samsung/x210/config.mk
```

- 本段代码主要负责了正式make之前的配置过程，即在控制台输入“make x210_sd_config”，其依赖是unconfig，此变量代表了未配置的意思，通过这个方法，我们便可以多次配置。MKCONFIG这个变量代表的是源码目录下最关键的一个shell文件即mkconfig，这个shell文件负责了make之前的配置过程。
- 行首的@代表静默执行，这段代码在执行$(MKCONFIG)前还把$(@:_config=)、arm、s5pc11x、x210、samsung、s5pc110这6个主要参数传给了mkconfig
- 其中参数$(@:_config=)是引用了一个替换函数，将该规则中的目标（用@表示）中的_config用空替换，故$(@:_config=)的值为x210_sd
- 关于mkconfig，请移步 [U-boot根目录下的mkconfig详尽分析]
- 最后把TEXT_BASE的值填充入config.mk文件，指定uboot的虚拟链接地址，完成所有配置工作